Programming Project - SnapShop

For this program, you are provided several .java source files. The first step is to create a Project in Eclipse and import those source files into the project. When you get the source code compiled and the application running, it should look and work like the provided "runnable .jar file". You run the runnable .jar file by double-clicking on the java icon (the icon representing a hot cup of coffee).

What to submit: For maximum credit, you will have to modify some of the files you started with, and create some new .java files.

- Submit any **new** files you've created, and **any .java files you have modified**.
- **A runnable .jar file of your application**. In Eclipse, right click on your Project name in Package Explorer and select Export -> Java -> Runnable JAR file.
- Remember to include your short report (described below) in the header comments of the file SnapShopConfiguration.java.

You may have a number of files to send, you can either zip them into a single file, or attach them individually to a message to me.

Extra credit and grading: There are 3 ways to go with this project.

- Submit FlipVertical.java, Negative.java, SnapShopConfiguration.java (with report in header comment block), and the runnable jar file containing your application. (These are the Simple Transformations described below) If your application executes properly, has appropriate comments in each of the .java files, and has a short report in the header comment block of SnapShopConfiguration.java, you are guaranteed a **70%** grade on this assignment. I suggest that everyone start here! If you can't manage to debug your more advanced code before the due date, at least make sure you get these points.

- To get **100%** on this programming project, complete all of the above, and the 3X3 filters described below.

- To get extra credit, up to **120%**, implement one or more of the following suggestions, or do some research and be creative!

In this assignment, you will practice doing the following:

- designing classes (using an interface),
- using 2D arrays

Digital image processing has completely revolutionized the way images are created and used in news photography, publishing, commercial art, marketing, and even in some of the fine arts. Adobe's Photoshop program has become so ubiquitous that it has even become a verb - "this picture is a mess, I need to Photoshop it". In this assignment, you'll implement some of the core image transformation algorithms used by image processing programs like Photoshop.

Image Representation

A digital image is a rectangular array of pixel objects. Each pixel contains three integer values that range from 0 to 255, one integer each for the red, green, and blue components of the pixel, in that order. The larger a number, the brighter that color appears in the pixel. So a pixel with values (0,0,0) is black, (255,255,255) is white, (255,0,0) is pure red, (0,0,255) is pure blue, and so forth.

We can represent a Pixel as a simple Java object containing three int instance variables and, for convenience, a constructor to create a new pixel given its rgb component values. We'll treat these as simple data objects and allow direct access to their fields.

```
/** Representation of one pixel */
public class Pixel {
    public int red; // rgb values in the range 0 to 255
    public int green;
    public int blue;

    /** Construct a new pixel with specified rgb values */
    public Pixel(int r, int g, int b) {
        this.red = r;
        this.green = g;
        this.blue = b;
    }
}
```

An image is represented by an instance of class PixelImage. This class contains the methods `getData()` and `setData()` to retrieve 2-dimensional arrays of Pixel objects representing the pixels of the image. You can also use the methods `getHeight()` and `getWidth()` to get the height and width of the image.

The Application

Francois has implemented a PhotoShop-like Java application, called SnapShop. The application knows how to load image files, and provides all the user interface objects you'll need to apply your filters to the image. You will need not implement anything in the SnapShop class, but create a few new classes. The file loader of the SnapShop class expects filenames to be fully specified, that is, you must say something like `c:\directory\image.jpg`. Normal (forward) slashes also work: `c:/directory/image.jpg`. To save you some work, we've provided a shortcut so you don't need to always retype the file name; details below.

SnapShop has a main method. Once you have the files imported into the project and compiled, selecting 'run' in Eclipse will run the SnapShop application.

Also provided is an **interface** class 'Filter'. An interface simply specifies the methods another class must implement, and cannot be used to make objects itself. You will be writing classes that implement this Filter interface, one for each transformation you write. Each class implementing the Filter interface must have a method called `filter()`, which takes a PixelImage as an argument. The method then applies a transformation to the data in the image. As an example, we have included the FlipHorizontalFilter class, which flips an image horizontally.

You will need some way to tell our SnapShop class which filters you have implemented. So we've provided a class called SnapShopConfiguration, with a single method, `configure()`. In this method, you can call methods for the SnapShop object. The two methods you'll be interested in are `addFilter()`, which creates a button in the application to apply your filter, and `setDefaultFilename()`, which lets you specify a default path or filename for the file loader, to aid you in testing. For each filter that you create, there should be a call to `addFilter()` in the `configure()` method.

(A note on `setDefaultFilename()`. Windows path names have backslashes (\) in them. To specify this in a string in a Java source program, you need to put two backslashes. For example, the Java string for `c:\directory\image.jpg` would be `"c:\\directory\\image.jpg"`.)

Here are the files you need to download to get started:

SnapShop.java, SnapShopConfiguration.java, Pixel.java, PixelImage.java, Filter.java, FlipHorizontalFilter.java.

You should use billg.jpg to test your code, as that is the input used to create the reference images.

If you use other images as test images, be sure that they aren't too much bigger than billg.jpg. Your program will take a long time to process a large image, and the window may not wind up looking right.

Simple Transformations

There are two kinds of transformations that you are required to implement. The simple transformations can be implemented by replacing each Pixel in the existing image with the updated one. The more complex 3x3 transformations require creating a new array of Pixels with the transformed image, then updating the image instance variable to refer to the new array once it is completely initialized.

