Emotican

李汉青 孙克斌 杨天宇 郑钿彬 周辉

1. Information Hiding

1.1. Modulation

To realize information hiding, one way is to program the interface. Because programming to the interface will hide the details of the operations and the attributes of the class. So we create a ***Photo*** interface and class ***PhotoImpl*** will implement this interface.

```
public interface Photo {
    public Feature getFeature();
}
public class PhotoImpl implements Photo{
    private Feature feature;
    public Feature getFeature(){return this.feature; }
}
```

The ***photoImpl*** is a class which includes information that is essentially to represent a photo. Then we create a Class ***Feature***, it is used to grab the feature of the photo.

```
public class Feature {
    public Feature getFeature(Photo photo){ return photo.getFeature();}
}
```

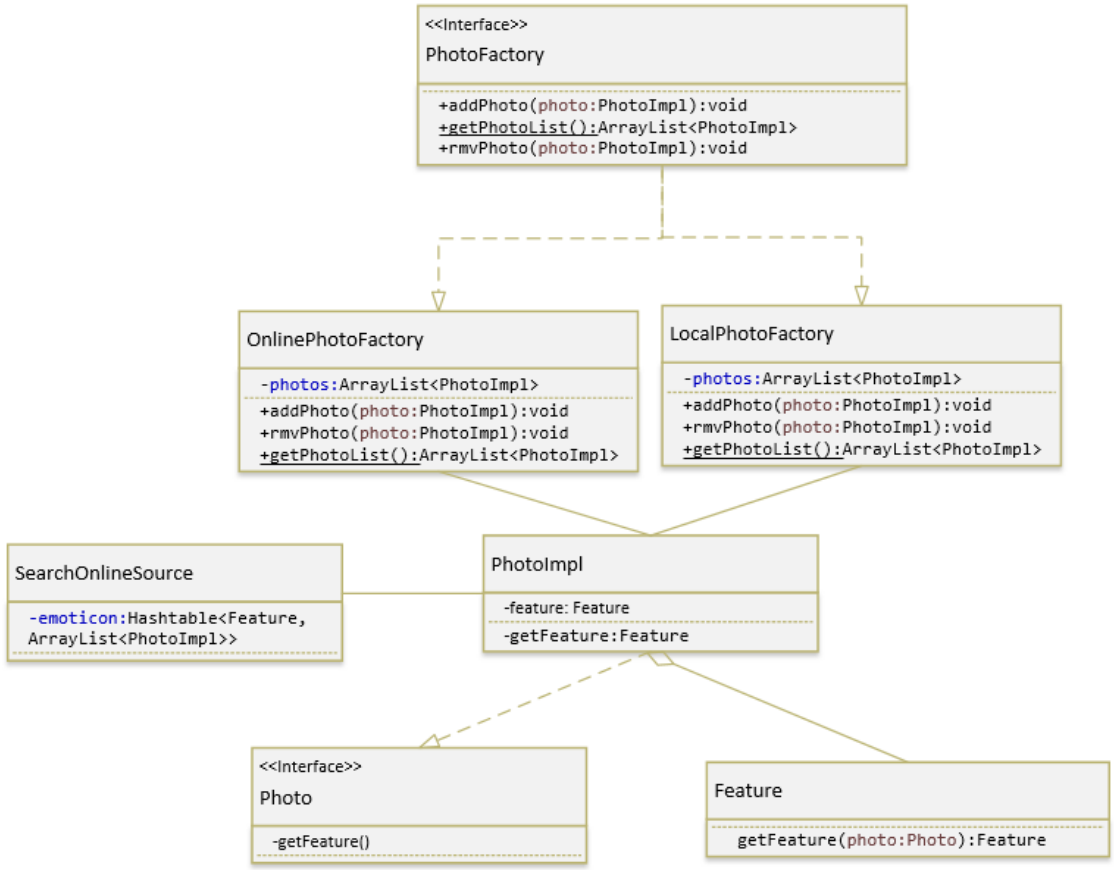
Considering the information hiding, we only pass the interface into the ***getFeature*** function. Though we cannot instance an interface, we can instance a ***Feature*** object, by using the ***getFeature*** function in the ***Feature*** class we are able to invoke the ***getfeature*** function in the ***photoImpl*** to get the feature of a photo, thus the information of the ***photoImpl*** is protected. Another way is to claim the instances to be private and use the get method to get its value. It is common because all of the photos of the client have their own emoticons and they normally do not want to be known by others.

