GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computer Science & Applied Physics

B.Sc. in Software Development

Object Oriented Programming - Main Assignment (50%)

A Multithreaded Image Filtering System

1. Overview

You are required to develop a Java API that can apply one or more ***kernel filters*** to an image or a directory of images. The API should uphold the principles of loose-coupling and high cohesion throughout its design by correctly applying abstraction, encapsulation, composition and inheritance. An overview of the system is shown below in Fig. 1.

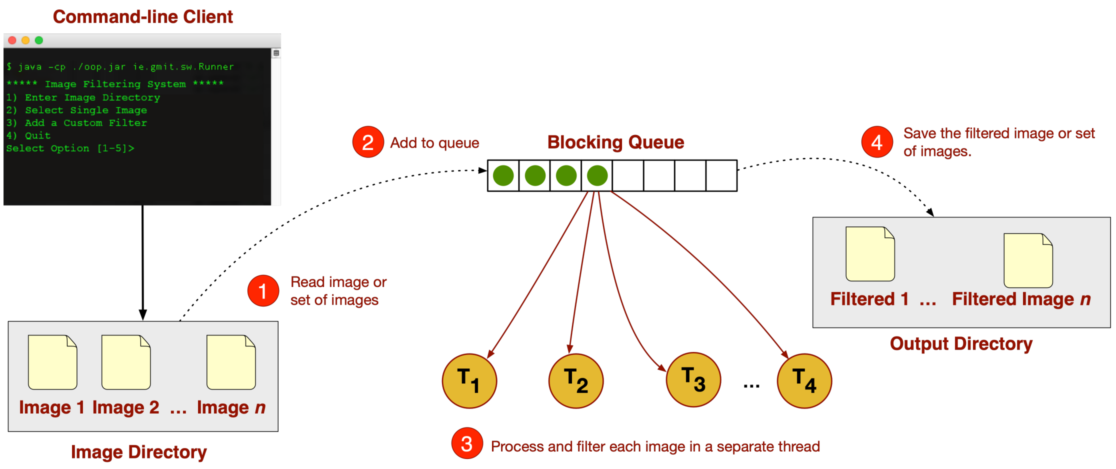

```
$ java -cp ./oop.jar ie.gmit.sw.Runner
***** Image Filtering System *****
1) Enter Image Directory
2) Select Single Image
3) Add a Custom Filter
4) Quit
Select Option [1-5]>
```
Command-line Client
1 Read image or set of images
2 Add to queue
Blocking Queue
3 Process and filter each image in a separate thread
4 Save the filtered image or set of images.
Image 1 Image 2 ... Image n
Image Directory
T1 T2 T3 ... T4
Filtered 1 ... Filtered Image n
Output Directory

Fig. 1

All the resources required for the assignment are available on Moodle. Your submission will be scored on a preview-enabled Java 15 JVM, so feel free to use any of the new features of the language like records or sealed types.

2. Minimum Requirements

- Use the package name ***ie.gmit.sw***. The application must be deployed and runnable using the specification in Section 3.
- Create a console-based ***menu-driven UI*** to input the path to an *image directory* or the path to a single file and an output path to write out processed images. Any other parameters should also be input through the menu-driven UI. *Do not hard code any files, drive letters or system paths* and do not use a GUI!
- The application should be ***fully thread***ed and use *Runnable*, *Callable* and *Future* types where appropriate. You should give careful consideration to using the suite of data structures in the ***java.util.concurrent*** package, e.g. a *BlockingQueue*.
- Provide a **UML** diagram of your design and **JavaDoc** your code.

3. Deployment and Delivery

The project must be submitted by midnight on Friday 8th January 2021. Before submitting the assignment, you should review and test it from a command prompt on ***a different computer*** to the one that you used to program the project.

- The project must be submitted as a Zip archive ***(not a 7z, rar or WinRar file)*** using the Moodle upload utility. You can find the area to upload the project under the "*A Multithreaded Image Filtering System (50%) Assignment Upload*" heading of Moodle. *Do not add comments to the Moodle assignment upload form.*

- The Zip archive should be named <*id*>.zip where <*id*> is your GMIT student number.

