Eruba Ekedayen and William Sun

Design critique

The code that we received was overall designed well. The code has properly separated functionality around handling the model, GUI, operations, controller, and more. Furthermore, these functionalities work off of each other in a proper manner. Some minor critique we would offer is grouping the different functionality together in packages so the code is a bit more organized. For example, the only class in the actual controller package is ImageController, while GUIController is in its own package. Furthermore, we would also advise creating a GUIUtil which specifically handles different commands within the GUI instead of handling all of the GUIUtil functionality in the ImageProcessor class.

Implementation critique

The code was implemented in a very unique way; however, we believe that the way the code was implemented is restrictive and would possibly hinder the addition of future features. Currently, all the red green blue values of every pixel are stored in a massive list of integers. Naturally, since the massive list of integers has dimensions which are multiples of three, the code crunches all of these numbers and then puts triplets of values read from a scanner into a map. Pixels are thus stored not as class objects, but rather triplets of integer red green blue values that are meticulously parsed. Furthermore, functionality such as sepia, luma, intensity, and value are done via a massive switch. All of these pixel-oriented things are done separately without much interaction, and we believe this could've been done a lot easier had the creator of the code just created a Pixel object.

The critique we would offer is instead of meticulously parsing and working with a bunch of these values stored in a list that represent different aspects of every pixel such as their red green and blue values, you could just create a Pixel object to handle the storing. Thus, we would have defined an image to be a 2d array of Pixel objects. This would make implementing future features a lot easier, because instead of meticulously creating a whole new list of random objects to represent certain traits of all the pixels, you could just add another field in the class. Finally, this is just a minor thing, but some of the variable names and javadoc comments were done in a funny and nondescript manner. Some of the variables were named “bruh” and there were some questionable javadoc comments to describe some of the methods.

Documentation critique

Overall, the documentation is decent. It is clear that the creators of this code took time to make sure their documentation was thorough. There were very specific functionality updates for every new assignment, and the documentation is written in a way that is easy to understand and follow. The user also provided examples of how to use certain functionality, which was incredibly convenient. However, there is some misleading documentation that suggests certain functionality works when we don’t believe it actually does. We had problems with scripting and loading image files. After taking a closer look at the implementation, we can see the root of these problems in the ImageProcessor class. Overall though, the documentation was quite well done.

Design/code strengths, limitations, and suggestions on how to suggest them

A strength of the design and implementation of the code is that it was unique and avoided getter/setter methods. However, we believe that the current code design is quite restrictive due to the way the pixel attributes are parsed. For example, to parse the red, green, and blue values for a pixel, one has to go through a complex and meticulous journey of going through a massive list of values and selecting every three values. This is incredibly limiting for adding future functionality because any attribute for any pixel for any future functionality will have to be stored in a similar manner to a massive, nondescript list.

The way we would fix this implementation is by making the project more object-oriented. We would create a Pixel object and then store all of the attributes of a pixel within the object rather than in a massive list. Thus, it would be a lot easier to add future functionality because all we would have to do is create a new field within that pixel class. Additionally, the code would be a lot more readable to people who try to work on it in the future, because they would not have to follow a complex train of logic that extracts the attributes of each pixel from a massive list.

On top of adding a pixel object, we would refactor the model to run on a coordinate system created by doing math on the big list of integer values that stores each pixel's red green blue values. Refactoring the model to run on a coordinate system would simply the creation of a position class that documents the position of each object with its x and y position. We don't necessarily require a position class to find the distance from a pixel and a mosaic node for the mosaic method because you can technically do the math to calculate a pixel's position if given the amount of rows your image has; however, a position class would drastically simplify the

mosaic method implementation and make the code design more compatible for implementing future features.

Additionally, the way the GUI works in this implementation is that it pops up a text box that prompts a user input whenever the user asks for any values. For example, the GUI brighten command will pop a text box up whenever a user tries to run a brighten function. The issue with this is that it makes testing impossible because it is not possible to set the user input value while running a test to simulate the brighten command. One way to work around this is by not creating a pop up to accept user input and by creating a method that can manually set the value for a certain command input field for testing purposes only.