CSI-5-OOP Object-oriented Programming
Coursework assignment 2

Kamran Bhatti 3807942

Contents

Design

(Balsamiq Studios, LLC, 2008) was used to develop low fidelity designs for how the individual interfaces should look.

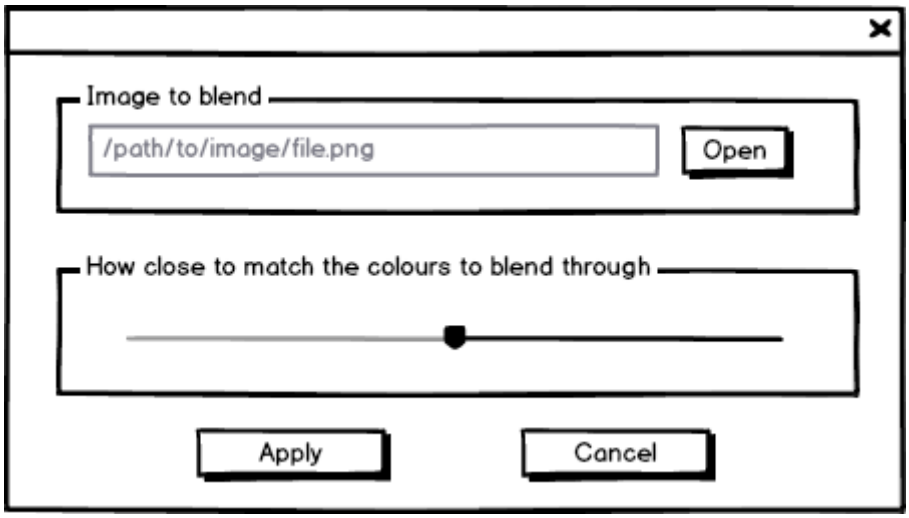

Figure 1 – BlendUI

Figure 1 – BlendUI represents the interface for the Blend operation where a user selects an image to blend with the currently loaded image via the open button, and then specify the intensity to blend the images through.

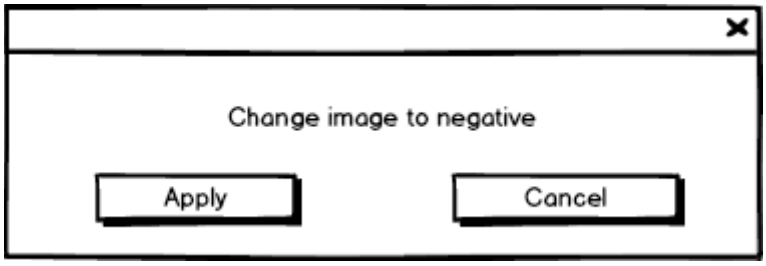

Figure 2 – NegativeUI

Figure 2 – NegativeUI depicts the interface for the Negative operation where a user chooses to apply the operation to the image. As the operation would only invert the colours of the loaded image, only the option to apply or cancel the operation would be necessary which is similar to the implementation of the grayscale interface

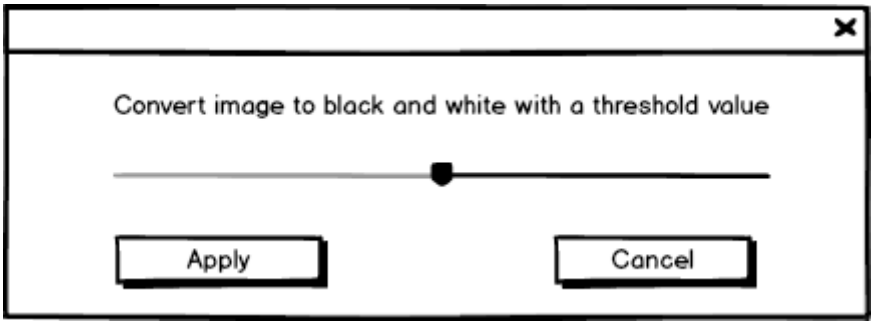

Figure 3 - ThresholdUI

Figure 3 - ThresholdUI depicts the interface for the Threshold operation where a user selects a threshold value, via the slider, to convert the image to black or white. This is similar to the interface for the tint operation however, this doesn't utilise an option to choose a colour to tint the image with.

Class diagram

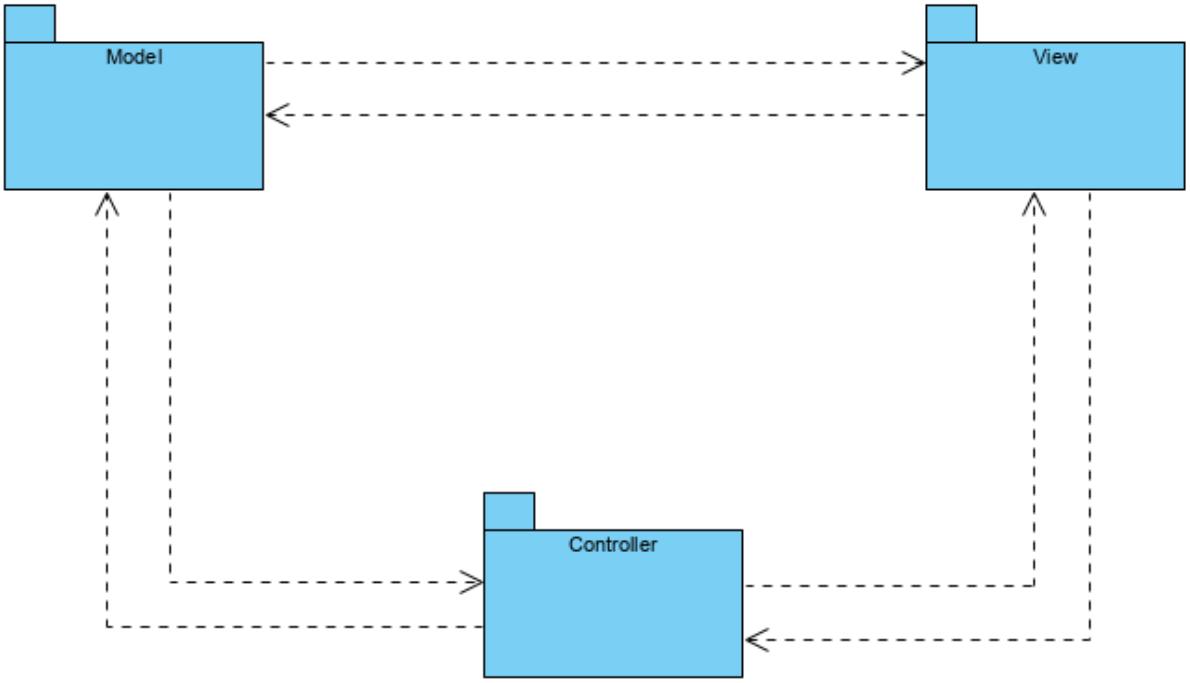

Figure 4

Although relationships between each class can be depicted separately, Figure 4 depicts a simplification of how each package relates to each other to reduce the complexity of the class diagram.

