Project Advanced Object-Oriented

Jesper Holmblad, Fredrik Mårtensson

2018-05-25

Introduction 2
Design 3
Testing 5
Problems 6
Result 7

Introduction

This project aims to introduce basic object oriented programing structures by implementing a user friendly system for photo editing. The requirements of the project is to be able to create a photo editor that allows the user to create, edit and save photos on demand. The project should be built on an object oriented foundation as per what the course have taught. This includes implementing patterns, framework, easy navigation through GUI, abstract methods and interfaces are a few of the requirements that are to be included during the project. Beside technical requirements there are a few additional goals to be implemented such as: ease of implementing extensions and an option for editing properties of filters.

Design

Building a project structure is the most important step for a good foundation. By dividing the project into smaller parts the different components can be recognized as design patterns. These patterns are then reassembled into a fully-fledged program. The framework is one of the first parts that gets constructed to allow functionality that would fit the requirements. The framework includes functions to create a frame that serves as a container to organize the components: filter list, filter GUI, Image panel, and the menu bar items. The components aims to create an easy and comfortable interface to utilize the different features of the software.

As mentioned the project use different design patterns that are common within object oriented languages. Examples of common design patterns are: template pattern, strategy pattern, prototype pattern, observer pattern and decorator pattern.

The **template pattern** shown in figure 1 works between an Abstract class and its children. For instance AbstractFilter and all the filters. Compared to working directly with the filter interface the developer doesn't have to re-implement all the interface methods when a new filter is added. This way code redundancy is reduced which gives cleaner and more consistent code. The AbstractFilter holds default definitions for "getGUI" and "toString" but in the event that the developer need a more specific implementation these can be overridden.

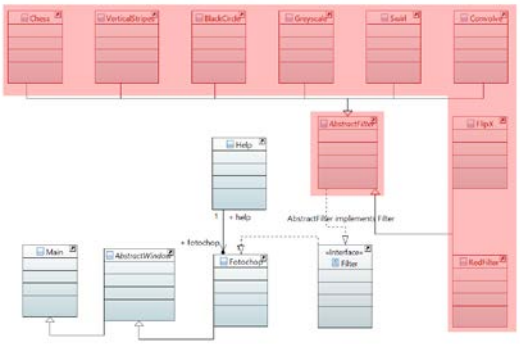

Figure 1 A example of how template is implemented into the application

The **Strategy pattern** shown in figure 2 is what would be if a filter template is avoided in other words all filters would have to implement the filter interface. Using only this pattern code redundancy would be at large but the reason for strategy pattern is different from the template pattern. Here we aim to tag classes so that other classes know there's a minimum set of methods available. This allows classes to be treated the same in the context described by the interface. Similar to ad hoc polymorphism for methods but for classes instead.

Figure 2 A example of how strategy is implemented into the application

Another common pattern is the **prototype pattern** shown in figure 3**.** This is used when we want to create instances of a set of classes unknown to the parent class. In order for the parent to know what set of classes is available a

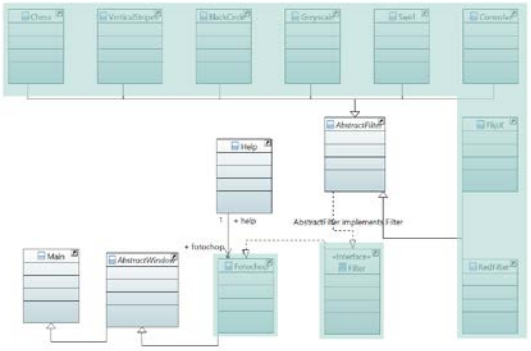

Figure 3 A example of how the prototype pattern is implemented into the application

set of prototypes is provided. This set could either be: hardcoded, a parameter or even determined at runtime by utilizing a ClassLoader.

The **Observer pattern** is used when you want to observe some data. This is done simply by creating a data model for our data and attaching our observers to the model. The data model serves as a relay to notify all observers that the data has changed. This eliminates the need for our observer to constantly pool for updates.

Decorator patterns used to obtain different instances of the same class. For instance say we have two instances of a class called goat. Goat on its own have properties that could sufficient for most of its common applications such as climbing but say we want to decorate our goat to create a goat with the “pretty” property. To do this we add a tuxedo to our goat but do note that even with the tuxedo it is still a goat. A more real example would be containers as they have to be decorated in order to be useful.

The focus on the design is to create a flexible yet simplistic filter interface. An example of this is the filter GUI where the user may pass any JComponent as a GUI but doesn’t need to know how to update the image. Another example of simplicity is how filters are added to the program. The user doesn’t need knowledge about prototyping as this is resolved at runtime. Only requirement is that the user implements the filter interface or extend the AbstractFilter

```
1  package filters;
2
3  import javax.swing.JComponent;
6
7  public abstract class AbstractFilter implements Filter {
8      public JComponent getGUI() {
9          return null;
10     }
11
12     @Override
13     public String toString() {
14         return this.getClass().getSimpleName();
15     }
16 }
```

Figure 4 A code snippet of the AbstractFilter class

```
1  package framework;
2
3  import javax.swing.ImageIcon;
5
6  public interface Filter {
7      public ImageIcon filter(ImageIcon image);
8      public JComponent getGUI();
9  }
10
```

Figure 5 A code snippet of the Filter interface

class and place it in the filter package and the ClassLoader will handle the rest.

