Group 013 Utalk Zhijun Wang 1005701037

Adapter Design Pattern
This pattern aimed to make incompatible classes work together by wrapping existing classes to a new interface (the code example wrap Line and Rectangular class together to a Shape interface) and matching old component to a new system (the code example alter the input of Rectangular class to the bottom left's and top right point's coordinates and use RectangleAdapter to convert). The main idea of this pattern is to keep the common feature of two incompatible classes, make two originally independent classes be the subclass of an interface. Doing this can assist the programmer with the easier implementation of methods simultaneously, as the public interface keeps all common features and this feature can be converted by the adapter in order to exhibit all original properties before using an adapter. This technique is effective and applicable, though it seemed to be tedious at the beginning, it still works well for a few methods cases and clearly shows the pattern of coding. In the coding example, AdpterDemo is really effective after using adapter and Shape interface, as we didn't have to cast variables in order to show the desired results, adapters (LineAdapter and RectangleAdapter) have done all conversion for us.

Some code explanations will be shown as following

```java
class RectangleAdapter implements Shape {
    private Rectangle adaptee;

    public RectangleAdapter(Rectangle rectangle) {
        this.adaptee = rectangle;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int x = Math.min(x1, x2);
        int y = Math.min(y1, y2);
        int width = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);
        adaptee.draw(x, y, width, height);
    }
}
```

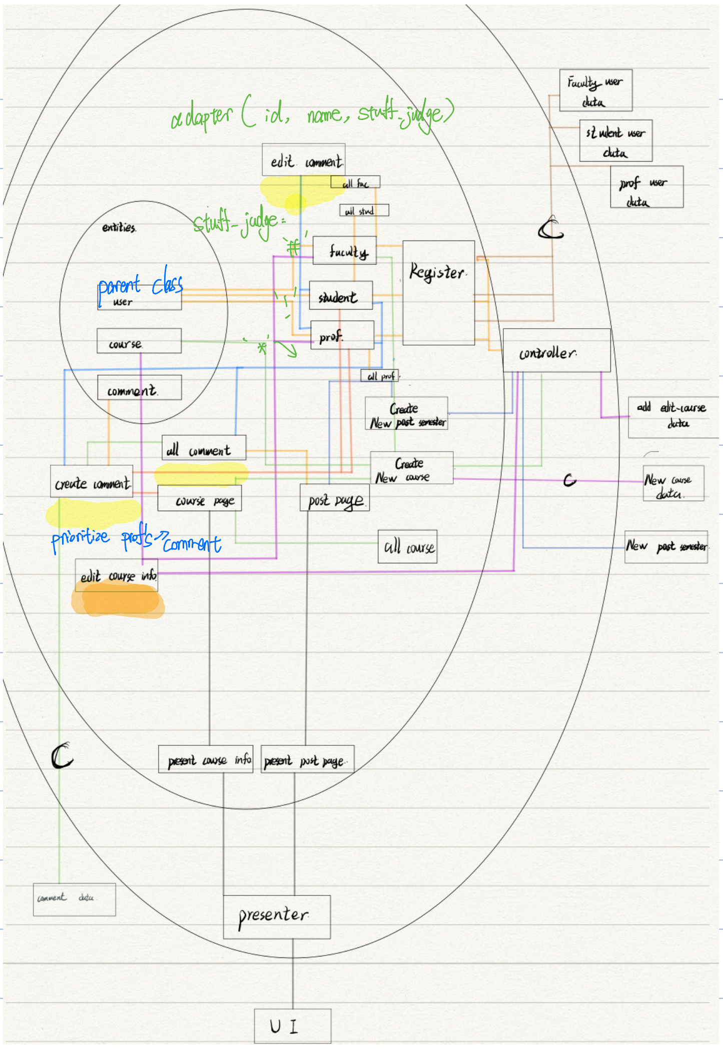
Part I Code

As Shape takes in coordinate of two point, to keep property of rectangle
(bottom left and top right point in this case)
Rectangle Adapter uses input to calculate information of width, height
and bottom left point. (Convert to input of before vision)
This adapter ensures that client can input similar information but
output different class objects