Controller package

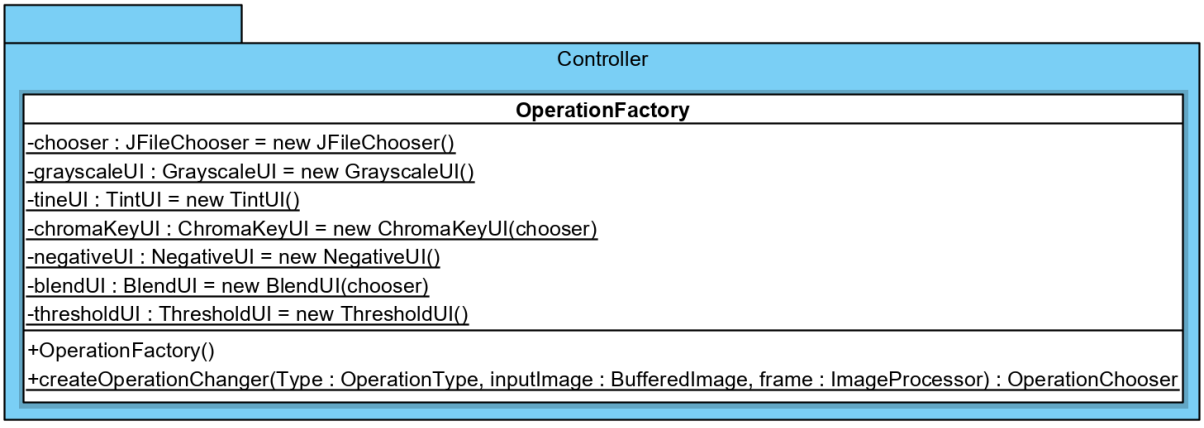

Figure 5

Figure 5 depicts the factory class *OperationFactory* which, similar to (Child, 2020), utilises a static method named *createOperationChanger* to compare the Enum type passed to the method with the Enum literals specified in *models.OperationType*, related to *OperationFactory* by a dependency relationship, to initialize a new object. However, the *createOperationChanger* method has additional arguments to accommodate for the *BufferedImage* and *ImageProcessor* objects. Static variables are used to store each interface for each operation.

Model package

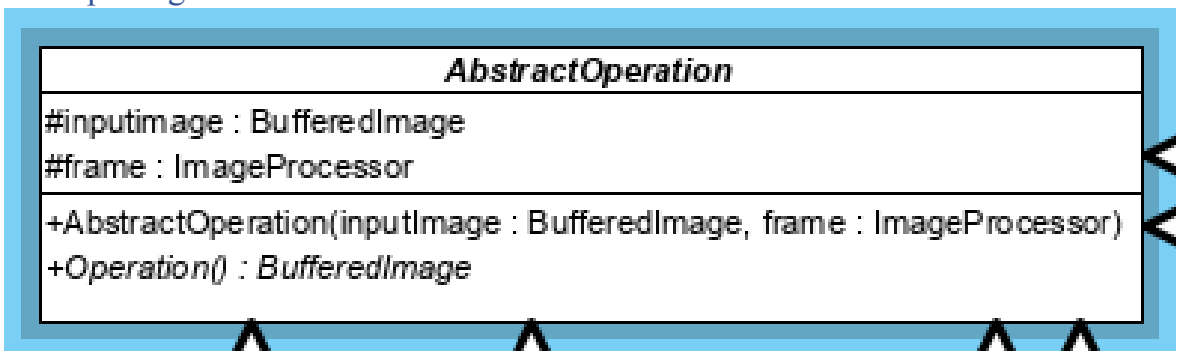

Figure 6

Figure 6 depicts *AbstractOperation*, an abstract class, which has a constructor that takes a *BufferedImage* and *ImageProcessor* object thus, all classes which extend *AbstractOperation* inherit *an Operation*, an abstract method.

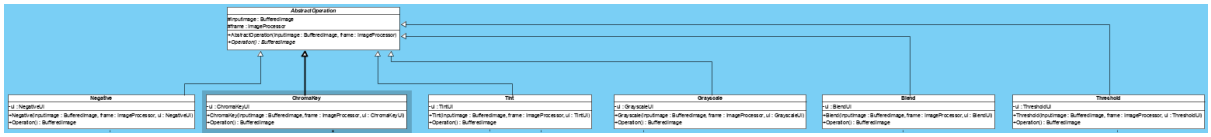

Figure 7

Figure 7 depicts how each operation in an individual class, where the name of each operation class isn't specified with "UI", extends the *AbstractOperation* abstract class thus inheriting the "Operation" method in the *OperationChooser* interface.

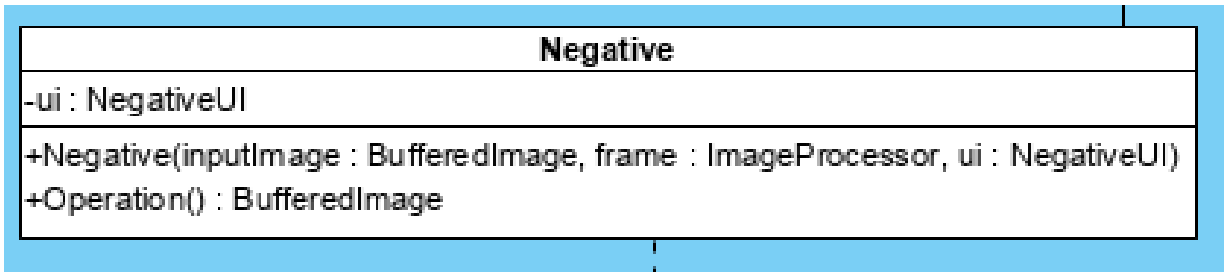

Figure 8

Figure 8 depicts the *Negative* class which has a dependency relationship to the view.*NegativeUI* class.

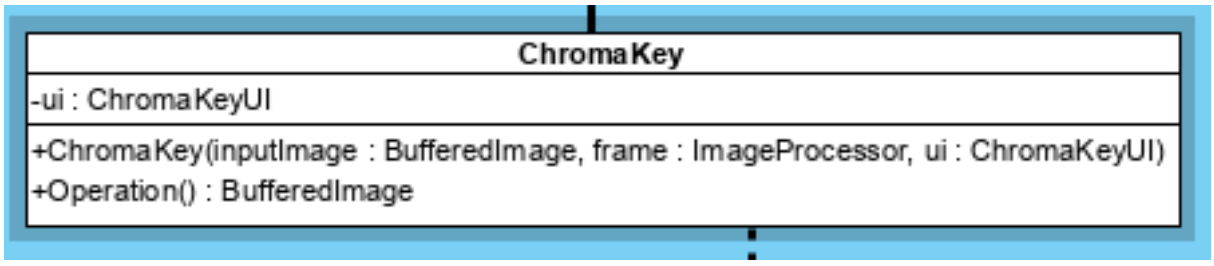

Figure 9

Figure 9 depicts the *ChromaKey* class which has a dependency relationship to the *view.ChromaKeyUI* class.