By default the AbstractFilter class shown figure 4 should be extended instead of implementing the Filter interface shown in figure 5. Doing this makes the GUI component default to null which the system in turn interprets as a blank GUI. It also sets the default name of the filter to be the class name. A more adept programmer could theoretically pass a secondary program as the GUI however this is impractical do to the perhaps over simplistic approach in this project for instance there’s no way for the user to update the image manually.

A ClassLoader isn’t anything that had to be implemented however in terms of usefulness it repaid itself in full already at development phase. In the end product it is just as useful in saving time and reducing the knowledge required to implement your own filters. To keep users from sabotaging themselves safety checks are implemented ensuring that only .class files are read and only classes that can be cast into filters are added to the menu bar.

Testing

To be able to test and successfully find a working system with as few flaws as possible is both difficult and time consuming. To automate a test that would check the methods of a GUI has proved to be difficult without human feedback. This is due to the fact that there's no way to know what a user wants the application to look like. Most of the GUI methods used are tested manually and therefore we can't be sure that no error occurs during runtime. We can see if the called methods doesn't show the result expected. Normally a GUI glitch is quite visible so while crude the testing has eliminated an acceptable amount of unwanted features.

```
void testNewImage() {
    fc.blankImage(50, 50);
    BufferedImage img = (BufferedImage) fc.image.getImage();
    assertEquals(img.getWidth(),fc.image.getIconWidth());
    assertEquals(img.getHeight(),fc.image.getIconHeight());
    for (int x = 0; x < img.getWidth(); x++) {
        for (int y = 0; y < img.getHeight(); y++) {
            assertEquals(-1,img.getRGB(x,y));
        }
    }
}
void testgetFilters() {
    /**
     * Add objects and let classloader load them
     */
    assertEquals("BlackCircle",fc.getFilters()[0].toString());
    assertEquals("Chess", fc.getFilters()[1].toString());
}


void testdeepCopy() {
    /**
     * Testing colorspace
     * Resolution
     */
    BufferedImage newBI = new BufferedImage(100, 100, BufferedImage.TYPE_INT_ARGB);
    for (int x = 0; x < 100; x++)
        for (int y = 0; y < 100; y++) {
            newBI.setRGB(x, y, (int)(Math.random()*0xFFFFFF));
        }
    fc.image.setImage(newBI);
    BufferedImage dc = (BufferedImage)fc.deepCopy(fc.image).getImage();
    assertNotSame(newBI, dc);

    for (int x = 0; x < dc.getWidth(); x++) {
        for (int y = 0; y < dc.getHeight(); y++) {
            assertEquals(newBI.getRGB(x, y), dc.getRGB(x, y));
        }
    }
    assertEquals(newBI.getHeight(), dc.getHeight());
    assertEquals(newBI.getWidth(), dc.getWidth());
}
```

Figure 6 A code snippet of the classes the are tested in the testing section

The automation of a JUnit could be used in the separate methods that doesn't return any event or GUI related object. Therefore the few of the functions that could be tested is newImage, getFilters and deepCopy shown in figure 6. But to test these functions they have to be changed to public so for this security was sacrificed to allow testing. Note testing serves as evidence that you code works which is more important than protecting you software.

Problems

A few interesting and well used methods could be the lambda expressions (for event changes), key events, arrays for JMenu and the AbstractFilter. By combining these we minimized code repeating. With the help of lambda the interface for the action listener could be shortened as shown in figure 7.

```
JMenuItem newImage = new JMenuItem("New");
newImage.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N, ActionEvent.CTRL_MASK));
newImage.addActionListener(event -> newPhoto());
```

Figure 7 A example of lambda expressions used in the code

Instead of repeating the code for the Menu Items they could be included into an array and then loop them into the different sections without needing to add them separately each time. By doing this the code becomes a lot more readable and simple to understand. This is illustrated in figure 8.

```
ArrayList<JMenuItem[]> menuItems = new ArrayList<>();
menuItems.add(new JMenuItem[] { newImage, open, save, exit });
menuItems.add(new JMenuItem[] { filters, refresh, undo });
menuItems.add(new JMenuItem[] { help, about });
JMenu[] menus = { new JMenu("File"), new JMenu("Filters"), new JMenu("Help") };
menuBar = new JMenuBar();
IntStream.range(0, menus.length).forEach(i -> {
    Arrays.stream(menuItems.get(i)).forEach(item -> {
        menus[i].add(item);
    });
    menuBar.add(menus[i]);
});
```

Figure 8 A compact way to assemble a menu bar

Another problem that were found during development is the need of shortcuts. Each time we wanted to test something we would have to take extra time navigating the menu and going through each step manually. Therefore shortcuts were introduced as a way for us and our users to simply access the standard functions without having to manually search for it in the menu lists.

One feature that we wanted to add into the project is the possibility to paint onto the image itself. Unfortunately this was not possible as the filter GUI, filter list and image had no way to actively communicate with each other. A solution for this problem would be to create a data model and make them communicate with the filter GUI and filter list. Optionally the filter GUI could be passed an instance of the frame for reference. This was not thought of at the beginning difficult to implement in hindsight without redesigning core features of the code.

Result

Most of the program is working with the functionality as intended. It’s easy to add additional filters with extra functionality if so desired. The focus on user friendliness restricted the functionality of the filter GUI for better or worse but overall the goals were fulfilled. Drawing an object oriented UML map over the program would have revealed unwanted properties at early build stage which will be a lesson for future projects. Implementing new filters or filter GUI doesn't take more time than writing the code and moving the filter to the correct directory. Depending on what modifications a developer would apply the result could vary but mostly there won’t be any problem to change the source code.