- Do not include sample images with your submission or any other test data.

- The Zip archive should have the following structure (do NOT submit the assignment as an Eclipse project):

Marks	Category
oop.jar	A Java **archive** containing your API and runner class with a main() method. You can create the JAR file using Ant or with the following command from inside the "bin" folder of the Eclipse project: **jar –cf oop.jar *** The application should be executable from a command line as follows: **java –cp ./oop.jar ie.gmit.sw.Runner**
src	A directory that contains the packaged **source code** for your application.
README.pdf	A **PDF file** detailing the main features of your application in **no more than 300 words**. All features and their design rationale must be fully documented. You should consider also including the UML class diagram in this document to help explain your design clearly. Marks will only be given for features that are described in the README.
design.png	A UML **class diagram** of your API design. The UML diagram should only show the relationships between the key classes in your design. Do not show private methods or attributes in your class diagram. You can create high quality UML diagrams online at **www.draw.io**.
docs	A directory containing the **JavaDocs** for your application. You can generate JavaDocs using Ant or with the following command from inside the "src" folder of the Eclipse project: **javadoc -d [*path to javadoc destination directory*] ie.gmit.sw** Make sure that you read the JavaDoc tutorial provided on Moodle and comment your source code correctly using the JavaDoc standard.

4. Marking Scheme

Marks for the project will be applied using the following criteria:

Element	Marks	Description
Structure	10	*All or nothing.* The packaging and deployment correct. All JAR, module, package and runner-class names are correct.
README	7	All features and their design rationale are fully documented.
UML	10	Class diagram correctly shows all the important structures and relationships between types.
JavaDocs	8	All classes are fully commented using the **JavaDoc** standard and generated docs available in the *docs* directory.

Robustness	40	The fully threaded application executes perfectly, without any manual intervention, using the specified execution requirements.
Cohesion	10	There is very high cohesion between packages, types and methods.
Coupling	10	The API design promotes loose coupling at every level.
Extras	5	Only relevant extras that have been fully documented in the README.

You should treat this assignment as a project specification. Each of the elements above will be scored using the following criteria:

- 0–30% **Fail:** Not delivering on basic expectations
- 40-59% **Mediocre:** Meets basic expectations
- 60–79% **Good:** Exceeds expectations.
- 80-89% **Excellent:** Demonstrates independent learning.
- 90-100% **Exemplary:** Good enough to be used as a teaching aid

5. Using Kernels to Filter Images

Images are typically stored on a computer in a compressed lossy (JPEG) or lossless (GIF, PNG) format. When an image is opened in a graphics application like *Photoshop* or *GIMP*, it is usually rasterized and converted into a bitmap that uses a 32 bit integer to represent a pixel. Each pixel therefore can encode the four ARGB (alpha, red, green, blue) channels using 1 byte, creating a range of [0...255] for each channel. The 24 bits of the 3 RGB channels can be used to represent $256^3=2^{24}=16{,}777{,}216$ different colours, with the alpha channel controlling the transparency of the image. A raw bitmap image of dimensions 800 x 600 pixels will contain a total of 480,000 pixels.

Among the many features of a graphics application like *Photoshop* is the ability to filter an image to create some kind of an effect. These effects include edge detection, blurring, sharpening and embossing and are created by applying a matrix of values to each pixel in the image. The matrix of values is called a **convolution matrix**, a **kernel**, or a **kernel matrix**. A convolution is a mathematical operation that blends two functions together.