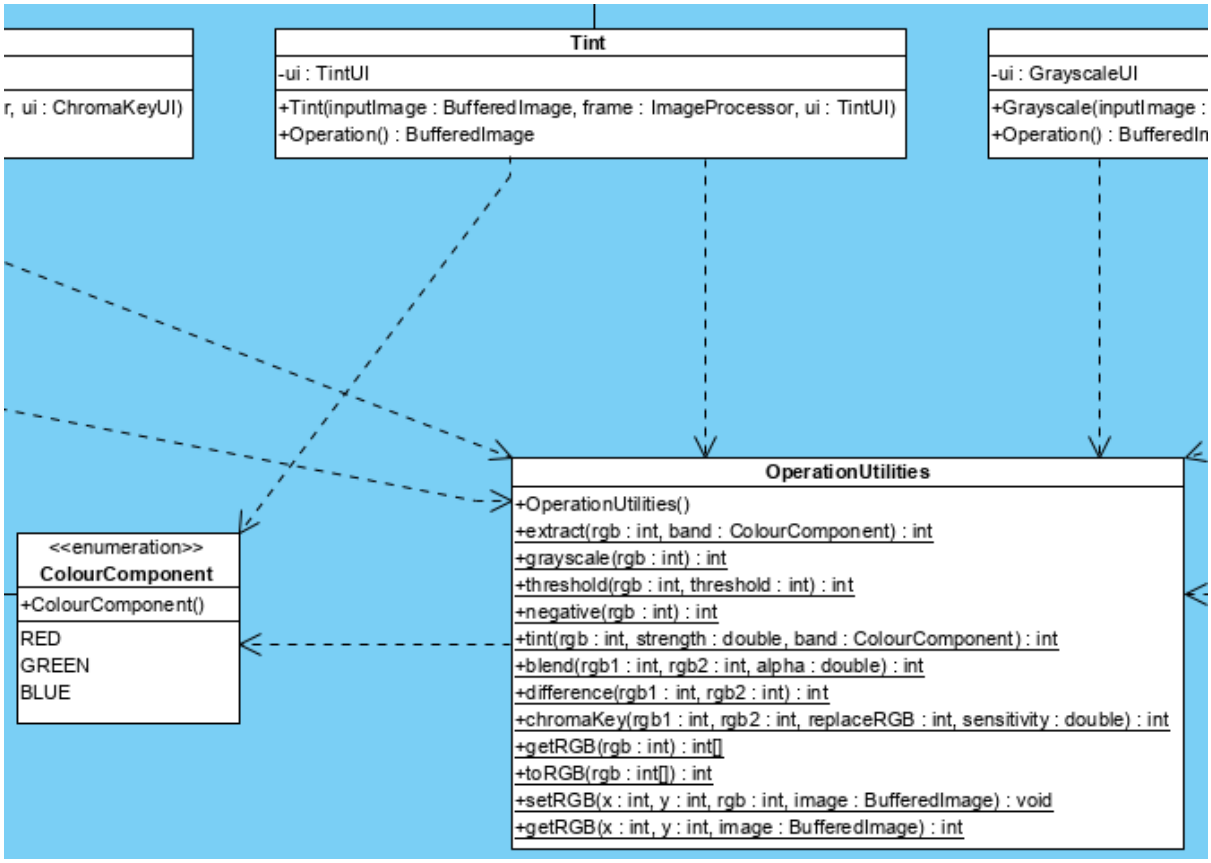

Figure 10

Figure 10 depicts the *Tint* class which has a composition relationship to the *view.TintUI* class which has a dependency relationship to the *ColourComponent* Enum object. All operation classes have a dependency relationship with *OperationUtilities.*

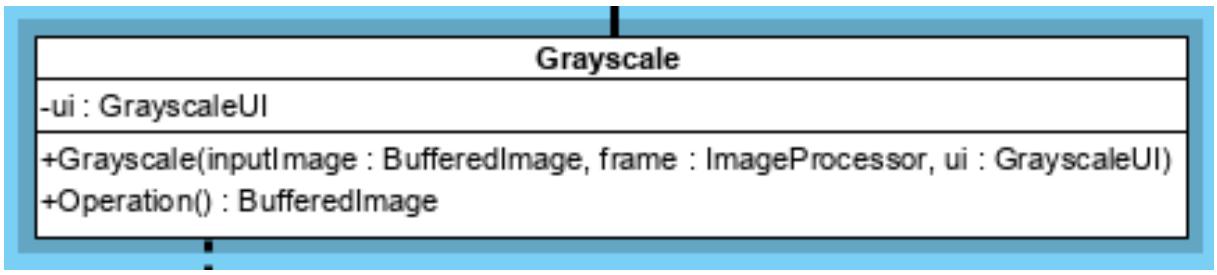

Figure 11

Figure 11 depicts the *Grayscale* class with a composition relationship to *view.GrayscaleUI*.

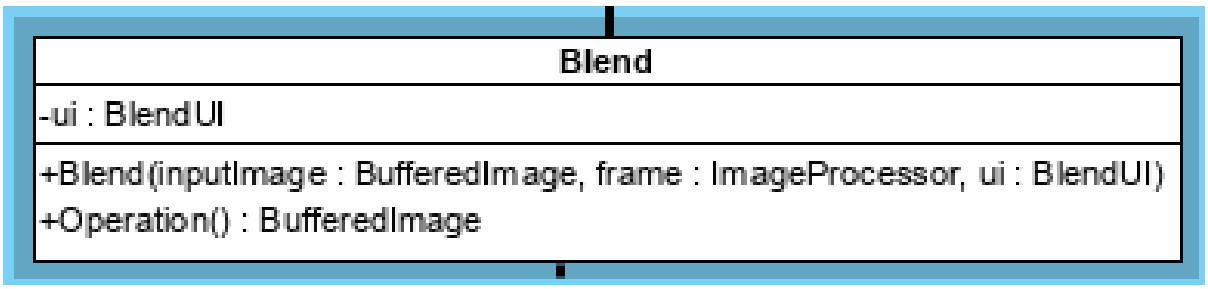

Figure 12

Figure 12 depicts the *Blend* class with a composition relationship to *view.BlendUI*.

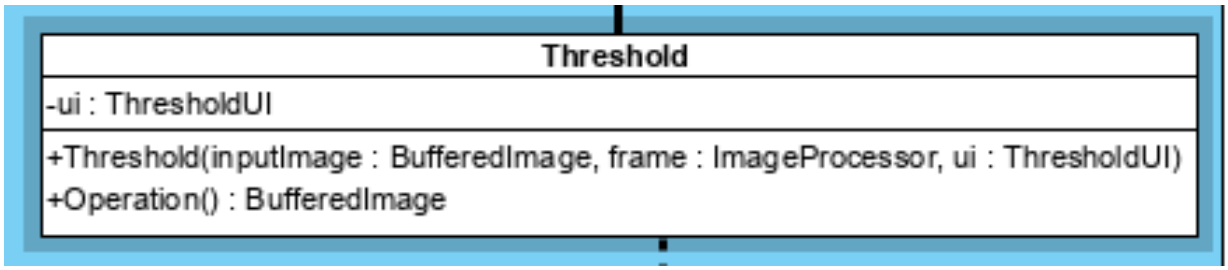

Figure 13

Figure 13 depicts the *Threshold* class with a composition relationship to *view.ThresholdUI*.

