FACULTY OF ENGINEERING AND ARCHITECTURE

DEPARTMENT INTEC
RESEARCH GROUP IBCN

Geavanceerde softwareontwikkeling

Lab Session III

Command, Strategy

GHENT UNIVERSITY

Introduction

In this Lab session we will implement a simple version of an image editing application using the Command and Strategy design patterns. We will implement a few different filters that will be applied to an image.

We will work with an uncompressed image format (bmp) that stores an RGB (red, green, blue) value from 0 to 255 per pixel for each color channel. Compressed image formats, on the other hand, share certain channel values for multiple pixels at the same time, to trick the human eye and in the meantime compress the needed memory to store the image.

Compressed images would make our session too complicated, which is why we will stick to uncompressed images. To help you with this lab, we implemented a Toolbox and some utility classes like Pixel. These will allow us to enter a string path to a file, and retrieve the 2-dimensional Pixel array of the uncompressed image to work with. Figure 1 shows how this works, the red rectangle is a part of the image that we zoom in on and the result is shown in Figure 2.

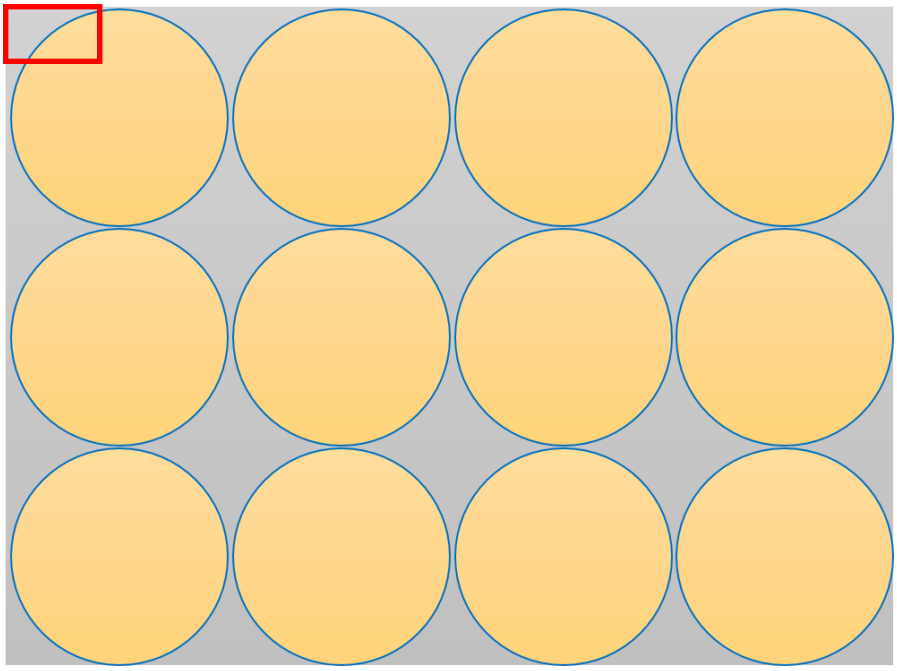

Figure 1: Sample image

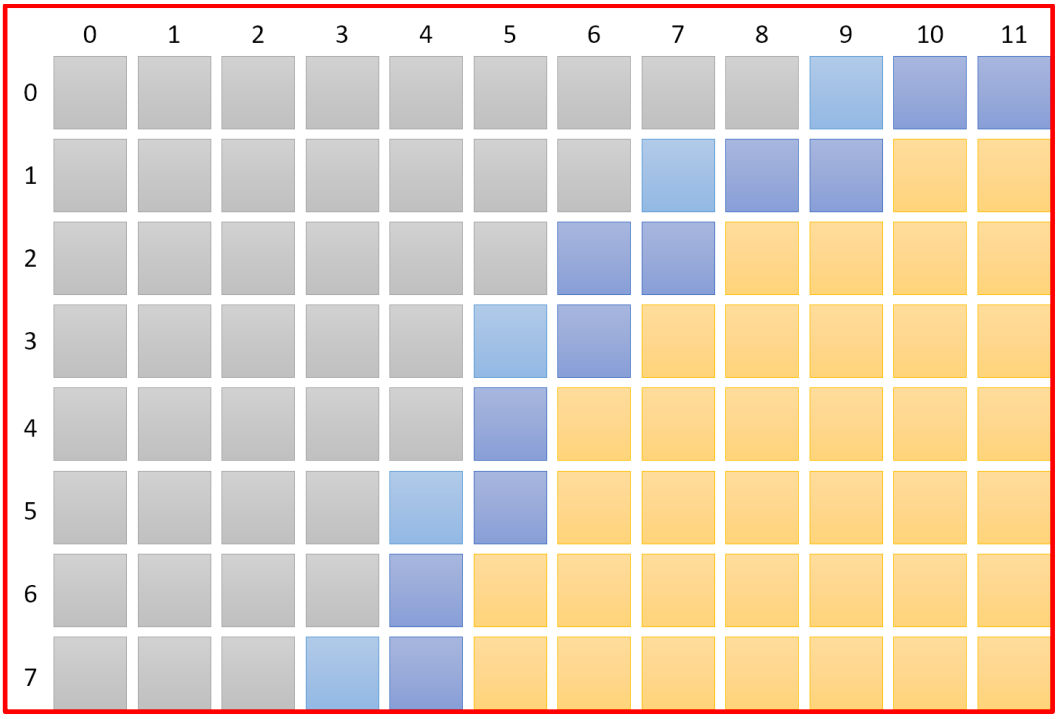

Figure 2: Detail of sample image

Each rectangle in Figure 2 represents an Object of class Pixel. That means they contain three fields: r, g and b; one for each channel. They have been modelled as public fields, because you will have to access them in calculations, and this makes it more readable.