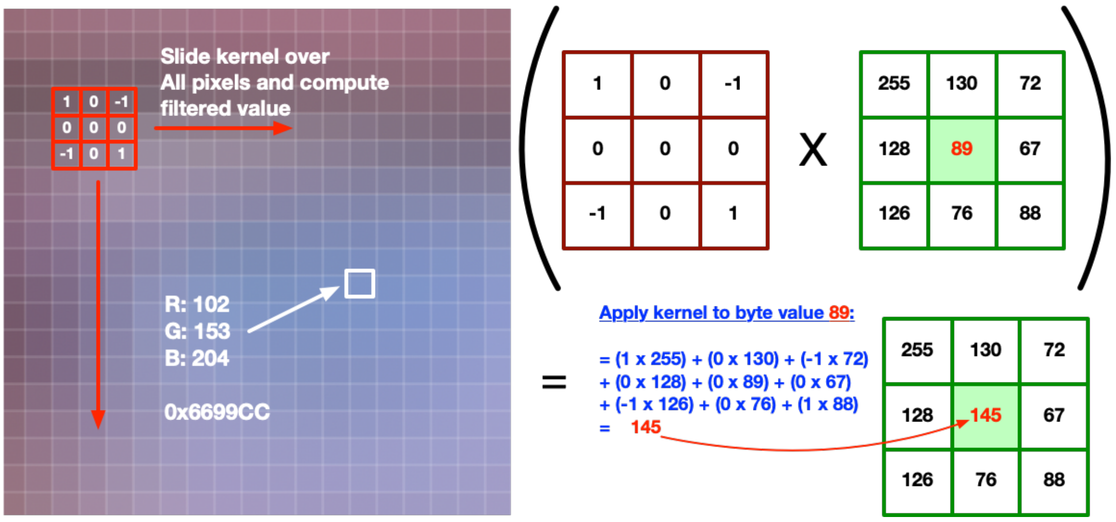
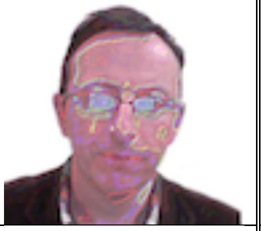
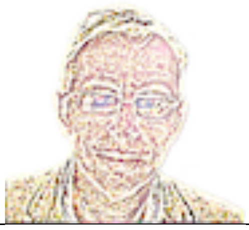
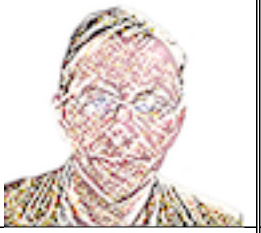
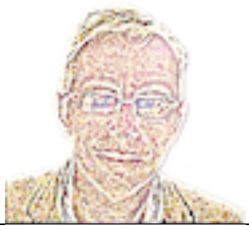
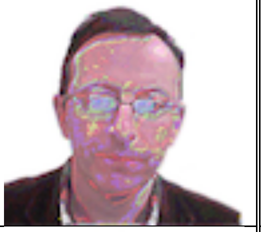
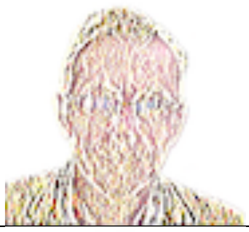
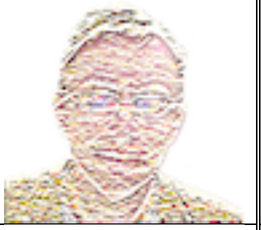
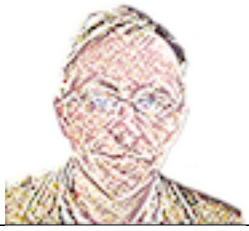
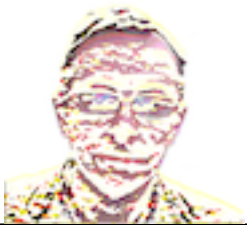
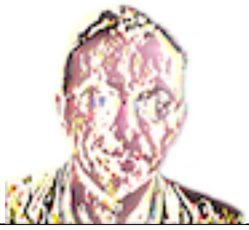
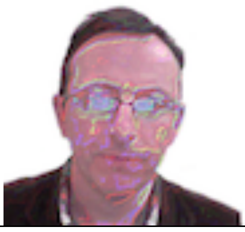

Slide kernel over
All pixels and compute
filtered value
R: 102
G: 153
B: 204
0x6699CC
Apply kernel to byte value 89:
= (1 x 255) + (0 x 130) + (-1 x 72)
+ (0 x 128) + (0 x 89) + (0 x 67)
+ (-1 x 126) + (0 x 76) + (1 x 88)
= 145

Fig. 2

The matrix itself is just a 2D array of numbers that slides across the pixels of the rasterized image and is used to compute a new value for each pixel. In the image in Fig. 2, the 3 x 3 kernel can be used to detect the edges in an image *by changing the value of a pixel using the weights in the kernel and the immediate surrounding pixels in the image*. In the example, the pixel value

89 is converted to 145 by the convolution. The same operation is applied to every pixel in the image, including edge pixels at the top, bottom and sides. If the image is in *greyscale* format, all pixels are encoded with 1 byte and have a value in the range [0..255] and the convolution can be performed by tiling over the image as shown in Fig. 2. If the image is in RGB colour, the convolution needs to be applied separately to each colour channel.