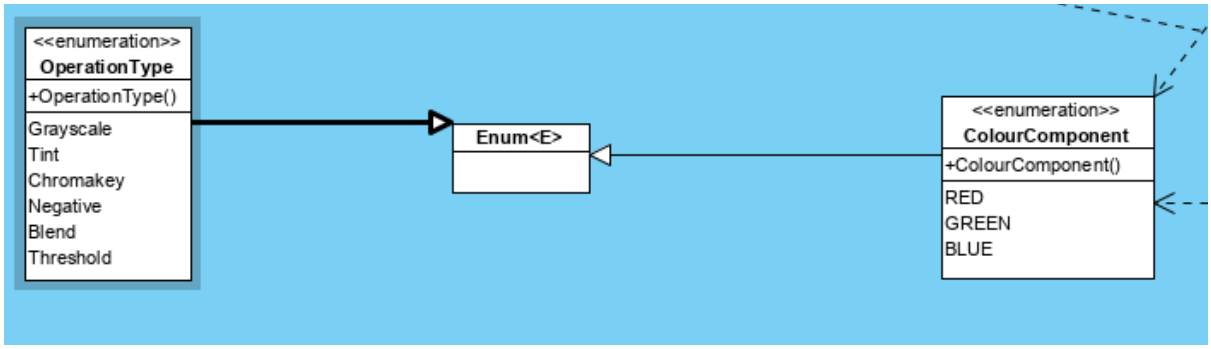

Figure 14

Figure 14 depicts the *OperationType* and *ColourComponent Enum*'s which extend the *Enum* class via a generalization relationship.

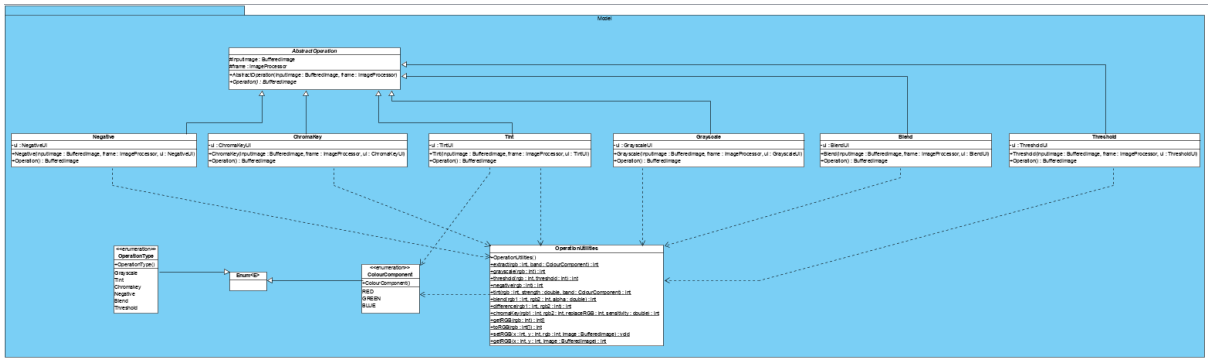

Figure 15

Figure 15 is the full class diagram for the model package.

View package

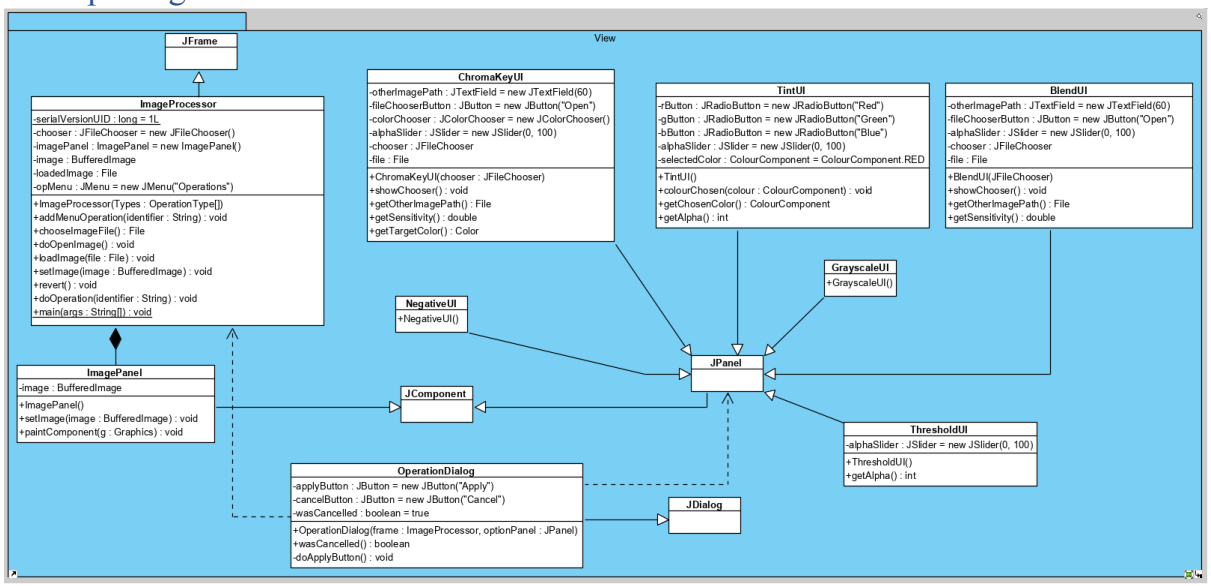

Figure 16

Figure 16 depicts the classes inside of view package. *ImageProcessor* extends *JFrame* and has a composition relationship to *ImagePanel* which extends *JComponent.* All of the "UI" classes extend *JPanel* where *JPanel* extends *JComponent*. *OperaitonDialog* extends *JDialog* and has dependency relationships to *ImageProcessor* and *JPanel.*

Implementation

All source code can be found at (Bhatti, 2020).

AbstractOperation

```
package model;

import java.awt.image.BufferedImage;

public abstract class AbstractOperation {

    protected final BufferedImage inputImage;
    protected ImageProcessor frame;

    public AbstractOperation(BufferedImage inputImage, ImageProcessor frame) {
        // TODO Auto-generated constructor stub
        this.inputImage = inputImage;
        this.frame = frame;
    }

    public abstract BufferedImage Operation();
}
```

Figure 17

The constructor in Figure 17 takes a *BufferedImage* and *ImageProcessor* instance as parameters in which they are assigned to protected variables named *inputImage* and *frame* respectively. There is also a public abstract method named *Operation* which returns a *BufferedImage* instance. The *Operation* method is abstract as each operation will execute with different logic; thus, an abstract method is most suitable as the contents of the method doesn't have to be defined in *AbstractOperation* but all subclasses must implement the *Operation* method.