2-Dimensional Pixel array

Figure 3 should give a clear idea of how 2-dimensional arrays actually work in Java. Use this to figure out how to iterate over the different pixels in your code. The terminology used to index a cell are *rows* and *columns* – in that order! *(note: this is opposite of x and y in a Cartesian coordinate system)*

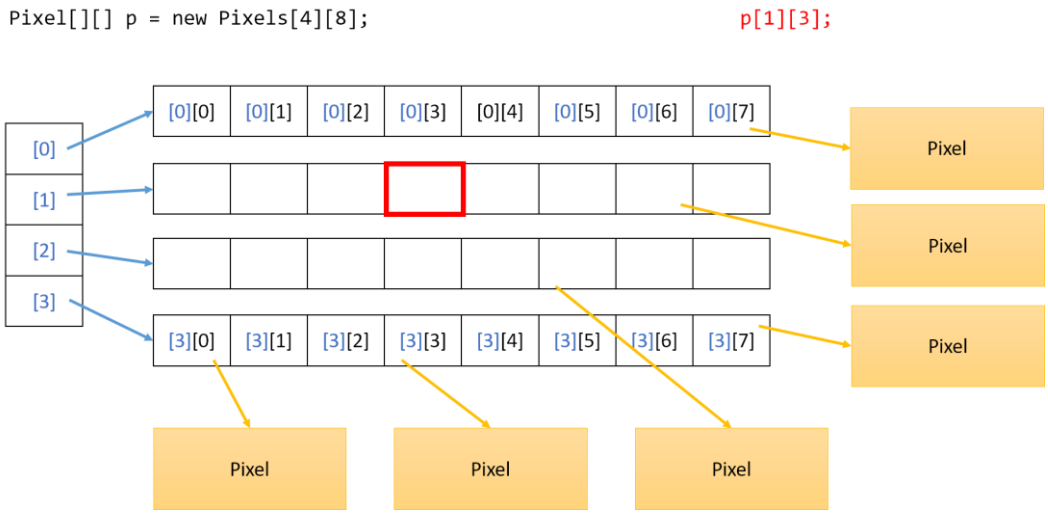

Figure 3: 2D arrays in Java

Image Filters

Our simple image editor will allow us to execute a few image filters on a given image. Image filters work by manipulating the color channels of the pixels of an image. We are working in the RGB color space for this lab session, so our filters will manipulate the RGB channel values.

There are four different filters you will have to develop, they are listed below.

Invert Filter

This filter will invert the channel values of the pixels to which it is applied. This will yield the so called 'negative' of the source image.

Figure 4: Invert sample

Shift Rgb Filter

This filter will shift the channel values forward:

- The new red channel value gets the original green channel value.
- The new green channel value gets the original blue channel value.
- The new blue channel value gets the original red channel value.

Figure 5: Shift RGB sample

Horizontal Mirror Filter

This filter will mirror the source image horizontally.