The table below shows a set of convolution kernels and their effect on an image of a charming gentleman that you may be familiar with. In all cases, the kernels are 3 x3 matrices and all except one use integer values (the values in the *Box Blur* represent $^1/_9$ and the total of the matrix sums to one… obviously). The kernel matrices do not have to be the same size, e.g. a *Gaussian Blur* can be applied with either a 3x 3 or a 5 x 5 matrix.

Kernel Filter	Effect	Kernel Filter	Effect
Identity {0, 0, 0}, {0, 1, 0}, {0, 0, 0}		**Edge Detection** {-1, -1, -1}, {-1, 8, -1}, {-1, -1, -1}	
Edge Detection {1, 0, -1}, {0, 0, 0}, {-1, 0, 1}		**Laplacian** {0, -1, 0}, {-1, 4, -1}, {0, -1, 0}	
Sharpen {0, -1, 0}, {-1, 5, -1}, {0, -1, 0}		**Vertical Lines** {-1, 2, -1}, {-1, 2, -1}, {-1, 2, -1}	
Horizontal Lines {-1, -1, -1}, {2, 2, 2}, {-1, -1, -1}		**Diagonal 45 Lines** {-1, -1, 2}, {-1, 2, -1}, {2, -1, -1}	
Sobel Horizontal {-1, -2, -1}, {0, 0, 0}, {1, 2, 1}		**Sobel Vertical** {-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}	
Box Blur {0.111, 0.111, 0.111}, {0.111, 0.111, 0.111}, {0.111, 0.111, 0.111}			

6. Reading and Writing Images in Java

The class *java.awt.image.BufferedImage* provides a useful abstraction of a double-buffered image that can be manipulated in memory. The utility class *javax.imageio.ImageIO* allows us to read and write PNG, GIF and JPEG files as shown below:

```
//Read in an image and convert to a BufferedImage
BufferedImage image = ImageIO.read(new File("picture.png"));
System.out.println(image); //This writes out a lot of useful meta-data about the image.

for (int y = 0; y < image.getHeight(); y++) { //Loop over the 2D image pixel-by-pixel
  for (int x = 0; x < image.getWidth(); x++) {
    int pixel = image.getRGB(x, y); //Get the pixel at an (x, y) coordinate

        image.setRGB(x, y, 0x00FF0000); //Set the pixel colour at (x, y) to red

        //We can get the RGB colour channels out of a 32-bit int as follows:
        int red = (pixel >> 16) & 0xff;
        int green = (pixel >> 8) & 0xff;
        int blue = pixel & 0xff;

        //We can re-create a 32-bit RGB pixel from the channels as follows;
        int rgb = 0;
        rgb = rgb | (red << 16);
        rgb = rgb | (green << 8);
        rgb = rgb | blue;
  }
}
//Write out the file in PNG format
ImageIO.write(image, "png", new File("out.png"));
```

Assuming that we have a convolutional kernel called ***filter***, we can iterate over it and process a pixel from the (x, y) position of an image as follows. Note that, although this implementation doesn't even try to apply the filter to the edges, it's still good enough to create the filtered images shown in the table above.

```
public int apply(int pixel, int x, int y){
  int element = pixel;
  for (int row = 0; row < filter.length; row++) {
    for (int col = 0; col < filter[row].length; col++) {
      try {
        //This will cause an exception if we overrun the edges of the image
        element = image.getRGB(x + col, y + row);
        //Do lots of cool stuff here…. Maybe use some of the code above…
      } catch (Exception e) {
        continue;  //Ignore any exception and keep going. It's good enough ☺
      }
    }
  }
  return element;
}
```