The first three transformations you should implement flip the image horizontally and vertically, and transform the image into a photographic negative of itself (that is, you should create a flipHorizontalFilter, flipVerticalFilter, and NegativeFilter class). We have implemented flipHorizontalFilter for you.

The first two require a simple rearrangement of the pixels that reverses the order of rows or columns in the image. The negate transformation is done by replacing each Pixel in the image with a new Pixel whose rgb values are calculated by subtracting the original rgb values from 255. These subtractions are done individually for each of the red, green, and blue colors.

These transformations can be performed by modifying the image array of Pixels directly. You should do these first to get a better idea of how the image is represented and what happens when you modify the Pixels. You should make every effort to get this far before the end of week 9. That will ensure that you've made good progress on this assignment, or at least know what you need to clear up in discussions during lecture.

Notes:

- You can assume that the image array is rectangular, i.e., all rows have the same length.
- All of these simple transformations are their own inverse. If you repeat any of these transformations twice in a row, you should get the original image back.
- You should use relatively small image files for testing. The program will work fine with large images, but there can be a significant delay while the transformed image is created if the image is large.

3x3 Transformations

Once you've got the simple transformations working, you should implement this next set, which includes Gaussian blur, Laplacian, Unsharp Masking, and Edgy. All of these transformations are based on the following idea: each pixel in the transformed image is calculated from the values of the original pixel and its immediate neighbors, i.e., the 3x3 array of pixels centered on the old pixel whose new value we are trying to calculate. The new rgb values can be obtained by calculating a weighted average; the median, minimum, or maximum; or something else. As with the negate transformation, the calculations are carried out independently for each color, i.e., the new red value for a pixel is obtained from the old red values, and similarly for green and blue.

The four transformations you should implement all compute the new pixel values as a weighted average of the old ones. The only difference between them is the actual weights that are used. You should be able to add a single method inside class PixelImage to compute a new image using weighted averages, and call it from the methods for the specific transformations with appropriate weights as parameters. **You should *not* need to repeat the code for calculating weighted averages four times, once in each transformation.** The method you add to PixelImage to do the actual calculations can, of course, call additional new methods if it makes sense to break the calculation into smaller pieces.

Here are the weights for the 3x3 transformations you should implement.

Gaussian

```
1  2  1
2  4  2
1  2  1
```

After computing the weighted sum, the result must be divided by 16 to scale the numbers back down to the range 0 to 255. The effect is to blur the image.

Laplacian

```
-1  -1  -1
-1   8  -1
-1  -1  -1
```

The neighboring pixel values are subtracted from 8 times the center one, so no scaling is needed. However, you do need to check that the weighted average is between 0 and 255. If it is less than 0, replace the calculated value with 0 (i.e., the new value is the maximum of 0 and the calculated value). If it is greater than 255, then replace the calculated value with 255. This transformation detects and highlights edges.

Unsharp masking

```
-1  -2  -1
-2  28  -2
-1  -2  -1
```

This transformation is created by multiplying the center pixel and subtracting the Gaussian weighted average. The result must be divided by 16 to scale it back down to the range 0 to 255. As with the Laplacian transformation, check for negative weighted averages or weighted averages greater than 255 (and do the same thing as in the Laplacian case to fix it).

Edgy

```
-1  -1  -1
-1   9  -1
-1  -1  -1
```

This adds the Laplacian weighted average to the original pixel, which sharpens the edges in the image. It does not need scaling, but you need to watch for weighted averages less than 0 or greater than 255.

Notes:

- The complication with these transformations is that the new value of each pixel depends on the neighboring ones, as well as itself. That means we cannot replace the original pixels with new values before the old values have been used to compute the new values of their neighbors. The simplest way to handle this is to create a new 2D Pixel array the

same size as the old image, compute Pixels for the new image and store them in the new array, then change the image instance variable to refer to the new array once it is completed.

- You should assume the image has at least three rows and columns and you do not need to worry about updating the first and last rows and columns. In other words, **only update the interior pixels** that have neighbors on all four sides. However, every position in the array of Pixels must have a reference to a Pixel object; you can't just leave a position in the array uninitialized.
- Debugging hint: From past experience, we've noticed that bugs in the implementation of these transformations tend to produce more spectacular visible effects with the Laplacian weights. You should start with this set of weights when testing your code for the 3x3 transformations.

Further explorations (for some amount of extra credit)

Once you have the basic assignment working, turn it in so you've got something submitted. Then feel free to experiment with additional transformations. Besides varying the weights, you can try replacing each pixel by the minimum or maximum value in the neighborhood, or the median. Try weights that are not symmetric.

Modify the same method you wrote for the 3x3 filter, so that it works for a 5x5, 7x7, 9x9 filter, etc.

Or look on the web for additional things you can do with the images. Some amount of extra credit (up to 20 points) may be awarded for particularly imaginative work.

Report

Write a short report (as a comment) at the beginning of the SnapShopConfiguration file.

Describe:

- what additional filters (besides the three simple transformations and the four 3x3 transformations) you wrote and submitted, if any
- what works and what doesn't
- any surprises or problems you encountered while implementing the transformations.

(-5 points if you don't provide a report)

Your program has to be your own.