Rajat Bhageria
CIS 350
Prof. Chris Murphy

HW 4 Writeup

Factory Method Design Pattern:

I have implemented the Factory Method Design Pattern in the following classes part of the datamanagement packages: `FileTypeReader, JSONReader, ReaderFactory`, and `TextReader`.

In particular, `FileTypeReader` is an interface with a single method definition, read(). Both `TextReader` and `JSONReader` implement `FileTypeReader` and have a `read()` method that reads in either a JSON or a TEXT file.

The ReaderFactory is the actual factory and contains an if-else statement that examines whether the `typeOfFile` entered in as an args argument was “TEXT” or “JSON.” Then, based on which it was, it returns a `FileTypeReader` of type `JSONReader` (if the argument was “JSON”) and a `FileTypeReader` of type `TextReader` (if the argument was “TEXT”).

Then, in the `Main` class around line 46, I create an instance of the `FileTypeReader` by calling `FileTypeReader reader = ReaderFactory.getReader()` (a static method). Then, I set an arrayList equal to the return value of the statement `reader.read()`. I do the same for the

Observer Design Pattern:

I have implemented the Observer Design Pattern in the following packages part of the logging package: `Logger, Subject, ScreenLogger`, and `FileLogger`.

The Logger is our Observer and is an abstract class with a single method definition, info (equivalent to `update()`).

Both the FileLogger and the `ScreenLogger` extend the `Logger` and implement the info method.

The Subject class is the Subject. This class has a private state field which stores the last statement logged. I can change the state by passing in a string via the setState() method. This class also has an add() method that allows me to associate a particular Observer instance with this subject (it stores these in an ArrayList). If `notifyAllObservers()` is called, all the observer's (in our case the `ScreenLogger` and the `FileLogger`) `info()` method is called and the String with the state in it is printed/logged.

In the main, I have created a global `Subject` instance around line 20 using "private static." From lines 38-39 in main, I create an instance of `FileLogger` using the singleton `FileLogger fileLogger= FileLogger.getInstance()`, and an instance of `ScreenLogger` using the singleton `ScreenLogger screenLogger = ScreenLogger.getInstance()`. Then, I simply call `subject.add(fileLogger)` and `subject.add(screenLogger)` in order to associate my `fileLogger` and my `screenLogger` with the subject.

Now, all I have to do to print/log anywhere in the program is call `Main.subject.info(message)`. This will print/log both on the screen using println and on the log file.

Strategy Design Pattern:

I implemented the Strategy Design Pattern mainly in the controller package in the following classes: `Context, Strategy, CoursesAboveQualityRating, CoursesByInstructor, LowestDifficultyRatio`, and `InputReader` (in the UI package)

Strategy is an interface with a single method definition, `getAnswer()`. The `CoursesAboveQualityRating, CoursesByInstructor, and LowestDifficultyRatio` classes all implement Strategy and thus have a `getAnswer()` method that returns an ArrayList of the responses to the various questions.

In the `Context` class, the constructor takes in a `Strategy`, and then there's a single method called `executeStrategy()` that calls the `getAnswer()` method of the `Strategy` that was passed into the parameter.

In the UI package in class `InputReader`, I instantiate instances of 3 different contexts, and then pass in one of the three strategies into each one; this happens at line 32, 50, and 59 in the `InputReader` class. I then set an `ArrayList<String> array = context.executeStrategy()`, and then print out the ArrayList.

Because of this setup, to add a new question to ask, all the programmer has to do is create a class in the controller class that implements the `Strategy` class and has a `getAnswer()` method that returns an `ArrayList` of the responses to print. Then, to actually display the answer, all the programmer has to do is call `Context context = new Context (new newClass())` and then call
`ArrayList<String> array = context.executeStrategy()`

Compare to HW 1:

This iteration of this assignment is better set up in many different aspects than HW 1, but one of the major aspects is its modularity. This homework is incredibly modular and easy to change.

For example, if I want to add a new type of Logger, all I have to do is create a the new Logger class, make sure it extends Logger (in the logging package), and then whenever I set up the Subject, I just have to add this logger to the subject. Then, just as I call the other Observers using Subject.setState(), this new Observer will also launch it's info() method. The Observer method makes the program a lot more modular.

The homework is also very modular in how the programmer can add new types of Files that can be read fairly easily. All the programmer has to do is make sure that the new class implements the FileTypeReader interface, and then ask the user what kind of file it is (just like we asked the user this time around whether it's a "TEXT" or a "JSON" file). Now, we can just go to ReaderFactory, and add an if clause to figure out which kind of File Type was entered. The Factory Method design pattern makes the program a lot more modular.

The program is also very modular in that it easily allows the programmer to add new questions to ask fairly easily. All the programmer has to do is create a class that implements the Strategy interface. Then, whenever trying to get the answer, all the user has to do is call `Context context = new Context (new newClass())` and then call `ArrayList<String> array = context.executeStrategy().` Thus, the Strategy design pattern makes the program a lot more modular than the original HW1.

All these cases would have been a lot more difficult to implement in HW1. If you look at my original submission, there were only two classes and modularizing/changing/adding new types would be quite hard to do. But now, there are over 15 classes that are well labeled and so adding new types and changing things are a lot easier. The addition of interfaces and abstract classes also make this version a lot easier to modularize and create realizations/extensions of concrete classes.

Please note: In the command line, don't enter the classpath (only enter "ratings.txt" or "ratings.json"). I have already added the .txt and .json files to the correct classpath in the program itself (src/edu/upenn/cis350/hwk4/)