ChromaKey

```java
package model;

import java.awt.image.BufferedImage;

public class ChromaKey extends AbstractOperation {

    private ChromaKeyUI ui;

    public ChromaKey(BufferedImage inputImage, ImageProcessor frame, ChromaKeyUI ui) {
        super(inputImage, frame);
        // TODO Auto-generated constructor stub
        this.ui = ui;
    }

    @Override
    public BufferedImage Operation() {
        final OperationDialog dialog = new OperationDialog(frame, this.ui);
        dialog.setVisible(true);
        if (!dialog.wasCancelled()) {
            try {
                double sensitivity = this.ui.getSensitivity();
                BufferedImage otherImage = ImageIO.read(this.ui.getOtherImagePath());

                int targetRGB = this.ui.getTargetColor().getRGB();

                BufferedImage output = new BufferedImage(inputImage.getWidth(), inputImage.getHeight(),
                        inputImage.getType());
                for (int x = 0; x < output.getWidth(); x++) {
                    for (int y = 0; y < output.getHeight(); y++) {
                        int inputRGB = OperationUtilities.getRGB(x, y, inputImage);
                        int otherRGB = OperationUtilities.getRGB(x, y, otherImage);
                        int outputRGB = OperationUtilities.chromaKey(inputRGB, otherRGB, targetRGB, sensitivity);
                        OperationUtilities.setRGB(x, y, outputRGB, output);
                    }
                }
                return output;
            } catch (IOException ex) {
                ex.printStackTrace();
                return inputImage;
            }
        } else {
            return inputImage;
        }
    }

}
```

Figure 18

As depicted in Figure 18, *ChromaKey* extends *AbstractOperation* thus implementing the *Operation* method. The constructor takes a *BufferedImage, ImageProcessor* and *ChromaKeyUI* instance as parameters where the *BufferedImage* and *ImageProcessor* instances are passed to the superclass *AbstractOperation* to assign these instances to the *ChromaKey* instance. The *ChomaKeyUI* instance is stored inside a private variable named *ui* which would be associated with the *ChromaKey* object when the class is instantiated.

Similarly, all classes which extend *AbstractOperation* implement the *Operation* method with a similar constructor as seen in Figure 18 however, the *ui* variable is typed according to the operation being executed, for instance, the *Tint* class will use a *TintUI* whereas the *Grayscale* class will use a *GrayscaleUI.* Furthermore, the *ChromaKey, Tint* and *Grayscale* classes are the only classes which take the code supplied in (Child, 2020) inside the *ImageProcessor* class under the *doChromaKey*, *doTint* and *doGrayscale* functions and moves the logic inside individual *Operation* methods respectively. For instance, the logic under the *doChromaKey* function inside *ImageProcessor* would be placed inside the *ChromaKey* class under the *Operation* method.

Grayscale

```java
package model;

import java.awt.image.BufferedImage;

public class Grayscale extends AbstractOperation {

    private GrayscaleUI ui;

    public Grayscale(BufferedImage inputImage, ImageProcessor frame, GrayscaleUI ui) {
        super(inputImage, frame);
        // TODO Auto-generated constructor stub
        this.ui = ui;
    }

    @Override
    public BufferedImage Operation() {
        final OperationDialog dialog = new OperationDialog(frame, this.ui);
        dialog.setVisible(true);
        if (!dialog.wasCancelled()) {
            for (int x = 0; x < inputImage.getWidth(); x++) {
                for (int y = 0; y < inputImage.getHeight(); y++) {
                    final int inputRGB = OperationUtilities.getRGB(x, y, inputImage);
                    final int outputRGB = OperationUtilities.grayscale(inputRGB);
                    OperationUtilities.setRGB(x, y, outputRGB, inputImage);
                }
            }
        }
        return inputImage;
    }

}
```

Figure 19

Tint

```java
package model;

import java.awt.image.BufferedImage;

public class Tint extends AbstractOperation {

    private TintUI ui;

    public Tint(BufferedImage inputImage, ImageProcessor frame, TintUI ui) {
        super(inputImage, frame);
        // TODO Auto-generated constructor stub
        this.ui = ui;
    }

    @Override
    public BufferedImage Operation() {
        final OperationDialog dialog = new OperationDialog(frame, this.ui);
        dialog.setVisible(true);
        if (!dialog.wasCancelled()) {
            final ColourComponent band = this.ui.getChosenColor();
            final double alpha = this.ui.getAlpha() / 100.0;
            for (int x = 0; x < inputImage.getWidth(); x++) {
                for (int y = 0; y < inputImage.getHeight(); y++) {
                    final int inputRGB = OperationUtilities.getRGB(x, y, inputImage);
                    final int outputRGB = OperationUtilities.tint(inputRGB, alpha, band);
                    OperationUtilities.setRGB(x, y, outputRGB, inputImage);
                }
            }
        }
        return inputImage;
    }

}
```

Figure 20

Negative

```
package model;

import java.awt.image.BufferedImage;

public class Negative extends AbstractOperation {

    private NegativeUI ui;

    public Negative(BufferedImage inputImage, ImageProcessor frame, NegativeUI ui) {
        super(inputImage, frame);
        // TODO Auto-generated constructor stub
        this.ui = ui;
    }

    @Override
    public BufferedImage Operation() {
        final OperationDialog dialog = new OperationDialog(frame, this.ui);
        dialog.setVisible(true);
        if (!dialog.wasCancelled()) {
            for (int x = 0; x < inputImage.getWidth(); x++) {
                for (int y = 0; y < inputImage.getHeight(); y++) {
                    final int inputRGB = OperationUtilities.getRGB(x, y, inputImage);
                    final int outputRGB = OperationUtilities.negative(inputRGB);
                    OperationUtilities.setRGB(x, y, outputRGB, inputImage);
                }
            }
        }
        return inputImage;
    }

}
```

Figure 21

This class' *Operation* method is similar to the *Operation* method in *Grayscale* however, *Grayscale* uses *OperationUtilities.grayscale(inputRGB)* for the *outputRGB* variable whereas, Figure 21 uses *OperationUtilities.negative(inputRGB)*.

NegativeUI

```
package view;

import javax.swing.JLabel;

public class NegativeUI extends JPanel {

    public NegativeUI() {
        // TODO Auto-generated constructor stub
        add(new JLabel("Change image to negative"));
    }

}
```

Figure 22

Figure 22 is similar to the implementation used for *GrayscaleUI* as the user is only asked to confirm the execution of the operation thus, a custom *JLabel* has been used to inform the user that they are about to execute the *Negative* operation.

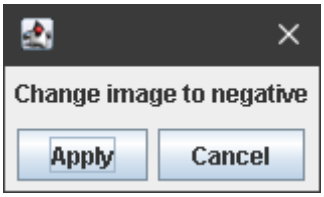

Figure 23

Figure 23 is the interface for the negative operation.

