FACULTY OF ENGINEERING
AND ARCHITECTURE

DEPARTMENT INTEC
RESEARCH GROUP IBCN

Geavanceerde softwareontwikkeling

Lab Session IV

Observer, Adapter, Decorator, Façade & Factory

GHENT
UNIVERSITY

Introduction

For this fourth Lab session, we'll revisit the image editing application of the previous session and integrate it with a graphical user interface (see Figure 1). This GUI is built using JavaFX and its main controller employs the *façade design pattern* to relay all user actions to an engine implementation that performs the actual image operations. Once these operations are completed, the GUI is notified using the *observer design pattern*, so the view can be redrawn based on the updated image data.

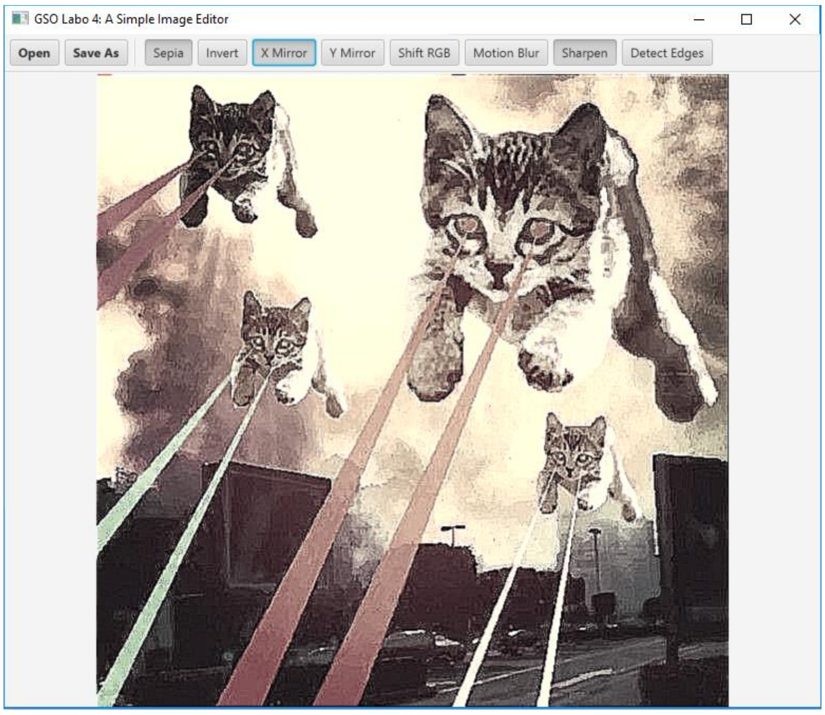


Figure 1: Screenshot of the editor UI

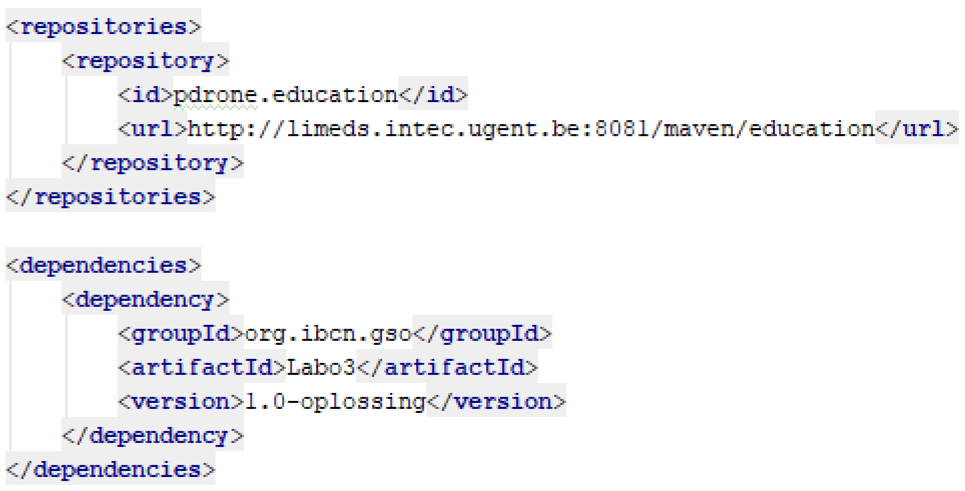
The image operations are applied on the entire source image using the *decorator design pattern*. Each effect acts as a layer on top of the image that is already there and effects can be undone by removing the associated decorator from the rendering chain.