1.2. Robust to Changes

If the new features come in. One possible features are the new photo, we can use ***PhotoFatory*** interface to do the add and remove operations. PhotoFactory is an interface of ***LocalPhotoFactory*** and ***OnlinePhotoFactor*** class. Another possible feature is the feature of a photo. If a new feature come in, the class ***Feature***, ***PhotoImpl*** and the interface ***Photo*** class need to be changed but do not affect other class. Relatively it is stable. But a good thing is that the information hiding will not be influenced.

```
public interface PhotoFactory {
    ArrayList<PhotoImpl> getPhotoList();
    void addPhoto(PhotoImpl photo);
    void rmvPhoto(PhotoImpl photo);
```

```
}
public class LocalPhotoFactory implements PhotoFactory{
    private ArrayList<PhotoImpl> photos;
    public void addPhoto(PhotoImpl photo){ photos.add(photo); }
    public void rmvPhoto(PhotoImpl photo){ photos.remove(photo); }
    public ArrayList<PhotoImpl> getPhotoList(){return this.photos; }
}
public class OnlinePhotoFactory implements PhotoFactory{
    private ArrayList<PhotoImpl> photos;
    public void addPhoto(PhotoImpl photo){photos.add(photo);}
    public void rmvPhoto(PhotoImpl photo){ photos.remove(photo); }
    public ArrayList<PhotoImpl> getPhotoList(){return this.photos; }
}
```



2. Design Patterns

2.1. Factory Pattern

2.1.1. Problems Faced

In our apparent design, the most significant function of our app is the picture editing. The editing needs a lot of different kinds of elements to be added up according to our assumption of making emoticons. Thus, we need a way to add

them into the editor. Driven by this requirement, we decide to adapt factory pattern to this part of project.

2.1.2. Analysis

As what are mentioned above, the factory pattern used here aims to provide a centralized way for adding new elements into our image editor. Our project provides an online library for users to choose the elements from. So, the elements here are images stored either in local or online library. The app will ask the users to choose get elements from local or online resources. Thus, for the consistency in our image editor, the factory should give the chosen image of user no matter where its location.

Driven by this intention, we implement a factory interface, and all the factories need to implement this interface. At the beginning, we decide to implement two factories: LocalFactory and OnlineFactory. The LocalFactory is used to get elements from the local library, and returns the elements selected by user. The online library works pretty similar, it gets the element from online library instead of local one, so it needs to be embedded in a communication module.

Both of the factories will return images to the image editor instance. The image editor keeps the instantiated instance of these two factories, but it doesn't care what they actually do when fetching images. The editor just let the specific factory to fetch the image for it when user requires adding images and points out what kind of library he want to visit. Since all the factories implements the factory interface, they all keeps the methods defined in interface. The editor can always call the methods in the interface to manage the factory instance no matter what the factory class is implemented as long as no errors occurred. By this design pattern, the implement of the factory class is also robust. No matter what the factory class do when it trying to connect the local database or online library, it doesn't affect the image editor class.

2.1.3. Advantages and Disadvantages

The use of factory pattern can lead the generating of our new image object much easier to do in the image editor class. The image editor class has already piled with image processing functions. Adding new image objects is necessary but not an intended work of image editor. An image factory solved this problem. On the other hand, the image objects can be stored on either local database or online server. The image factory class also holds the function of visit these two storages, which make the codes organized.

