Design critique

1. Pixel: Pixel could have been an interface implementing the required functionalities. With the current design if we implement a new pixel having different channels than the RGB we would have to change in places like ImageModel which would lead to a lot of changes. This would be fixed by having an interface that would have made the extensibility a lot easier.
2. In the ImageImplementation class operations like Luma, Value, and Intensity, contain repeated for-loops (code duplication). Since such operations can be isolated so that you could design a singular "transformation" method that iterates through the entire pixel array and all we do is pass in the function to be applied on each pixel (using visitor design pattern), thus reducing for loops in each method to 1 nested loop with different functions passed as input.

Implementation critique

1. The GUIView has public methods such as displayImage, showGUI, getCurrentImage which is not present in the interface. This can lead to issues when we have to change the controller to use a different view instead of the Swing library since we are using the actual implementation of ImageProcessingGUI instead of the ImageProcessingView interface.

   The ideal way for this is to add the method in the interface, implement the same, and use the interface in different classes rather than the actual implementation. (As opposed to how it's done in ImageProcessingGUIController: 19)

2. The image processing controller has the context of swing operations like JOptionPane. This tightly coupled relationship between controller and view can pose an issue when we want to use another library instead of Swing, we would have to change the implementation of the controller along with the view.

   This can be resolved by giving all the UI context was given to the view since we would have to change only the view and not the controller.

3. Saving an image is the responsibility of controller but this is being done by the model. Moving it to the controller will adhere to true MVC principles.
4. The SplitRGB method updates channel fields and separate methods must be called to fetch data (e.g., RedComponent, GreenComponent), creating coupled methods. Since the underlying Pixel array is assigned using setters, if a new pixel array is assigned it would require re-calling the SplitRGB method before fetching channel data.

   Possible solutions: This method can return an object containing all three channels or create individual methods to fetch each of the three components individually or.

Code Documentation

The entire code is well documented with what the method or class does along with detail explanation on its individual methods. All the functions are defined in the model interface which acts as a blueprint to

understand the features supported by the program, if we want to add new features we can easily add a new method in this model interface and implement the same.

Design/code limitations

1. All controllers are initialized in the main method, which also includes logic for what parameters to pass to each controller (view, model, etc). This fails to adhere to the Single Responsibility Principle since the main method is also parsing the argument coming in and also worrying about different views and controllers to initialize. If we need to change an existing command line or add new arguments, it fails to adhere to Open Close Principle since we are now changing a piece of code which is already in production.

   One way to overcome this is to delegate the responsibility to an interface ArgumentParser which can have a method parseArgument taking in the arguments and returning the appropriate controller. This can be implemented by GUIArgumentParser, ConsoleArgumentParser and ScriptArgumentParser respectively. Doing so will split the parsing responsibility and is also easily extensible when we need to add new arguments.

2. The view is initializing the GUI controller on its constructor. This can lead to issues since the view can now access controller methods (as it is of controller type and not feature interface) that can directly change the model leading to unintended consequences. This is an issue when the project becomes big with multiple developers working it where they now get access to controller methods which the original author might have intended to not give. Inorder to initialise the controller the view also initializes model (and hence has access to it) which does not adhere to MVC principles.

   This can be fixed by the view exposing a feature listener and the controller at the time of initialising can register itself as a feature listener(call back) to the view. This will also ensure view is not initialising model for controller adhering to true MVC design where the view does not talk to the model.

Design/code strengths

1. With the current design it's really easy to implement new features as its clear from the code structure on how to add new methods to model interface and also call the same from the controller.
2. The view interface is cleanly designed with clear methods. This makes easier to add newer views like FileBasedViews without having to change controller or the model.