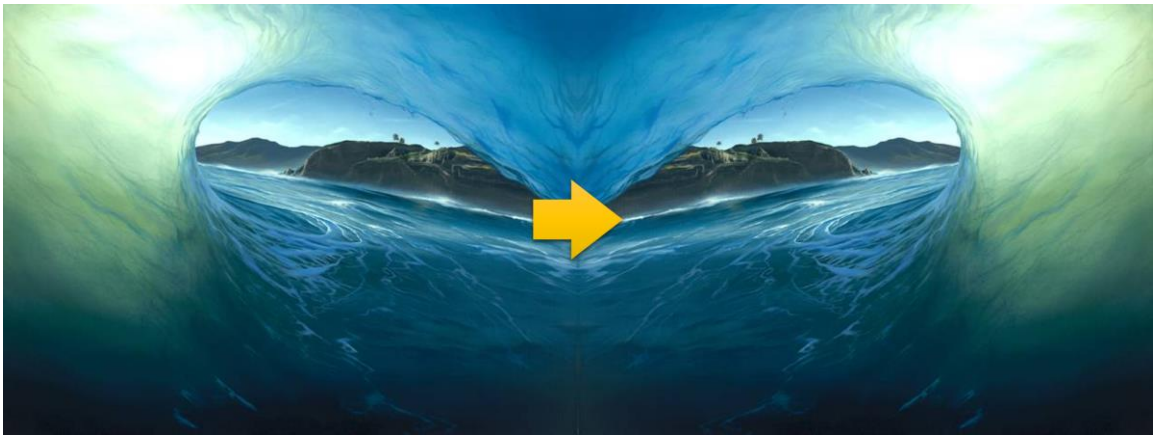

Figure 6: Horizontal mirroring sample

Vertical Mirror Filter

This filter will mirror the source image vertically.

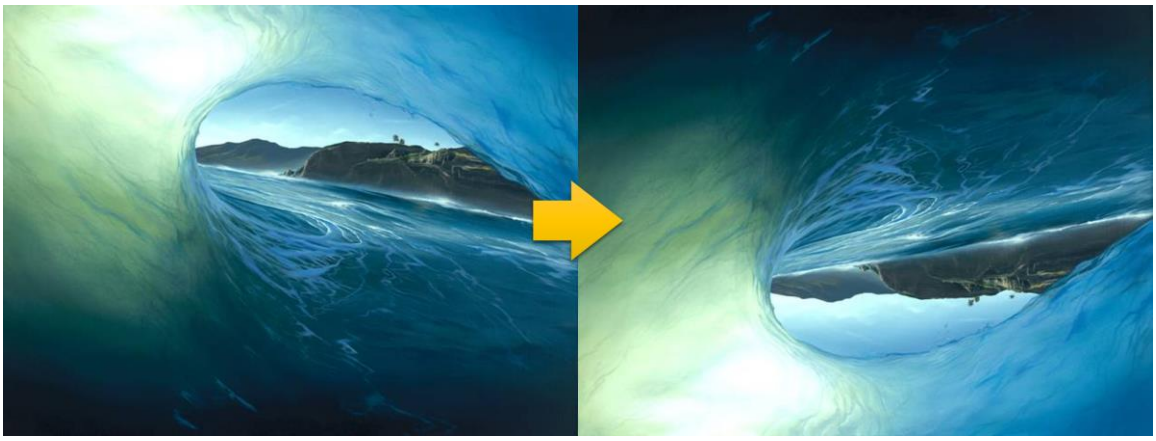

Figure 7: Vertical mirroring sample

Note: Although you got some samples of the filter effects above, note that these filters were applied to the complete image. You will be required to implement filters that can be applied to certain regions of the image only (using the provided Region class).

Commands

We will not just apply filters to our images. Just like in a real image editor, we want to be able to undo the filters that we applied previously. The Command design pattern is a great candidate for this, as it allows us to encapsulate the necessary object references to execute the filter on the required Region of the image. It also allows us to store the command instances, so that we can call their `undo()` method later.

You will have to support four different Commands, one for each Filter. While your Filter implementations only have an `apply(pixels: Pixel[][], region: Region)` method, your Commands will also implement the operation that is required to undo the effect of the associated filter. (Think about the inverse of the Filter effect, and how to achieve that)

ImageProcessor

This is the class that will handle our command history. Use the provided UML in Figure 8 to complete this class. Note that in this implementation we want the `addCommand(command: Command)` method to both add the command to our Stack **ánd** execute the Command.

ImageEditorApplication

This class is where the application's main method resides. There are different tests implemented to test your application in the main method. (Simply comment out the method calls of the ones you don't want to test right now.) They will allow you to test if your filters are working or not.