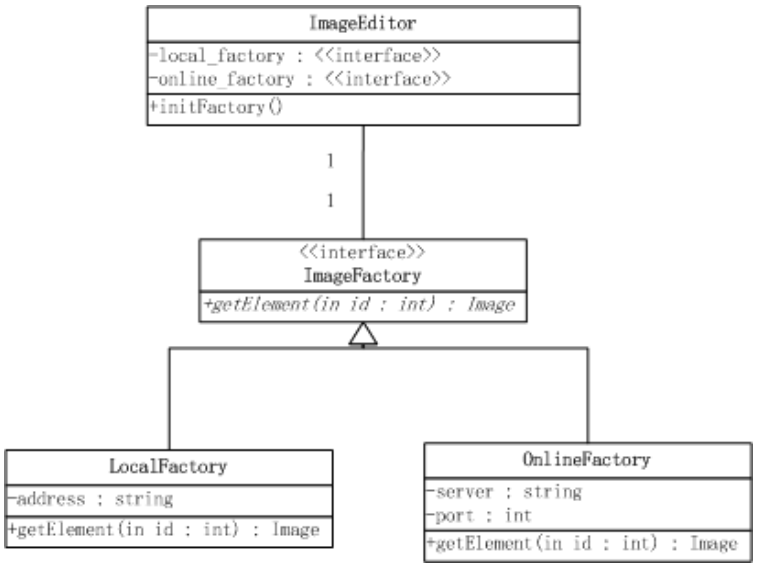
2.1.4. Code Example

```
public interface ImageFactory {
    Image getElement(int id);
}
public class LocalFactory implements ImageFactory {
    @Override
    public Image getElement(int id) {
        return null;
    }
```

```
}
public class OnlineFactory implements ImageFactory {
    @Override
    public Image getElement(int id) {
        return null;
    }
}
public class ImageEditor {
    ImageFactory LocalFactory;
    ImageFactory OnlineFactory;

    private void initFactory(){
        LocalFactory = new LocalFactory();
        OnlineFactory = new OnlineFactory();
    }
}
```

2.1.5. Class Diagram



2.2. Observer Pattern

2.2.1. Problems Faced

Observer pattern is widely used when a publish and subscribe mode exists. In our image editor class, it keeps a lot of instance of different objects. These objects are different images containing some other attributes. There should have a function for our image editor to adjust some properties, like brightness, saturation etc. Under this situation, once the user select the button or slider to adjust the global property, we need to change the attributes of these elements. An observation pattern is pretty useful here.

2.2.2. Analysis

The image editor packs lot of functions. The main processes of building a new emoticon can be roughly divided into two processes. The first is the addition and

recombination of kinds of elements, which can be handled by the factory pattern implementation we described above. The second process is optimization of the image, such as adjust the saturation, brightness etc. This kind of operations need to adjust the attributes of some or all the elements. An observer pattern is used here to decide which components needs to be adjusted when an adjust requirement is adopted by user. For the convenience of management, we build a class named ObserverList. This class holds the all the observer that listens for a change and implements Java ChangeListener interface. The class provides an addListener method to add listeners to the observer object, as well as the delListener method to remove. For each component in the image editor class, it will instantiate a correspond ObserverList object, and add it as the listener to that component to listen the changes. Once the changes happen, the ObserverList object will inform all the registered listeners. By this way, we solve the problem of letting different groups of elements to listen for different changes, and can also be easily managed.

2.2.3. Advantages and Disadvantages

The advantages for implementing observer pattern here is distinct. To achieve the function we intended to, we need to inform different groups of objects for different kinds of changes. In this way, all these elements need to know when the changes happen and what the changes are. An observer pattern design can solve this problem without occupying oceans of resources. If we let those elements to consecutively examine whether there is a concerned change happens, it will take a lot of unnecessary CPU time. That is completely a loss. The observer pattern, on the other side, takes just a little of time.

The disadvantages for implementing observer pattern may lies in making the project little bit more difficult to manage. The ObserverList object needs to be instantiated, and the object needs to get themselves registered when they are created. Whether the expected ObserverList is in exist when the image object is instantiated is a problem needs to be notified.

2.2.4. Code Examples

```
public class ImageEditor {
    ArrayList<ObserverList> list = new ArrayList<ObserverList>();
}
public class ObserverList implements ChangeListener{
    ArrayList<inform> list;

    public ObserverList(){
        list = new ArrayList<inform>();
    }

    public void register(inform listener){
        list.add(listener);
    }
```

```
        public void delete(inform listener){
            list.remove(listener);
        }

        @Override
        public void stateChanged(ChangeEvent arg0) {
            for(inform i:list){
                i.inform(arg0);
            }
        }
    }
    public class ListenerSample implements inform{
        @Override
        public void inform(ChangeEvent e) {
            System.out.println("Received");
        }
    }
    public interface inform {
        public void inform(ChangeEvent e);
    }
```

2.2.5. Class Diagram