Blend

```
package model;

import java.awt.image.BufferedImage;

public class Blend extends AbstractOperation {

    private BlendUI ui;

    public Blend(BufferedImage inputImage, ImageProcessor frame, BlendUI ui) {
        super(inputImage, frame);
        // TODO Auto-generated constructor stub
        this.ui = ui;
    }

    @Override
    public BufferedImage Operation() {
        final OperationDialog dialog = new OperationDialog(frame, this.ui);
        dialog.setVisible(true);
        if (!dialog.wasCancelled()) {
            try {
                BufferedImage otherImage = ImageIO.read(this.ui.getOtherImagePath());
                double sensitivity = this.ui.getSensitivity();

                BufferedImage output = new BufferedImage(inputImage.getWidth(), inputImage.getHeight(),
                        inputImage.getType());
                for (int x = 0; x < output.getWidth(); x++) {
                    for (int y = 0; y < output.getHeight(); y++) {
                        int inputRGB = OperationUtilities.getRGB(x, y, inputImage);
                        int otherRGB = OperationUtilities.getRGB(x, y, otherImage);
                        int outputRGB = OperationUtilities.blend(inputRGB, otherRGB, sensitivity);
                        OperationUtilities.setRGB(x, y, outputRGB, output);
                    }
                }
                return output;
            } catch (IOException ex) {
                ex.printStackTrace();
                return inputImage;
            }
        } else {
            return inputImage;
        }
    }

}
```

Figure 24

Similar to the *Operation* method in *ChromaKey* where *ChromaKey* uses *OperationUtilities.chromakey(inputRGB, otherRGB, targetRGB, sensitivity)* for the *outputRGB* variable and contains a *sensitivity* variable typed as a *double*, Figure 24 uses *OperationUtilities.blend(inputRGB, otherRGB, sensitivity)* and doesn't use or initialize a *targetRGB* variable.

BlendUI

```java
package view;

import java.awt.BorderLayout;

public class BlendUI extends JPanel {

    private final JTextField otherImagePath = new JTextField(60);
    private final JButton fileChooserButton = new JButton("Open");
    private final JSlider alphaSlider = new JSlider(0, 100);

    private final JFileChooser chooser;
    private File file;

    public BlendUI(final JFileChooser chooser) {
        super(new BorderLayout());

        this.chooser = chooser;

        final JPanel pathPanel = new JPanel();
        pathPanel.add(this.otherImagePath);
        pathPanel.add(this.fileChooserButton);
        pathPanel.setBorder(BorderFactory.createTitledBorder("Image to blend"));
        alphaSlider.setBorder(BorderFactory.createTitledBorder("How close to blend the images through"));

        add(pathPanel, BorderLayout.NORTH);
        add(this.alphaSlider, BorderLayout.SOUTH);

        this.otherImagePath.setEditable(false);

        this.fileChooserButton.addActionListener(ev -> showChooser());
    }

    private void showChooser() {
        if (this.chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
            this.file = this.chooser.getSelectedFile();
            this.otherImagePath.setText(this.file.getPath());
        }
    }

    public File getOtherImagePath() {
        return this.file;
    }

    public double getSensitivity() {
        return this.alphaSlider.getValue() / 100.0;
    }
}
```

Figure 25

Although similar to *ChromaKeyUI;* Figure 25 doesn't utilise a *JColourChooser* object as it's not necessary for the blend operation. Furthermore, as the *targetRGB* isn't needed for the blend operation, the *getTargetColour* method in the *ChromaKeyUI* isn't present in Figure 25.

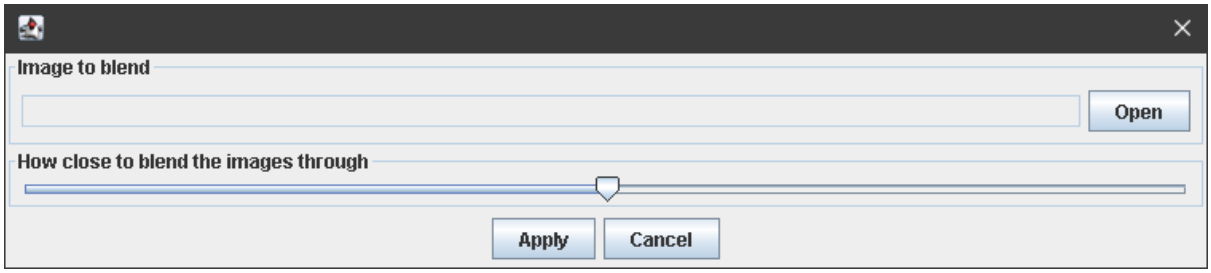

Figure 26

Figure 26 is the interface for the blend operation.

Threshold

```java
package model;

import java.awt.image.BufferedImage;

public class Threshold extends AbstractOperation {

    private ThresholdUI ui;

    public Threshold(BufferedImage inputImage, ImageProcessor frame, ThresholdUI ui) {
        super(inputImage, frame);
        // TODO Auto-generated constructor stub
        this.ui = ui;
    }

    @Override
    public BufferedImage Operation() {
        final OperationDialog dialog = new OperationDialog(frame, this.ui);
        dialog.setVisible(true);
        if (!dialog.wasCancelled()) {
            final int alpha = this.ui.getAlpha();
            for (int x = 0; x < inputImage.getWidth(); x++) {
                for (int y = 0; y < inputImage.getHeight(); y++) {
                    final int inputRGB = OperationUtilities.getRGB(x, y, inputImage);
                    final int outputRGB = OperationUtilities.threshold(inputRGB, alpha);
                    OperationUtilities.setRGB(x, y, outputRGB, inputImage);
                }
            }
        }
        return inputImage;
    }

}
```

Figure 27

Similar to the *Operation* method in *Tint* where *Tint* uses *OperationUtilities.tint(inputRGB, alpha, band)* for the *outputRGB* variable and contains a *band* variable typed as a *ColourComponent*, Figure 27 uses *OperationUtilities.threshold(inputRGB, alpha)* and doesn't use or initialize a *band* variable.

ThresholdUI

```java
package view;

import java.awt.BorderLayout;

public class ThresholdUI extends JPanel {

    private final JSlider alphaSlider = new JSlider(0, 100);

    public ThresholdUI() {
        super(new BorderLayout());

        add(new JLabel("Convert image to black and white with threshold value"), BorderLayout.CENTER);
        add(this.alphaSlider, BorderLayout.SOUTH);
    }

    public int getAlpha() {
        return this.alphaSlider.getValue();
    }
}
```

Figure 28

Although similar to *TintUI*, Figure 28 utilises a *JSlider* object so that when a user specifies an intensity from the *JSlider,* it will be used as the threshold value for the operation. Furthermore, as the

alphaSlider is used to store the threshold value for this operation, the *getAlpha* method in the *TintUI* is also present in Figure 28.

Figure 29

Figure 29 is the interface for the threshold operation.