All the Filter tests start from the 00_start.bmp file in the root directory of your project folder. They apply the filter and write the result to that same directory. After that, the `undoAllCommands()` method is called. This method must be implemented by you (see Assignment section).

This will revert the operation you just did, and again write the resulting image to that same directory. Now you can look at the images in the root directory and see if the start and control image match up, and if the filter effect was correct.

Assignment

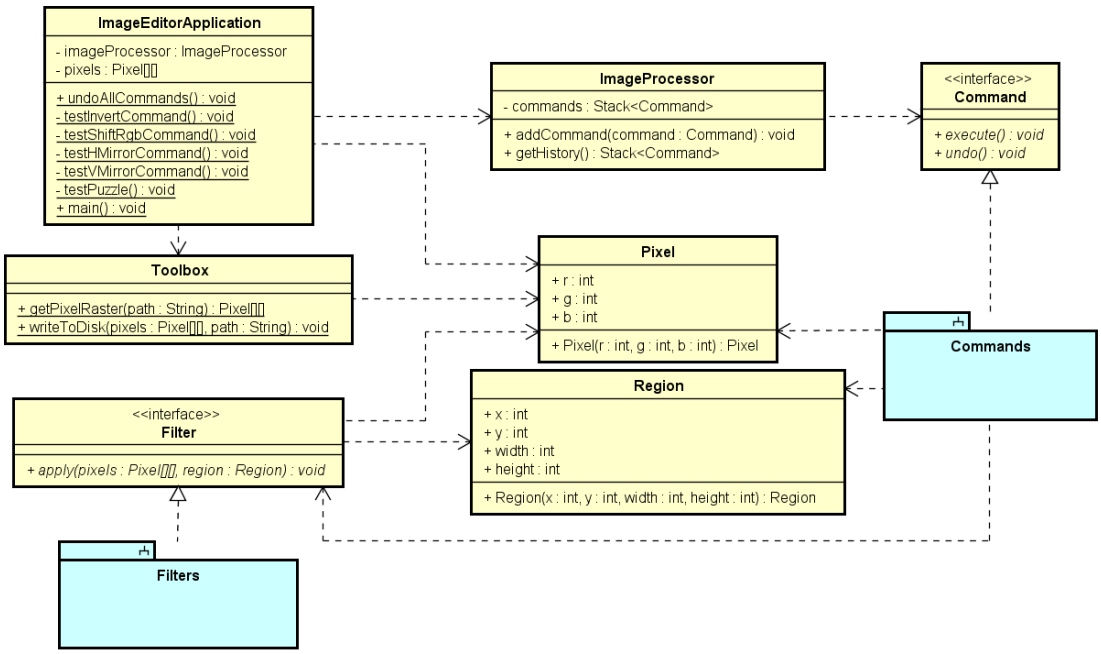

Figure 8: Incomplete UML class diagram of the application

Given the previous incomplete UML of our application (Figure 8), we want you to perform the following steps:

- Complete the ImageProcessor class, as described by the UML in Figure 8
- Implement the `undoAllCommands()` method in the ImageEditorApplication class.
- Implement the complete structure marked in blue as the Filters subsystem.
    - Four Filters: InvertFilter, ShiftRgbFilter, HMirrorFilter, VMirrorFilter
    - Apply the Strategy design pattern, thus making sure that if it was ever necessary, a certain Filter could easily be replaced by another no matter where it was used.
- Implement the complete structure marked in blue as the Commands subsystem.
    - Four Command classes
        - InvertCommand
        - ShiftRgbCommand
        - HMirrorCommand
        - VMirrorCommand
    - Think about how to correctly apply inheritance and/or polymorphism here
- Draw the remaining parts of the class diagram in UML (the blue subsystems). You can use the simple class notation for the already given classes of Figure 8. (square containing only class name, no attributes, no methods)

Submitting your solution

1. Important: labs are solved individually
2. Bundle your source solution files in one zip file named ***lab3_name_surname.zip*** (e.g. *lab3_bruno_volckaert.zip*)
3. **All .java files** need to be included in the zip file!
4. **Your UML class diagram** as requested in the Assignment section **must also be include in the zip file!**
5. Send this zip-file via Minerva dropbox to **Wannes Kerckhove, Thomas Dupont** and **Bruno Volckaert**, deadline for sending this is exactly one week from the start of this lab. I.e. for this lab the deadline is Friday 27th of October at 15:59.