Four new effects are introduced of which Motion Blur, Sharpen and Detect Edges can be implemented using convolution matrices, while a simple algorithm is provided for the Sepia effect. The Filter implementations of Lab 3 will be reused for the remaining effects, but we will need to apply the *adapter design pattern* to make them compatible with the decorator-based rendering chain.

Finally, the *factory design pattern* is used to integrate both the new effects and the adapted filter effects with the façade implementation in a uniform way.

Important: to make sure that you can all start from the same baseline for this Lab, we have provided our own solution for Lab 3 that you have to use instead of copying your own files.

Our Lab 3 solution is automatically imported in your project using Maven, a build and dependency manager tool for Java and other programming environments. This is done by adding the following information to the POM file (where Maven stores your project information):



```xml
<repositories>
    <repository>
        <id>pdrone.education</id>
        <url>http://limeds.intec.ugent.be:8081/maven/education</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.ibcn.gso</groupId>
        <artifactId>Labo3</artifactId>
        <version>1.0-oplossing</version>
    </dependency>
</dependencies>
```

Figure 2: pom.xml

The repository element specifies where the web server can be found that will host our Lab 3 build as a deployable Java library. The dependency element declares a dependency of your Lab 4 project on a specific version of the Lab 3 solution that we've provided.

What follows is a step-by-step guide of how we can get this graphical image editor to work.

Step 1: Design the Render Chain

In Lab 3 we've relied upon the Strategy and Command design patterns to setup our filter chain. You might have noticed that the requirement for providing an `undo()` method for each Command, restricted us to easily reversible Filter implementations (unless of course we would have stored additional state information for each command).

In Lab 4 we will use an alternative approach based around the decorator pattern that doesn't suffer from this restriction (but as a drawback requires more processing power).

Add an interface `ImageProvider` to the project that declares a single method:

- `Pixel[][] getImage()`: returns the image provided by this `ImageProvider` instance as a 2D Pixel array.

Provide a basic implementation class for this interface called `FileImageProvider` that loads the 2D Pixel array from a file (specified in the constructor of the class) **each time** the `getImage()` method is called. Use the Lab 3 `Toolbox` class to process the File (you may **ignore** the Exception thrown by the `getPixelRaster` method by adding a try / catch block with an empty catch section).

Next, we complete the decorator pattern by adding an abstract class `ImageProviderDecorator` that also implements the interface `ImageProvider` and has the following methods:

- `ImageProviderDecorator(ImageProvider provider)`: the constructor for `ImageProviderDecorator` that takes the `ImageProvider` instance to which this decorator should be applied to as argument.
- `ImageProvider getProvider()`: returns the provider for this decorator.
- `void setProvider(ImageProvider provider)`: set a new provider instance to apply this decorator to.
- `EffectType getEffectType()`: returns information about the type of effect that is implemented by this decorator.

Notice that `ImageProviderDecorator` does not implement `getImage()`, as this method will be supplied by the concrete decorator classes (implementing a specific image effect). Using the same rationale, `getEffectType()` should be an abstract method.

