Design Patterns Exercise

In lecture we talked about a number of design patterns. In this exercise you apply them to some small Java examples.

Part 1: Questions: Decide which Pattern

Before we implement some patterns, let's first ensure we know when to use each pattern.

What pattern best meets each of these situations below?

1. The model stores a collection of blocks. Blocks can be medal or wood; can be painted, sanded, or chrome plated; and can sometimes be radioactive or magnetized. What design pattern would allow the system to easily add new types of blocks without changing existing code?
   - Explain how two design *principles* apply to this design.
2. The model for a game stores robot. The robot navigates a maze that has obstacles. While playing the game, the robot can be upgraded with new parts that change its abilities like speed, weapons, and shields. Which design principle allows the robot object to change its behaviour at runtime in flexible ways.
   - Explain how two design *principles* apply to this design.
3. The model stores a phone number, and the UI (a keypad) allows the user to enter digits one at a time. A different part of the UI wants to respond to each key being entered; however a) the two parts of the UI should be decoupled, and b) the model should be decoupled from the UI (model knows nothing about the UI). Which pattern would allow this?
   - Explain how two design *principles* apply to this design.
4. A library supports recursively searching a directory for files. It allows the client code to provide it an object to filter the results. For each file which the library finds, it will ask the filter object if that file should be accepted or rejected. (See Java's FileFilter)
   - Explain how two design *principles* apply to this design.

Part 2: Implementing Patterns

Follow the directions for each section. **You may refactor the provided code** any way you like as long as you implement the expected patterns.

Start with code in this starter code project for IntelliJ.

a) Telephone

The `telephone` package in the starter project contains partial code for making some parts of a (pretend!) telephone.

Change the code so that it uses the Observer Design Pattern as follows:

- In the model, define an interface for the observers.
- Have the model notify the observers whenever a new digit is entered for the phone number.
- Have the UI (the Screen class) create two observers:
  1. The first observer prints the newest digit out to the screen
  2. The second observer prints "Now dialing 12345678901..." out to the screen (where the number is the number the model has).

Sample output

Constraints

- Only the UI can print to the screen

- The model must be decoupled from the UI (model must not know about the UI).

b) Web Search

The `websearch` package in the starter project contains partial code for a pretend web search engine. It will read a data file and "pretend" that each line is a query someone sent to a search engine. Our goal is to extract, on the fly, "interesting" queries meeting certain conditions.

Note that the code already uses the Observer Pattern to notify the `Snooper` of each query. The `Snooper` then prints everything out.

Change the code so that it uses the Strategy Design Pattern as follows:

- In the model, create a new interface which describes the interface for a policy object that will define a query filter.
- A query filter policy object will have one method which will be passed a string (the query) and return true if the model should notify the observer about this query; returns false if the observer is not interested in this string (the query). For inspiration, see the Java FileFilter class.
- Change the model so when an observer is registered, the registration method to also accept a query filter policy object.
- Change the model to, for each query (string from the file), check if an observer is interested in the query before notifying it.
- Change the client (`Snooper.java`) to create two query observers:
  1. One prints out "Oh yes! <query>" whenever the query contains the word 'friend' (case insensitive).
  2. One prints out "So long....! <query>" whenever the query is more than 60 characters long.

Sample output

Constraints

The model should not know anything about the implementation of the query filter policy objects, other than they implement the required interface.

c) Bakery

The `bakery` package in the starter project contains partial code for a bakery. The bakery makes two types of cakes: vanilla and chocolate. They now want to make more complex cakes such as a "Multi-layered Vanilla cake with sprinkles that says 'Hello World!'"

Change the code so that an order can contain such complex cakes using the Decorator Pattern:

- Create the necessary decorator classes:
  - For multi-layered cakes, add $5 and print "Multi-layered" out at the front of the name.
  - For sprinkles, add $2 and print "with sprinkles" at the end of the name.
  - For a cake with the saying *X*, add nothing to the cost, and print "with saying '*X*'" at the end of the name.
- Add the new type of cake: strawberry cake, which costs twice as much as a Cake.
- Change the client to add the following to an `Order`, and print the `Order`:
  - Chocolate cake
  - Vanilla cake with saying "PLAIN!"
  - Vanilla cake with sprinkles with saying "FANCY"
  - Multi-layered Strawberry cake with double sprinkles and two sayings "One of" and "EVERYTHING"
    Suggested output is: Multi-layered Strawberry cake with sprinkles with sprinkles with saying "One of" with saying "EVERYTHING"

Sample output

Constraints

- Adding a new type of cake does not change any existing code (except to instantiate it)
- Adding a new way to decorate or style the cake (such as multi-layer, or with sprinkles) does not change any existing code (except to instantiate it)

Submission

In IntelliJ, export your entire project as a zip file: File > Export > Project to Zip File...

Submit the ZIP file to CourSys. There is nothing that needs to be submitted for the initial "Questions: Decide which pattern".

Marking

- Exercise marked on coming up with a reasonable solution using the indicated patterns.

- Output formatting not important; OOD is important.
- Each code section marked out of 5 points:
    - 5: Great! It's the pattern and works.
    - 4: Reasonable in most places, but missed a part of the pattern, mostly works.
    - 3: A start on the pattern, but did not get more than half way, mostly works.
    - 2: Some changes, but not using the pattern, partially works.
    - 1: Poor changes, not the pattern, does not work.
    - 0: Not submitted
- "Part 1 Questions: Decide which Pattern" worth no marks; nothing submittable for it.