1 Image Manipulation and Combating Piracy

The software piracy of expensive and powerful image-editing software (such as Adobe Photoshop) is a widespread problem, harming both the conglomerates that rely on genuine sales of their product and the end users, whose hard drives quickly and inevitably become cluttered with endless stream of configuration files for features they don’t understand and half-finished, poorly-edited images. I believe the introduction of a lightweight, free image manipulation program, with enough features to satisfy and serve the standard picture-taking human – but not so many as to bog one down – is crucial in the fight against software piracy.

In creating this program, I hoped to refine my understanding of object-oriented code and become one-again comfortable with the Java programming language (aside from aiming to end piracy, of course). My end-goal for the program was to create a simple image editing tool with basic, yet widely used, manipulations and the ability to add more filters with ease and speed, all while maintaining a codebase clear enough for a new programmer to understand. In addition, I wanted to practice good programming practices, such as consistent formatting style and informative commenting, and attempt to design useful and powerful abstractions.

2 Types of Manipulations and Solution Design

2.1 Seeing the Future

Since I wanted to keep up with the development progress of commercial image editing software, I thought being able to quickly and easily support new filters and manipulations as the latest techniques become more refined would be helpful. Sifting through research papers and implementing intricate algorithms seemed difficult enough, so I decided to help future me out and find ways to make my codebase more extensible.

After thinking about a few widespread transformations, the first common theme that stood out to me was about how the manipulations actually went about changing the image: pixel-by-pixel, modifying certain pixels based on specific guidelines. Regardless of what those guidelines were, they all had to somehow access each pixel and perform an action.

2.1.1 Prediction #1: Pixel-By-Pixel, Top-To-Bottom, Left-To-Right

I realized that for every manipulation I could think of, the order in which I accessed the pixels didn’t matter — they all just needed to iterate through the pixels in the image data. So, I decided to simply go through the image data from top-to-bottom and left-to-right.
At first glance, it also seemed that, while on a single pixel, the modification would be working only on that one pixel. However, I thought of some counterexamples that helped me form my next prediction.

2.1.2 Prediction #2: Modifying Other Pixels

As I started thinking about how to add reflection effects (like VerticalReflect), I noticed that while it was certainly possible to create a new, blank image (by initializing a two-dimensional array of the same size) and fill pixels from one side of the “line down the middle” accordingly, a more efficient solution would be to simply swap a pixel on one half with its respective partner on the other side.

In doing so, on each step, my pixel-by-pixel iterator would need to edit a pixel it currently wasn’t on. It would also need to access the image data and modify the image in place, which brings me to my next two predictions.

2.1.3 Prediction #3: Additional Data

In order to swap pixels on opposite sides of the dividing line, both VerticalReflect and HorizontalReflect needed access to the “partner” of the current pixel. Additionally, I thought that some manipulations may need to access more or less restricted amounts of data. For example, neighborhood filters require the pixels that surround our current pixel.

2.1.4 Prediction #4: Modifying Image Data In-Place

Many filters, particularly those that don’t rely on pixels other than the current pixel, don’t require creating and then populating a blank image of the same size. For example, NoRed, GreenOnly, and Threshold can work in-place.

However, if you try working in-place with a neighborhood filter, like Dilate, each pixel modification decision will be made using partially modified image data (for example, given a particular pixel, suppose 4 of the 8 surrounding pixels have already been dilated). In these cases, it’s necessary to keep the original image data intact for use as a reference.

2.2 Preparing Parent Classes

Every effect is a subclass that extends the abstract ImageEffect class. In the starter code, this class specified an abstract method apply, which took the original image data and some parameters, and returned manipulated image data. Being an abstract class, it was up to any class extending ImageEffect to programmatically define its effect.

My first prediction (see 2.1.1, Prediction #1) inspired me to first modify this apply method to always iterate the image data (top-to-bottom, left-to-right), calling a new modify method meant to modify the current pixel. I intended to keep the original apply method backwards-compatible while attempting to reuse the pixel-by-pixel iteration code.

While trying to satisfy Prediction #2 and Prediction #3, I first tried simply passing the appropriate data to each effect extending ImageEffect; however, I realized that was very inefficient — particularly gathering neighboring pixels when they are useless (such as hue filters) to support neighborhood filters. Instead, I created two protected data fields for the ImageEffect class: usesPixels and usesNeighborhood, to pass the entire image data or the neighboring pixels to the individual pixel modify method, respectively. Similarly, I added a protected data field inPlace to address Prediction #4.
Lastly, I created custom constructors to set these fields and modified the original apply method to act accordingly. For example, if usesNeighborhood is set to true, calling apply will gather the directly adjacent neighbors of the current pixel and pass this data along to the individual pixel modify method. Likewise, usesPixels will pass along the image data and inPlace sets the to-be-modified original image data to the source image data. By default, inPlace is set to true, usesPixels to false, and usesNeighborhood to false. When either usesPixels or usesNeighborhood are set to false, an empty variable of the correct type is passed to modify.