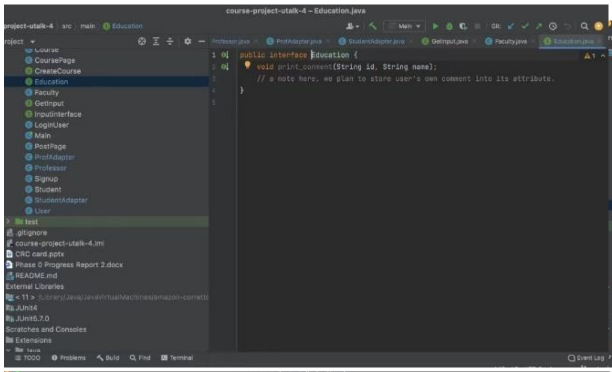
Part 2.

Adapter Design Pattern in Group Project Utalk

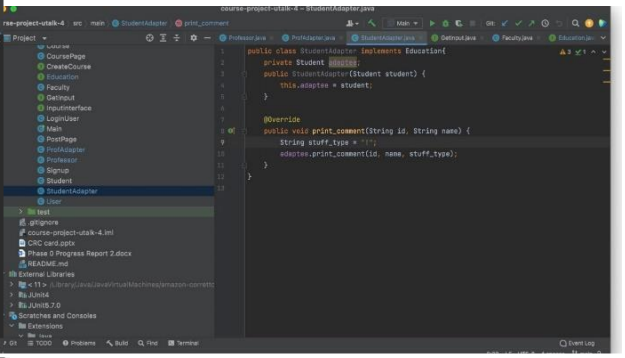
After reconsidering the responsibilities of the user's subclass, I found that the Student class and Professor class share many identical methods, including Course_enrolled, Create_comments, Delete_Own_Comment, Edit_Own_Comment. In order to distinguish one from the other, I suggest that we can add more features to Professor class including automatically adding star *(can be other distinct marks) in Professors' id, adding # for Faculty account. This feature indicates that this user can edit the course page, which is created by faculty. For the student's class, the adapter changes the original information id, account name into format: id, account name, stuff_type = '!' by checking if the student's id contains '!'. Similarly, for the professor's class, the adapter changes the original information id, account name into format: id, account name, stuff_type= '*'. For the faculty's class, the adapter changes the original information id, account name into format: id, account name, stuff_type = '#'. As I said before, this can avoid casting if we use an adapter when implementing most of the methods. Take the Create_comments method for example. We want to prioritize the professor's comment, to do this, we can directly look at the value of stuff_type, if stuff_type == '*', we know it's a professor class and we need to put his or her comments at the front, which is convenient for our implementation. This is a quick technique to avoid type checking and merging similar methods together by using the interface Education.