To get accustomed to this decorator type, we will now implement a first effect by adding the class `SepiaEffect` that extends from `ImageProviderDecorator`. Make sure that you correctly initialize the super class by providing an `ImageProvider` instance (to indicate what the image source is for this decorator). The `getEffectType()` method should always return the enum value **SEPIA**. Implement the `getImage()` method by acquiring the 2D Pixel array from the specified provider and by then applying the following algorithm as a filter (pseudo-code):

```
for each row in the image
    for each column in the image
        pixel = image[row][col]
        r = pixel.r
        g = pixel.g
        b = pixel.b
        pixel.r = min(r * 0.393 + g * 0.769 + b * 0.189, 255)
        pixel.g = min(r * 0.349 + g * 0.686 + b * 0.168, 255)
        pixel.b = min(r * 0.272 + g * 0.534 + b * 0.131, 255)
```

Step 2: Integrate the existing Lab 3 Filters

An important aspect of programming is reusing existing code and libraries instead of solving the same problems over and over again. With that in mind, we'll adapt our Filter implementations from Lab 3 to be compatible with the decorator render chain.

To do this, add a new class `FilterAdapter` to your project, which will act as the bridge between the `Filter` classes and the decorator render chain by extending from `ImageProviderDecorator`.

The constructor of our `FilterAdapter` class will take on two arguments:

- The `ImageProvider` instance that will act as the image source.
- The Lab 3 `Filter` instance that is to be used.

Because `FilterAdapter` extends from `ImageProviderDecorator`, two methods must be provided:

- `Pixel[][] getImage():` returns the processed image based on the apply method of the provided `Filter` instance.
- `EffectType getEffectType():` returns the `EffectType` value that matches the provided `Filter` instance[1]: SEPIA, INVERT, HORIZONTAL_MIRROR, VERTICAL_MIRROR, RGB_SHIFT, MOTION_BLUR, SHARPEN or DETECT_EDGES.

Step 3: Implement the Effect Factory

Now that the decorator chain is defined and we have a way to integrate the Lab 3 Filter implementations, everything is in place to move on to the implementation of the Effect Factory.

Add an interface defining the Effect Factory called `ImageProviderDecoratorFactory` to your project that specifies the following method:

- `ImageProviderDecorator create(ImageProvider source, EffectType effectType):` returns an `ImageProviderDecorator` instance that decorates the specified source to apply the requested `EffectType`.

The goal of this factory is to separate the implementation details of constructing the decorator instances (that are associated with the various supported effects) from the façade implementation that should only be concerned with applying the effects and not with how they are setup.

[1] Tip: use the Java `instanceof` operator to determine the Filter type.

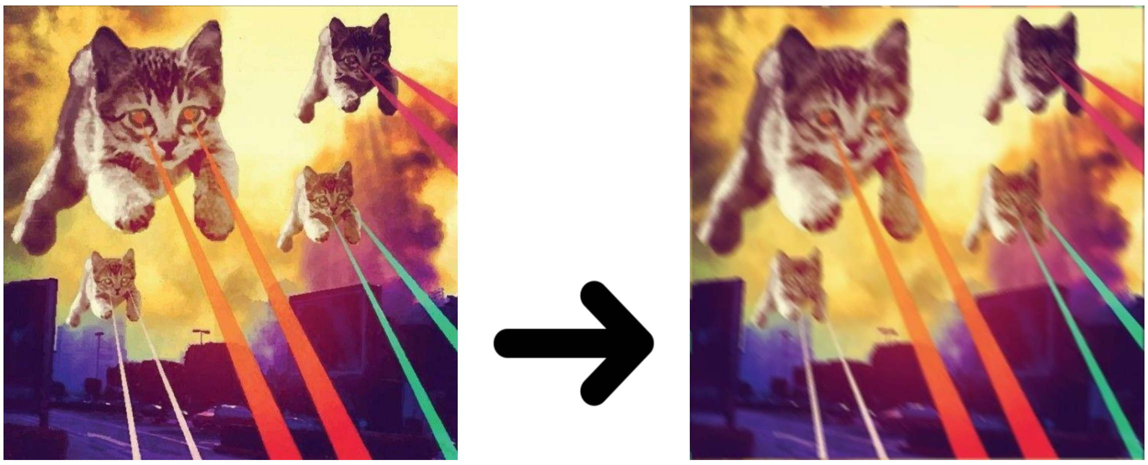
Next, add a class called `EffectFactory` that implements the create method from the `ImageProviderDecoratorFactory` interface. Try to return a correctly setup decorator instance for each EffectType:

