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.*

Here is a sample run of a completed application:

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

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

NOTE: 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.

Composite Pattern

In this section, you will use the composite pattern to create a Flock class, which is the a composite for the Duck class. The requirements for the composite class are:

- The Flock must be a Duck
- The Flock will hold a collection of Ducks
- The Flock must display the word "Flock" followed by the first letter in the name of every Duck it holds (where the Bling would normally be)
- A Flock cannot have Bling
- The fly method works as usual on a Flock (it uses the Duck default)

- The quack method works as usual on a Flock, but it should the string "Noise!" instead of "Quack!"
- The joinDSCW method works as usual on a Flock, and every Duck that a Flock holds will also join the DSCW. Similarly with quitting the DSCW.
- The capture method works as usual on a Flock, and every Duck that a Flock holds will be captured. Similarly with releasing the Flock.

You will need to make some changes to the model and the controller in order to effectively use this class. First add a getSelectedDucks method to DuckSimModel:

```
public List<Duck> getSelectedDucks() {
    List<Duck> result = new ArrayList<>();
    for (Integer i : selected) {
        result.add(ducks.get(i));
    }
    return result;
}
```

Next, change the code in DuckSimContoller that executes when the New-Duck button is pressed:

```
if (view.clickedNewDuckButton(e)) {
    new NewDuckController(model, view).createNewDuck();
}
```

Finally, add a new class called NewDuckController that controls what happens when you click the "New Duck" button. The class should have fields "model" and "view" to hold the model and the view, and a single method called createNewDuck. The createNewDuck method should create a NewDuckDialog when no ducks are selected (as was done originally), but when one or more ducks are selected, it should create a new Flock and add it to the model.

```
public void createNewDuck() {

    if (model.noSelectedDucks()) {
        MakeDuckDialog makeDuckDialog = new MakeDuckDialog(model, view);
        makeDuckDialog.setSize(300, 200);
        makeDuckDialog.setVisible(true);
    } else {
        // get the selected ducks from the model
        // filter the selected ducks by removing any flocks
        // if there is more than one duck after removing flocks,
        // create a new flock with the selected ducks

        Flock flock = new Flock(ducks);
        model.addNewDuck(flock);
        view.repaint();

    }
}
```

Essentially, the New-Duck button is used to create both Ducks and Flocks (composites), depending on whether any ducks are selected.