Homework - 1

Q1) Let's say you developed a solution for an object oriented design problem. Does that design solution constitute a design pattern? What makes a design solution a "design pattern"? (**5 points**)

You have been appointed team leader to a new project. The customer wants you to develop an image processing solution (like photoshop); a program that can apply basic filters to a given image.

You have talked with the customer's representative about her requirements, and you have taken the following notes:

a) the program must be able to apply smoothing and edge detection filters to a given image.
b) a filter can do only one thing: it processes a given image object and returns the new filtered image object.
c) there are multiple algorithms for implementing each filter:
for the edge detection filter: the morphoInt and morphoExt algorithms,
for the smoothing filter: the median and average algorithms,
and your application must be able to switch dynamically between algorithms for every filter.
d) new filters and new algorithms might be added in the future by the users of your software, so your design must be expandable and provide an easy to use API.

As a seasoned developer you immediately see the opportunity to apply a design pattern for solving this problem efficiently and effectively. Which one? (Hint: it starts with "S" and ends with "trategy")

Q2) How would you apply the Strategy pattern to this problem? Design the corresponding UML diagram (assuming that your implementation language will be Java and that image display and loading functionalities are provided by some external library containing the Image class, Display2D and Load classes). Be careful with the arrow types. (**20 points**)

Q3) Explain how your design satisfies each of the customer's requirements. (**20 points**)

Q4) Code your design (see below for implementation details). Make sure you provide a main method where all filters are demonstrated and dynamic algorithm switching is realized. (**30 points**)

Q5) Your image filtering application is so successful that a digital surveillance company has come to you with a project. They have a camera security system, that they install to residential houses, in order to monitor events (theft, trespassing, etc). The cameras are equipped with hardware based real-time change detectors. The company wants to use your basic image filtering solution from Q4 to detect fire and burglary. In case of fire your app should call the fire department (using the `contactFireDepartment()` method), or if it is a burglary it should contact the police (using the `contactPolice()` method).

They already have some software in place. Whenever two frames of the video stream are different, the hardware calls the following method automatically, but they need you to process the data:

```
class FancyDetector{
      public void changeDetected(Image frame){// todo ... }
}
```

Fortunately, you already have `FireDetector` and `BurglaryDetector` classes, that are equipped each with the following method, that returns true in case of the corresponding event (fire for `FireDetector` and burglary for `BurglaryDetector`), or false otherwise:

```
boolean detect(Image frame)
```

Moreover, your solution should be flexible enough to add easily additional controls in the future (for instance to check for any animals that might have triggered the system).

Plus, the hardware system is very sensitive. It can trigger the `changeDetected` method even when a leaf passes in front of the camera. You have developed a method:

```
boolean sensitivityCheck(Image frame)
```

that checks whether a frame has indeed a frame with a valid event, or a false alarm. You must use it effectively to reduce the number of false alarms and unnecessary calls to the fire and police stations.

You immediately realize that this problem can be *easily* solved with the Observer design pattern. How? Can you use JDK's Observer/Observable implementation or should you implement your own design? Draw the UML of your classes after applying the pattern and explain how it satisfies ALL of the customer's requirements. You are free to add any methods/classes/interfaces that you deem necessary. (**25 points**)

Good luck.

Appendix

A digital image is commonly a rectangular grid of pixels, where every pixel represents an intensity point on the image as a non-negative integer in [0, 255] (Fig. 1).

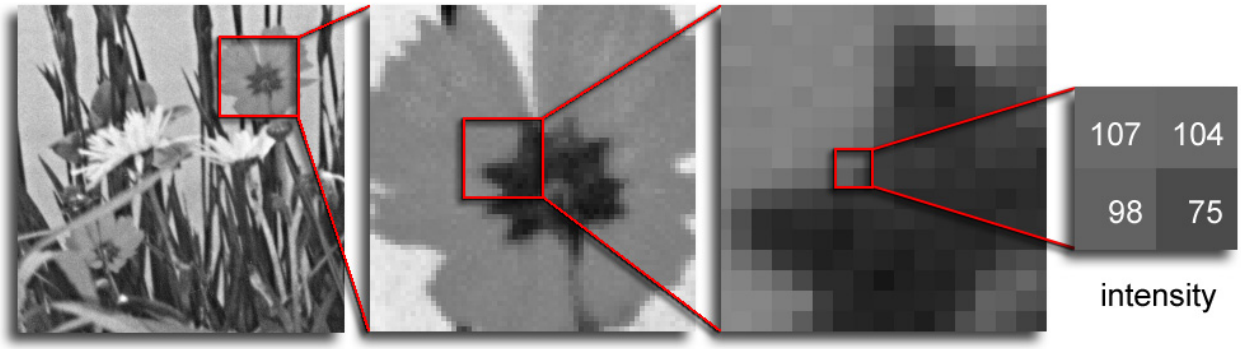

Figure 1

An image is represented through the class Image.

Loading an image from your drive:

```
Image img = Load.invoke("path/to/file/myImage.png");
```

Copy constructor (creates a black image "copy" of the same size as "img"):

```
Image copy = img.newInstance(false);
```

Display an Image object "img" visually

```
Display2D.invoke(img, "windowTitle");
```

Reading the pixel intensity of image "img" at row 300 and column 200:

```
int p = img.getXYByte(200, 300);
```

Setting the pixel intensity of image "img" at row 300 and column 200 to a new value 128.

```
img.setXYByte(200, 300, 100);
```

Filtering is easy: all you need is a “mask”. For the sake of clarity we will assume that our mask **is always** an empty (i.e. with no values) matrix of size 5x5:

How to filter? Assume **img** is your input image. First you create a copy of img: **output**. **Then you have to calculate the new value of every pixel of img and write it to output.** How? By propagating the mask over **img**. Say you want to calculate the new value of the pixel at row 300 and column 200 that has a value 128. Then you place the mask on top of **img**, so that the mask is centered on that pixel of row 300 and column 200.

…	…	…	…	…	…	…
…	100	101	99	102	103	…
…	110	116	115	118	118	…
…	109	109	**128**	122	123	…
…	99	126	132	137	131	…
…	127	123	130	135	138	…
…	…	…	…	…	…	…

at the original **img** object

…	…	…	…	…	…	…
…						…
…						…
…			**p**			…
…						…
…						…
…	…	…	…	…	…	…

at the **output** object

There are now 25 values of **img** under the mask. What you do afterwards depends on the filter.

Smoothing-Average: you calculate the average of the 25 values, round it to an integer and set that value (**p**) as the new intensity of the pixel at row 300 and column 200 at **output**.

Smoothing-Median: you calculate the median of the 25 values, and set that value (**p**) as the new intensity of the pixel at row 300 and column 200 at **output**.

Edge detection-morpholnt: you calculate the minimum of the 25 values, and set the difference of the current pixel value from the minimum as the new intensity of the pixel at row 300 and column 200 at **output**. For instance if the processed pixel is 128, and the minimum under the mask is 99, then the new value (**p**) will be 128-99 = 29.

Edge detection-morphoExt: you calculate the maximum of the 25 values, and set the difference of the current pixel value from the maximum as the new intensity of the pixel at row 300 and column 200 at **output**. For instance if the processed pixel is 128, and the maximum under the mask is 138, then the new value (**p**) will be 138-128 = 10.