- **SEPIA**: return an instance of the SepiaEffect class
- **INVERT, HORIZONTAL_MIRROR, VERTICAL_MIRROR & RGB_SHIFT**: return an instance of `FilterAdapter` configured with the appropriate Lab 3 filter instance.
- **MOTION_BLUR, SHARPEN, DETECT_EDGES**: return an anonymous implementation of `ImageProviderDecorator` that uses the appropriate convolution matrix and the `applyConvolution` method of the provided `EffectsUtil` class.

The following subsections explain the three convolution transformation and the matrix and factor parameters to use. For those interested in more information on convolution, we refer to the Image Filtering pdf document that was posted on Minerva (this document is not required to be able to solve this lab session though).

Motion Blur

The motion blur effect simulates movement of the camera while taking a picture.

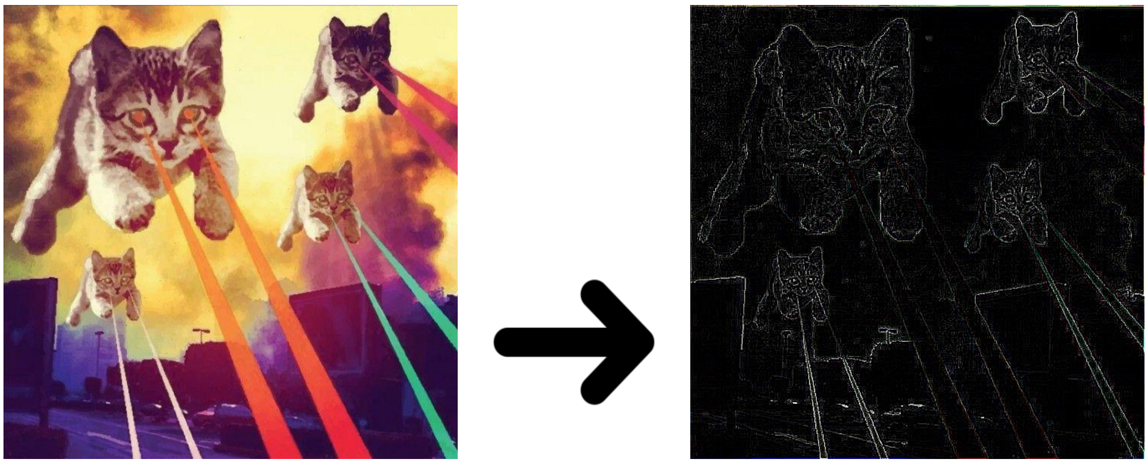


Use the following convolution matrix with factor `1.0 / 9.0`:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Detect Edges

The detect edges effect uses a mask to detect and isolate the edges of the input image.