2.3 Implementing Effects (Subclasses)

2.3.1 NoRed, NoGreen, NoBlue, RedOnly, GreenOnly, BlueOnly

All of these effects can modify the image data in-place and don’t require any additional data, so the classes that implement them simply extend ImageEffect under the default data fields. To remove a shade (such as in NoRed), I returned a new pixel with that particular shade’s RGB value set to 0, making no modifications to the other two shades. Similarly, to keep only one shade, I returned a new pixel with the other two shades’ RGB value set to 0.

2.3.2 BlackAndWhite

The RGB value (0, 0, 0) represents black and (255, 255, 255) represents white. In order to convert any color pixel to either a black or white pixel, I summed up the RGB values and saw if it was closer to $0 + 0 + 0 = 0$ or $255 + 255 + 255 = 765$ by comparing the total sum to the midpoint ($\lfloor \frac{765}{2} \rfloor = 383$). I also experimented with deciding by checking how many more RGB values were closer to 0 or 255; however, this produced black-and-white images that were missing some minor details (such as the fútbol on my shirt).

2.3.3 VerticalReflect, HorizontalReflect

I decided to implement these reflection effects by iterating over half the image (the left half for VerticalReflect and the top half for HorizontalReflect) and swapping each pixel with a specific pixel on the opposite half. Both reflection effects require additional data — particularly, each pixel accessed on one side of the image needs its respective pixel on the other side. I set the usesPixels field to true so these methods could access the image data and get the pixel necessary for swapping. Swapping was carried out with the aid of a temporary variable.

2.3.4 Grow, Shrink

Rather than devise another data field to generalize the actions of Grow and Shrink, I opted to override the apply method, create a new image with the proper new dimensions, and populate it according to some mapping. For Grow, I mapped each pixel in the original image to four in the new image without any modifications, whereas in Shrink, I mapped four pixels in the original image to one in the new image, by taking the average of the pixels’ RGB values. For both manipulations, I created a new two-dimensional image data array — for Grow, both dimensions were doubled, and vice versa for Shrink except the halving was via integer division ($\frac{3}{2} = 1$, for example).
2.3.5 Threshold

By deciding that one RGB component (suppose, red) of a pixel is closer to either the full shade (red) or nothing (i.e: red RGB value is 0 or NoRed), we can reduce the possible number of colors from $256^3 \approx 16$ million to $2^3 = 8$. I set a threshold at the midpoint ($\lfloor 255/2 \rfloor = 127$) to determine whether or not to keep or remove each particular color. Using the `params` parameter, I added a dialog to prompt the user for a custom threshold ranging from 0 to 255 inclusive, with a default value of 127.

2.3.6 (Good Karma) Smooth, DeNoise, Erode, Dilate

Each of these neighborhood filters required access to the neighborhood pixels of the current pixel. Moreover, they all could not edit the image in place, as explained above. Smooth took the average of the neighborhood pixels to determine the new pixel while Erode and Dilate took the minimum and maximum neighboring pixels, respectively. For each of these filters, I ordered pixels based on the sum of their RGB components. DeNoise looked at the median neighboring pixel, for which I stored the sums in an `int` array and mapped the sum to its respective pixel using a `HashMap`. I then sorted the array, found the median pixel’s sum, and used the `HashMap` to recover the pixel we were looking for.

2.4 Storing and Retrieving Data

There are two main data containers used in this program: an integer (`int`) for holding each pixel and a two-dimensional integer array (`int[][]`) for holding the image data (which is comprised of pixels). Under the hood, each integer representing a pixel stores RGB values (accessible using the `getRed`, etc. methods). Likewise, I used the `makePixel` method to create pixels. Given some image data `pixels[y][x]`, the first index (y) represents the row (starting from the top) and the second index (x) represents the column (starting from the left).

3 Implementation, Assumptions, and Edge Cases

3.1 Style, Abstractions, and Capabilities

As I worked on the codebase, I made sure to follow a consistent style for indentation, brace positioning, variable naming, and comment structure (as per Javadoc specification). Additionally, I spent time thinking about the different requirements for various effects, hoping to generalize as much as possible and reuse code when appropriate. This ultimately lead to the overhauled (yet backwards compatible) `apply` method that can be used for a variety of purposes, depending on which flags (`inPlace`, `usesPixels`, `usesNeighborhood`) are set to true.

Since I made no assumptions about the dimensions of the image (except that they must be at least 1 pixel by 1 pixel), and since most effects work in place, thereby reducing the amount of memory required, my program has a large scope of solution; that is, it can work on a large variety of various images.
3.2 Driving through Roadblocks

While creating this program, I ran into several `ArrayIndexOutOfBoundsException` while attempting to keep track of image data indices.

