Project 4 - Design Patterns

Submit Assignment

Due Dec 6 by 11:59pm **Points** 40 **Submitting** a file upload

Due: Sunday, December 6
Points: 40 points
Deliverables: A *.zip file of the ducksim folder containing all of your* .java files

In addition to the requirements given below, please implement the following two requirements.

1. *When you select a duck (left click on it), the background of the square should turn to the duck's color (instead of black)*
2. *When a duck joins the welcoming committee, a little "w" should appear in the bottom left corner of the duck's square.*

You can see a sample run of a completed application in **this video (https://drive.google.com/file/d/1AfyWnq7DaUOdmMGM6EVRZgTBdtLQyoIW/view?usp=sharing)** .

Download the **ducksim-starter-src** archive, create a project from it, and make the following modifications.

NOTE: I am considering adding another pattern to this project (composite). I should know more within a week.

Strategy Pattern

Use the Strategy Pattern to encapsulate flying behavior and quacking behavior. Create FlyBehavior and QuackBehavior interfaces. FlyBehavior should have 2 implementations: FlyNoWay and FlyWithWings. QuackBehavior should have 3 implementations: QuackNoWay, QuackNormal, and QuackSqueek. Implement the following capture and release behavior for ducks. When a duck is captured, it will neither fly nor quack. However, when a duck is released, it will exhibit the flying and quacking behavior that it did originally.

Decoy Duck

Implement a decoy duck that can neither fly nor quack. The color of the decoy duck should be orange.

Decorator Pattern

Add an abstract decorator class Bling to ducks. You should have 3 types of bling represented by the following classes: StarBling, MoonBling, and CrossBling. Each item of bling will be added to the display method by adding a colon and a `*` for star, a `)` for moon, or a `+` for cross. For example, the display method of a mallard duck with two stars and a moon will return the string `Mallard:*:*:)`. The DuckSimView component is already configured to handle the bling.

Modify the MakeDuckDialog component so that it includes a bling panel in between the duck panel and the button panel. The bling panel should have a grid layout consisting of 3 rows and four columns. The first row will have a label with `Star`, a label with `0`, a button with `+` and a button with `-`. The second row will be similar but its first label will be `Moon`, and the first label of the third row will be `Cross`. The buttons should have the following logic:

When a `+` button is pressed the number in its row is incremented. When a `-` button is pressed, the number is decremented. The numbers will never go below 0, and the sum of all three numbers will never go above 3.

Use lambda expressions for the action listeners on your buttons. See how I did the action listeners for the other buttons. For example:

```
cancelButton.addActionListener(e -> {
    this.dispose();
});
```

The addActionListener is expecting something that implements the ActionListener functional interface, which only declares one method (actionPerformed). As of Java 8, instead of writing a class that implements only one method, you can use lambda notation: just pass in a variable that represents the arguments to the method (in this case, e represent the ActionEvent argument to the function actionPerformed) and then pass in the method body. Done!

Factory Pattern

Add class DuckFactory to the simulator. The duck factory should handle the creation of all ducks. It will have a method called createDuck with the following signature:

```
public Duck createDuck(String duckType, int starCount, int moonCount, int crossCount)
```

Make the duck factory a singleton class!

Adapter Pattern

Write an adapter called GooseDuck for the following Goose class:

```
class Goose {
    String getHonk() { return "Honk!"; }
```

```
    String getName() { return "Goose"; }
}
```

The adapted goose should use the normal fly and quack strategies. However, the result of getHonk should appear when a goose “quacks” and the result of getName should appear on the goose’s button.

Observer Pattern

Update: Do NOT use the *Java* Observable class and Observer interface to implement the observer pattern!

Instead, create your own Subject and Observer interfaces, and use those to implement the observer pattern as show in HFDP. It is also fine to make Subject an abstract class, and implement methods registerObserver, removeObserver, and notifyObservers directly in the Subject class. However, Observer must be an interface, and it should have the update method.

The duck factory will be the subject and the ducks will be the potential observers. When a duck joins the DuckSim Welcoming Committee (DSWC) it registers with the duck factory so that when a new duck is created a `Welcome` message appears above the DSWC member’s name.

Modify the simulator so that if a duck is captured, it says `Beware!` instead of `Welcome!`.

The Welcome/Beware message should stay over the duck’s name until the screen is repainted.

Example Package Explorer

Your package explorer tab on Eclipse should look pretty similar to the image below when you are finished.


DuckSim_Solution
src
ducksim
Bling.java
CrossBling.java
DecoyDuck.java
Duck.java
DuckFactory.java
DuckSimController.java
DuckSimMain.java
DuckSimModel.java
DuckSimView.java
FlyBehavior.java
FlyNoWay.java
FlyWithWings.java
Goose.java
GooseDuck.java
MakeDuckDialog.java
MallardDuck.java
MoonBling.java
QuackBehavior.java
QuackNormal.java
QuackNoWay.java
QuackSqueek.java
RedheadDuck.java
RubberDuck.java
StarBling.java
State.java
JRE System Library [Java SE 8 [1.8.0_73]]
DuckSim_Starter