OperationFactory

```
package controller;

import java.awt.image.BufferedImage;

public class OperationFactory {

    private final static JFileChooser chooser = new JFileChooser();

    private final static GrayscaleUI grayscaleUI = new GrayscaleUI();
    private final static TintUI tintUI = new TintUI();
    private final static ChromaKeyUI chromaKeyUI = new ChromaKeyUI(chooser);
    private final static NegativeUI negativeUI = new NegativeUI();
    private final static BlendUI blendUI = new BlendUI(chooser);
    private final static ThresholdUI thresholdUI = new ThresholdUI();

    public static AbstractOperation createOperationChanger(OperationType type, BufferedImage inputImage,
            ImageProcessor frame) {
        switch (type) {
        case Grayscale:
            return new Grayscale(inputImage, frame, grayscaleUI);
        case Tint:
            return new Tint(inputImage, frame, tintUI);
        case Chromakey:
            return new ChromaKey(inputImage, frame, chromaKeyUI);
        case Negative:
            return new Negative(inputImage, frame, negativeUI);
        case Blend:
            return new Blend(inputImage, frame, blendUI);
        case Threshold:
            return new Threshold(inputImage, frame, thresholdUI);
        }
        return null;
    }

}
```

Figure 30

Figure 30 is the *OperationFactory* class which is called when an operation is executed. *createOperationChanger* returns an *AbstractOperation* which would be an instance of one of the operation classes. The method takes *OperationType, BufferedImage* and *ImageProcessor* objects as parameters. A *switch-case* statement is used to identify the type of operation being executed where when the *type* matches a *case* statement, an operation class related to that *case* statement is instantiated. Each operation class will take a *BufferedImage, ImageProcessor* and *JPanel* object. Static variables are used to instantiate each interface for each operation.

OperationType

```
package model;

public enum OperationType {
    Grayscale, Tint, Chromakey, Negative, Blend, Threshold
}
```

Figure 31

Figure 31 is an *Enum* that contains all of the operations a user may execute.

ImageProcessor

```
public static void main(final String[] args) {
    SwingUtilities.invokeLater(() -> new ImageProcessor(OperationType.values()));
}
```

Figure 32

Figure 32 depicts the *main* method of the *ImageProcessor* class where *OperationType.values()* is passed as a parameter to the constructor of *ImageProcessor* to retrieve a list of all of the values inside the *OperationType* Enum.

```
public ImageProcessor(OperationType[] Types) {
    this.chooser.setMultiSelectionEnabled(false);
    this.chooser.setCurrentDirectory(new File(".")); // set current directory

    setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    final JMenuBar menuBar = new JMenuBar();
    final JMenu fileMenu = new JMenu("File");

    final JMenuItem openItem = new JMenuItem("Open");
    openItem.addActionListener(ev -> doOpenImage());
    fileMenu.add(openItem);

    final JMenuItem revertItem = new JMenuItem("Revert");
    revertItem.addActionListener(ev -> revert());
    fileMenu.add(revertItem);

    menuBar.add(fileMenu);
    menuBar.add(this.opMenu);
    setJMenuBar(menuBar);

    add(this.imagePanel, BorderLayout.CENTER);
    pack();

    for (OperationType type : Types) {
        addMenuOperation(type.toString());
    }

    this.setVisible(true);
}
```

Figure 33

Figure 33 is the constructor of *ImageProcessor* taking an array of *OperationType* values which are then iterated over inside a for loop to add each *OperationType* as a string value via the *addMenuOperation* method.

```
private void doOperation(final String identifier) {
    OperationType type = OperationType.valueOf(identifier);
    AbstractOperation option = OperationFactory.createOperationChanger(type, this.image, this);
    setImage(option.Operation());
}
```

Figure 34

Figure 34 depicts the *doOperation* method inside of the *ImageProcessor* class. The *identifier* variable is converted from a *String* type to an *OperationType* which is parsed to the *OperationFactory.createOperationChanger* to decipher which operation to execute along with a *BufferedImage* and *ImageProcessor* object which were instantiated during the execution of *ImageProcessor*. After the *OperationFactory.createOperationChanger* method has been executed, it's stored in an *AbstractOperation* variable named *option* which is passed to the *setImage* method to execute the *Operation* method on the operation stored in the *option* variable.