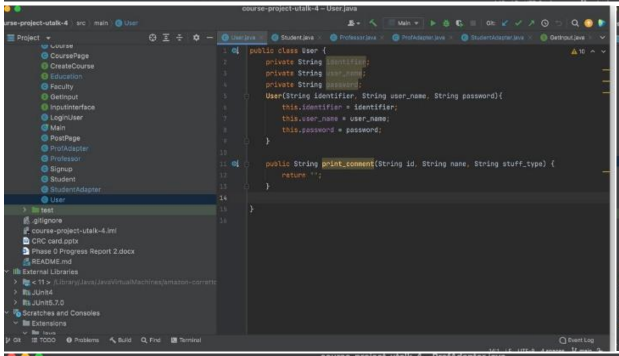
Adapter in our project



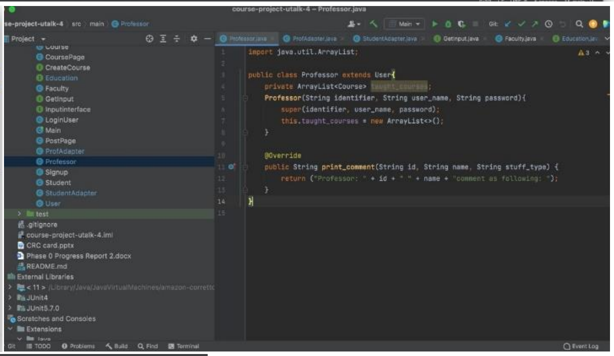
```java
public interface Education {
    void print_comment(String id, String name);
    // a note here, we plan to store user's own comment into its attribute.
}
```



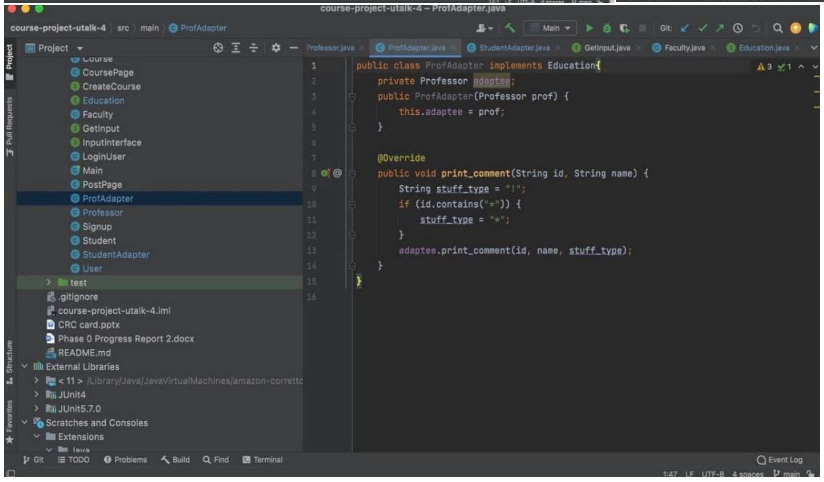
```java
public class StudentAdapter implements Education{
    private Student adaptee;
    public StudentAdapter(Student student) {
        this.adaptee = student;
    }

    @Override
    public void print_comment(String id, String name) {
        String stuff_type = "!";
        adaptee.print_comment(id, name, stuff_type);
    }
}
```



```java
public class User {
    private String identifier;
    private String user_name;
    private String password;
    User(String identifier, String user_name, String password){
        this.identifier = identifier;
        this.user_name = user_name;
        this.password = password;
    }

    public String print_comment(String id, String name, String stuff_type) {
        return "";
    }

}
```



```java
import java.util.ArrayList;

public class Professor extends User{
    private ArrayList<Course> taught_courses;
    Professor(String identifier, String user_name, String password){
        super(identifier, user_name, password);
        this.taught_courses = new ArrayList<>();
    }

    @Override
    public String print_comment(String id, String name, String stuff_type) {
        return ("Professor: " + id + " " + name + "comment as following: ");
    }
}
```



```java
public class ProfAdapter implements Education{
    private Professor adaptee;
    public ProfAdapter(Professor prof) {
        this.adaptee = prof;
    }

    @Override
    public void print_comment(String id, String name) {
        String stuff_type = "!";
        if (id.contains("*")) {
            stuff_type = "*";
        }
        adaptee.print_comment(id, name, stuff_type);
    }
}
```

I have implemented a new interface Education that contains common methods of Student and Professor class. (Take print_comment for example. Finally, we add more methods afterward) The adapter allows me to convert the input format (String id, String name) into a new one (String id, String name, String stuff_type). This can clearly show the type of User. Second thought of the implementation, I would suggest leaving the Education interface as a collection of unimplemented methods that are shared by Student and Professor because the Faculty class has many methods that are different from the Student but similar to Professor such as editing course page.