3.2.1 Special Case #1: Reflections

As previously mentioned, I implemented the two reflection manipulations by swapping pixels on opposite sides of the dividing line. On my first attempt at implementing `VerticalReflect`, I assumed that for a pixel at `pixels[y][x]`, the corresponding opposite pixel would be at `pixels[y][width - x]`. Implementing this quickly led to an out of bounds exception. After plugging in some values, I found a few (such as `x = 0`) that just slightly fell out of the range image data indices. Some more thought revealed that I should actually be subtracting from `width - 1` rather than `width` to account for the off-by-one numbering of array indices.

3.2.2 Special Case #2: Odd vs. Even Image Dimensions

Throughout the project, I attempted to avoid any assumptions regarding image dimensions so that the users of my program could manipulate an image of any width or height they pleased; however, this led to some confusion while writing the reflection effects. After some sketching, I noticed that the midpoint on the relevant axis was given by a specific formula, regardless of its parity: \( \lfloor l/2 \rfloor \) where \( l \) is the length of the other axis.

3.2.3 Edge Case #1: Median of an Even Number of Elements

Additionally, in finding the median for the neighborhood filter `DeNoise`, there is an edge case if there is an even number of neighboring pixels. In this case, the median is defined as the average of the two elements at the middle. I caught this edge case by checking if the number of neighbors modulo 2 equaled 0.

3.2.4 Assumption #1: Input Is at Least 1-By-1

I assumed that the image input would be at least 1 pixel wide and 1 high. Provided this assumption holds, the number of neighboring pixels will never be zero, since the current pixel is included in the neighboring pixels.

3.2.5 Edge Case #2: Not All Directly Adjacent Positions Contain a Pixel

If the `usesNeighbors` data field is set to true, the `modify` method will be passed neighboring pixel data. However, in some situations, the positions directly adjacent to the current pixels may not all contain pixels (for example, at edge positions). To avoid this edge case, I went through each of the directly adjacent position coordinates and confirmed that they were in the bounds of the image data indices before attempting to access them and add them to the neighboring pixels `ArrayList` to be passed to `modify`.

3.2.6 Edge Case #3: Shrunk Too Small

If either dimension of some image data is 1, applying the `Shrink` effect will bring that dimension down to 0 since the manipulation uses integer division (that is, \( 1/2 = 0 \)). Normally, this will cause an `ArrayIndexOutOfBoundsException`. I decided to, instead, display a 1-by-1 white pixel.
3.3 Enhancements

3.3.1 Good Karma: Neighborhood Filters

After I added the usesNeighborhood data field, adding neighborhood filters (such as Smooth, DeNoise, Erode, and Dilate) was as simple as interacting with the neighboring pixels as necessary for the filter. For example, in Smooth, I simply took the average of the neighboring pixels in the modify method.

3.3.2 Some Ideas

Due to the abstractions I implemented, it is easy to concisely implement and modify interesting filters and effects. For example, rather than looking at the 8 pixels neighboring the current pixel for a neighborhood filter (such as Smooth), you can look at the 4 pixels in the standard cardinal directions, which creates a less blurry output image, but still adequately smooth.

Given the image data, there are some interesting tools that can be implemented. For example, looking at color frequency or quickly determining if a particular color is not already in the image using a Kalman filter. One could create a method analyze that provides access to the relevant data and displays some result.

4 Correctness

As I worked on implementing each image effect, I continuously tested each filter with an image by running the program, opening an image, and manually selecting the filter to test. In retrospect, it would have saved me a good chunk of effort had I automated this process—but, I never imaged I’d be testing and playing around with each effect so many times!

Some images highlight the effectiveness and correctness of various filters better than others. In particular, I used

- the two-color images (8.jpg, 9.jpg, 10.jpg) to test NoRed, NoGreen, and NoBlue,
- the randomly colored pixel image (6.jpg) to test RedOnly, GreenOnly, and BlueOnly,
- the “paint swatch” images (3.jpg, 4.jpg, 5.jpg) to test Threshold,
- 7.jpg, being 1 pixel by 1 pixel, for debugging the edge case in Shrink,
- a picture of Dr. Lin and a picture of me wearing a t-shirt with text and a medium-sized soccer ball for BlackAndWhite, Smooth, DeNoise, Erode, and Dilate to make the transformation of both small details, large details, and overall blurring/distortions more apparent,
- a picture of me for VerticalReflect, HorizontalReflect, and Grow.

4.1 Testing Examples

By applying Threshold with threshold at 100, then BlackAndWhite, and lastly Erode 9 times to Dr. Lin’s face (11.jpg), I was able to uncover his (potential) alter-ego: Afro Lin (see afro.png).

Also, when I tried Grow-ing, Dilate-ing 3 times, Erode-ing 4 times, and then Threshold-ing with threshold at 127 the image of bacon (1.jpg), I ended up with a fire (see fire.png).

Lastly, starting with 21.jpg and applying Smooth once, DeNoise 3 times, Dilate 9 times, Erode 18 times, and then Grow once, I ended up with colorfulroom.png.