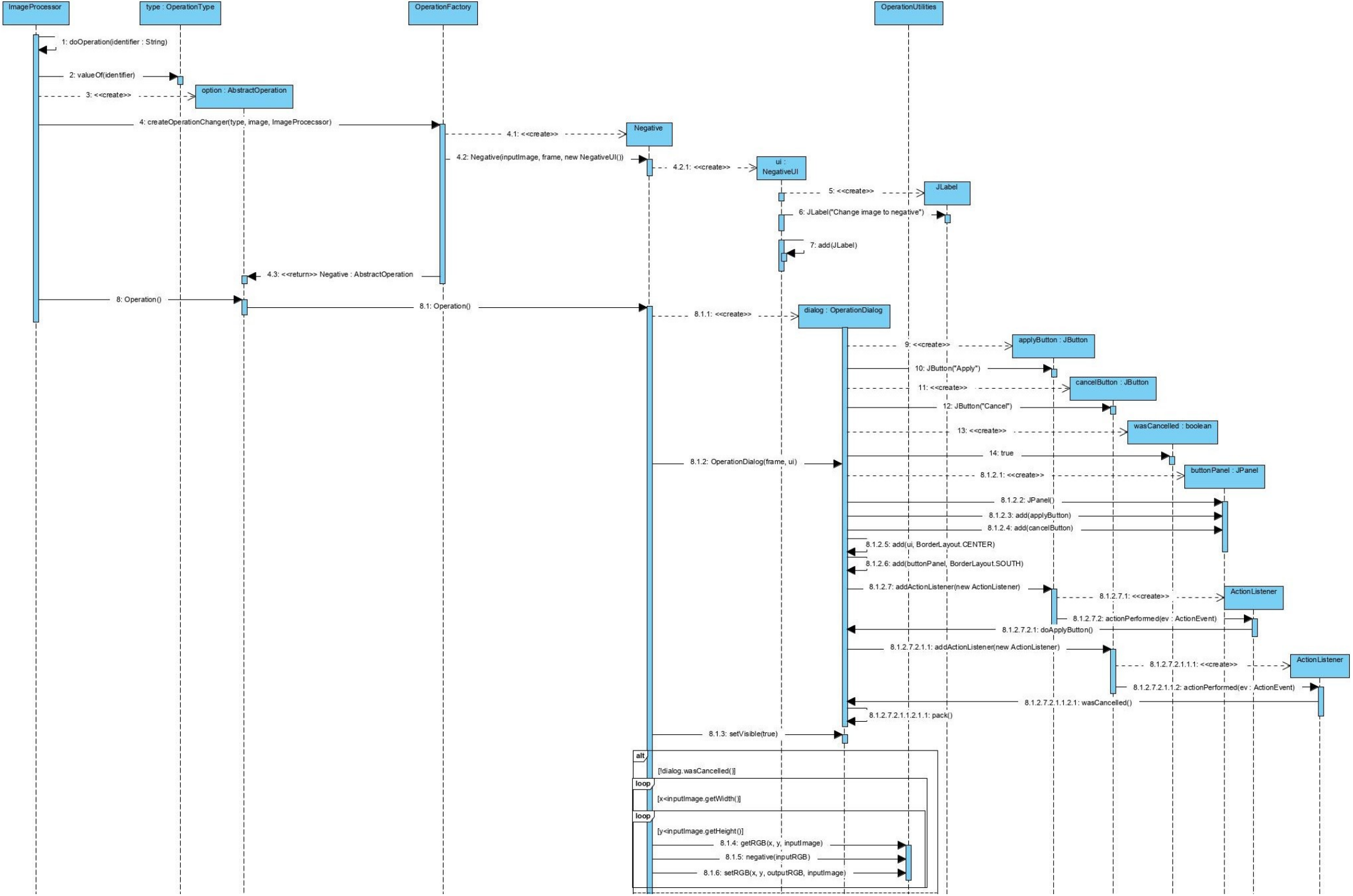
ImageProcessor
type : OperationType
OperationFactory
OperationUtilities
1: doOperation(identifier : String)
2: valueOf(identifier)
3: <<create>>
option : AbstractOperation
4: createOperationChanger(type, image, ImageProcecssor)
4.1: <<create>>
Negative
4.2: Negative(inputImage, frame, new NegativeUI())
4.2.1: <<create>>
ui : NegativeUI
5: <<create>>
JLabel
6: JLabel("Change image to negative")
7: add(JLabel)
4.3: <<return>> Negative : AbstractOperation
8: Operation()
8.1: Operation()
8.1.1: <<create>>
dialog : OperationDialog
9: <<create>>
applyButton : JButton
10: JButton("Apply")
11: <<create>>
cancelButton : JButton
12: JButton("Cancel")
13: <<create>>
wasCancelled : boolean
14: true
8.1.2: OperationDialog(frame, ui)
8.1.2.1: <<create>>
buttonPanel : JPanel
8.1.2.2: JPanel()
8.1.2.3: add(applyButton)
8.1.2.4: add(cancelButton)
8.1.2.5: add(ui, BorderLayout.CENTER)
8.1.2.6: add(buttonPanel, BorderLayout.SOUTH)
8.1.2.7: addActionListener(new ActionListener)
8.1.2.7.1: <<create>>
ActionListener
8.1.2.7.2: actionPerformed(ev : ActionEvent)
8.1.2.7.2.1: doApplyButton()
8.1.2.7.2.1.1: addActionListener(new ActionListener)
8.1.2.7.2.1.1.1: <<create>>
ActionListener
8.1.2.7.2.1.1.2: actionPerformed(ev : ActionEvent)
8.1.2.7.2.1.1.2.1: wasCancelled()
8.1.2.7.2.1.1.2.1.1: pack()
8.1.3: setVisible(true)
alt
[!dialog.wasCancelled()]
loop
[x<inputImage.getWidth()]
loop
[y<inputImage.getHeight()]
8.1.4: getRGB(x, y, inputImage)
8.1.5: negative(inputRGB)
8.1.6: setRGB(x, y, outputRGB, inputImage)

Figure 35 – sequence diagram

Figure 35 depicts a sequence diagram of the negative operation being executed. *doOperation* is called on *ImageProcessor* where the menu item selected is parsed to the *OperationType* Enum to retrieve the operation to execute and store it in a variable named *type*. Then an *option* value is created to store the operation class to execute where *createOperationChanger* is executed on *OperationFactory* to retrieve an *AbstractOperation* instance. If the menu item selected was negative, this would return a *Negative* class instance which would create a *NegativeUI* object containing a *JLabel.* When the *Operation* method is executed on the *option* variable, this would execute *Negative*'s *Operation* method as *option* stores a reference to the operation to execute.

Executing the *Operation* method on *Negative* will create an *OperationDialog* object which would store apply and cancel buttons and would place the *NegativeUI* instance above them. After instantiating the *OperationDialog* instance, it is assigned to a variable named *dialog* and is then made visible on the screen. If the user cancels the negative operation, nothing happens to the image. If the user clicks apply, calls to *OperationUtilities* are made to perform the negative operation on each pixel in the image inside the for loop. After the negative operation has been executed, the image is returned as a *BufferedImage* to *ImageProcessor.* In *ImageProcessor* the image displayed is updated by calling the *setImage* method.

Appraisal

Although Figure 1 – BlendUI, Figure 2 – NegativeUI and Figure 3 - ThresholdUI have been implemented the execution of these operations may not be as intended do to the vagueness of the specification. For instance, the threshold operation utilises a *JSlider* as seen in Figure 28 however, an input field could have been used to retrieve the threshold value.

Each operation class is separated from their user interface class but combining the code from the operation class and user interface class could reduce the complexity of the Class diagram, however, this would make the code base more difficult to decipher for a new user. Furthermore, the original codebase was structured by having all of the classes in the same package, however, I have separated each of the classes using an MVC architectural structure, as depicted in (Child, 2006), to simplify the file structure of the application.

All of the individual operation classes could implement an interface thus having an *Operation* method however, similar to content described in (Child, 2020), I used the AbstractOperation class to reduce the number of lines in each operation class as if an interface were implemented, additional lines would need to be added to each operation class to initialise and assign the *BufferedImage* and *ImageProcessor* objects.

When a new operation is added, the user interface must be placed inside the view package, the logic for the operation must be placed in the model package where this class must extend *AbstractOperation* and a new case statement must be added to the *OperationFactory* class inside the controller package to link the user interface and the operation logic. Also, to reduce memory usage from a new user interface class being created each time the operation class is called, the user interface should be assigned and stored inside a static variable as depicted in Figure 30. This may make the application difficult to understand for a new user but this allows for further modularity as now users may change how each package works alongside implementing new operations.

Bibliography

Balsamiq Studios, LLC, 2008. *Balsamiq Wireframes,* Sacramento: Balsamiq Studios, LLC.

Bhatti, K., 2020. *ImageProcessor.* [Online]
Available at: https://github.com/k5924/ImageProcessor
[Accessed 1 December 2020].

Child, M., 2006. *06 - Custom GUI components and MVC.* [Online]
Available at: https://vle.lsbu.ac.uk/mod/resource/view.php?id=1454664
[Accessed 28 October 2020].

Child, M., 2020. *04 - Abstract classes, interfaces and UML.* [Online]
Available at: https://vle.lsbu.ac.uk/mod/resource/view.php?id=1454658
[Accessed 14 October 2020].

Child, M., 2020. *10 Factories and enumerations.* [Online]
Available at: https://vle.lsbu.ac.uk/mod/resource/view.php?id=1454672
[Accessed 3 December 2020].

Child, M., 2020. *assignment-code.* [Online]
Available at: https://stulsbuac.sharepoint.com/sites/CSI-5-OOPObject-OrientedProgramming2021copy/Class%20Materials/assignment-code.zip
[Accessed 15 December 2020].