CS 349 Task 6: Decorator and State Patterns

Description

The goal of this task is to build a reasonably functional vending machine for coffee that uses the State pattern to control the Decorator pattern according to the actions of a user (who is not part of the solution). This transaction can have many paths, but the expectation is to insert money, select the coffee type, select the extras, get the product, and receive any change. Lectures 34 and 36 cover this process in detail.

The midterm exam and holiday disrupted the lecture schedule. We have already begun discussing the Decorator pattern, but State will come on Wednesday and probably carry over to Friday. To get you started, Part 1 is independent of Parts 2 and 3. You will submit it as a pretask.

Specifications

Unlike in previous tasks, the specifications in this document provide only an overview. The details are in the Javadoc posted on the task link. Pay attention to detail and implement everything as specified. If you are unsure, ask for clarification as early as possible in the development process. Do not assume anything. Do not deviate from the specifications.

Part 1: Money

This part represents the monetary aspects of a transaction. It primarily applies the Strategy pattern. The Singleton pattern could also play a role with respect to the Flyweight pattern coming soon.

Money

The first aspect represents an amount of money in standard dollars and cents. Primitive floating-point datatypes in Java (and indeed in any programming language) are inappropriate for financial calculations. Oddly enough, Java does not include a suitable library for such a common requirement (although the next version of Java finally does). The general-purpose `BigDecimal` is commonly used, but it provides no cohesive data and control for monetary processing. Your job is to create a better version called `Money`, which is based on an integer representation.

Currency

The second aspect represents the denominations of money that may be processed. For paper currency, there are one, two, five, and ten-dollar denominations; for coin currency, five, ten, and twenty-five cents, as well as a dollar. Your job is to create the `A_Currency*`, `CurrencyPaper*`, and `CurrencyCoin*` classes.

Currency Manager

The third aspect manages monetary transactions. It primary role is to make change from `Money` in terms of `A_Currency`. It also keeps track of indirect profit resulting from inexact change because penny is not a supported currency denomination.

Part 2: Coffee

This part represents the product aspects of a transaction. It applies the Decorator pattern to build a product by successively wrapping layers of ingredients. Each layer maintains a simple text description and cost. At any time, the product can recursively return the total cost and a list of the corresponding descriptions (ordered as specified during assembly).

Coffee

The first aspect represents the required single base ingredient of a product, which is either regular or decaffeinated coffee. The base ingredient cannot be added to itself or any other ingredient.

Extras

The second aspect represents the optional and unlimited extra ingredients of a product, which are cream, sugar, or milk. Each may be added to any base and extra ingredients.

Part 3: Vending Machine

This part represents the control aspects of a transaction. It applies the State pattern to manage a transaction by collecting money, allowing the user to select the ingredients of the product, returning the product and appropriate change or a refund.

The state transitions are as follows, read across:

A	`StateStartTransaction`
B	`StateBuildCoffeeBaseRegular`
C	`StateBuildCoffeeBaseDecaf`
D	`StateBuildCoffeeExtraCream`
E	`StateBuildCoffeeExtraSugar`
F	`StateBuildCoffeeExtraMilk`
G	`StateDeliverProduct`
H	`StateReturnChange`
I	`StateReturnRefund`

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
A		▨	▨						▨
B				▨	▨	▨	▨		▨
C				▨	▨	▨	▨		▨
D				▨	▨	▨	▨		▨
E				▨	▨	▨	▨		▨
F				▨	▨	▨	▨		▨
G								▨	
H	▨								
I	▨								

Your tests from `main()` interact with the current state to produce the next state, and so on. Begin at `StateStartTransaction`. There is no capability to exit the state machine.

When entering `StateDeliverProduct`, print to standard output the following:

```
[MACHINE] delivered
 -> description
```

where *`description`* is the description of each ingredient, each entry on a separate line.

When entering `StateReturnChange`, print the following:

```
[MACHINE] returned
 -> denomination
```

where *`denomination`* is the string representation of the each currency unit returned, each entry on a separate line.

When entering `StateReturnRefund`, print the following:

```
[MACHINE] refunded
 -> denomination
```

where *`denomination`* has the same behavior as in `StateReturnChange`.

Deliverables

Submit all your source files in a zip file. Include a plain-text `readme.txt` that indicates what does not work. If you believe everything works, indicate so.

You must package your code as specified. Include error checking where appropriate. You may add any code to facilitate your design, but you must not deviate from the specifications. It is not necessary to comment your code.

Code that does not compile will not be graded.