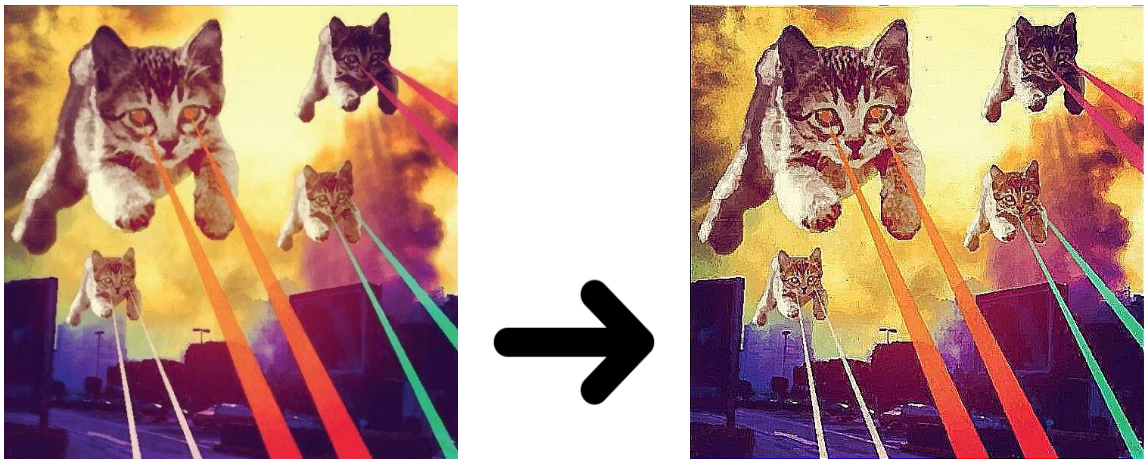


Use the following convolution matrix with factor `1.0`:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Sharpen

The sharpen effect adds the result of the detect edges filter to the source image in order to accentuate the edges (and thus creating the illusion of a sharper picture).



Use the following convolution matrix with factor `1.0`:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Step 4: Complete Image Load and Save As

Now we can move on to the façade implementation in the class `EngineImpl`. We'll start with the `load` and `saveAs` methods that take care of the File I/O for the application. `Load` is implemented using the `FileImageProvider` we have implemented earlier in combination with a new attribute in your `EngineImpl` class that will represent the decorator-based render chain.

For `saveAs` you can use the `Toolbox` class that was provided for Lab 3. You may ignore the exception that is thrown by the `writeToDisk` method.

Step 5: Complete Observable Code

You might have noticed that the `EngineFacade` interface also specifies an Observable for `ImageObserver` instances. The idea is that the `EngineFacade` implementation can easily notify all interested parties each time the represented image changes.

Implement the public methods `register(ImageObserver observer)`, `unregister(ImageObserver observer)` and add a protected method `notifyObservers()` that is called internally each time the image is updated.

Step 6: Complete Toggle Effect Code

In order to get the application to work, we just need to complete the toggle method. This method is called each time a toggle effect button is clicked in the GUI with a parameter `effectType` that corresponds with the effect the user wants to enable or disable.

Your code will have to decide whether to add the requested effect or remove it, based on the current state of your decorator chain. If an instance of `ImageProviderDecorator` with the desired `EffectType` is already present in the chain, you must remove it. Otherwise (when the end of the chain was reached without finding such an instance) a new decorator is created for the requested effect and applied to the rest of the chain.

Do not forget to notify the registered `ImageObserver` instances when the toggle method is completed!

Pseudo code for this step:

```
providerRoot //Deze variabele wordt bijgehouden in EngineImpl en stelt de eerste provider in de keten voor.

toggle(type):
    removed = null //Deze variabele zal de verwijderde decorator in de chain bevatten (indien gevonden)

    current = providerRoot //We starten met zoeken in de root
    previous = null //We houden de vorige decorator in de chain bij

    //Zoeken naar het gevraagde type (zolang het beschouwde element een decorator is en we nog niks gevonden hebben)
    while current is een decorator && removed == null:
        if type van current is gelijk aan gevraagde type:
            removed = current; //We hebben de te verwijderen decorator gevonden

            //Nu nog verwijderen uit de chain:
            if previous != null:
                //Verwijder de gevonden decorator uit de chain
                provider voor previous = provider van current
            else:
                //Nieuwe root toekennen want te verwijderen element was root
                providerRoot = provider van current

        //Verplaats onze cursor in de chain
        previous = current
        current = provider van current


    //Als er geen decorator verwijderd is, voegen we ze toe:
    if removed == null:
        providerRoot = factory.create(providerRoot, type)

    notify alle observers
```

Submitting your solution

1. Important: labs are solved individually
2. Bundle your source solution files in one zip file named ***lab4_name_surname.zip*** (e.g. *lab4_bruno_volckaert.zip*)
3. **All .java files** need to be included in the zip file!
4. **An UML class diagram** that clarifies the relationship between the classes that you've implemented for this Labo **must also be included 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 Friday 10th of November at 15:59.