CS 2212 B

Introduction to Software Engineering

Project Description

Project 3

Kostas Kontogiannis

1. Introduction

This document presents information on your third deliverable.

For the third deliverable (Project 3), you will provide implementations for the: i) Singleton Design Pattern for the `Controller`, `Façade`, `ItemRepository`, `HighQuantityProxy`, `LowQuantityProxy`, `SupplierProxy` and, `WelcomeProxy`; ii) Strategy Design Pattern for two Individual Pricing Strategies and two Restock Strategies and; iii) Factory Method design pattern for the `IndividualPricingStrategy`, and `RestockStrategy.`

Please refer to the system description provided in Project 1 (Section 2) “Overall System Structure” for a documentation of the system’s packages and classes.

2. Tasks for Project 3

Implement the following:

I1. Make Singletons the following classes `Controller`, `Façade`, `ItemRepository`, `HighQuantityProxy`, `LowQuantityProxy`, `SupplierProxy` and, `WelcomeProxy`. Follow the examples of the `DataManager`, `StockManager`, which have already been implemented as Singletons.

I2. The Strategy Design Pattern. You will implement:

1. Two Individual Pricing Strategies. You can call them `IndividualPricingStrategy1` and `IndividualPricingStrategy2` . The `calculate` method of the `IndividualPricingStrategy1` may just compute the price as the product of `quantity` and `price`, while the `calculate` method of the `IndividualPricingStrategy2` may add or reduce a 10% of the price computed as a product of `quantity` and `price`. You can see as an example the strategies. These are referenced by the `OrderItem` class (in the `utils` package) which has an attribute `pricingStrategy` (see the `private IndividualPricingStrategy pricingStrategy` attribute, and used in the `calculateItemPrice` method of the `OrderItem` class).

2. Two Restock Strategies. You can call them `RestockStrategy1` and `RestockStrategy2`. The `calculateQuantity` method of the `RestockStrategy1` may just return a number (say 50) as the quantity to restock, while the `calculateQuantity` method of the `RestockStrategy2` may indicate that if the item is “apples” then rectock by 100 else restock by 25. These strategies are referenced by the `StockManager` class (in the `viewers` package) which has an attribute `restockStrategy` (see the `private private RestockStrategy restockStrategy` attribute in the `StockManager` and used in the `inform` method of the `StockManager` class). After you compute the restock quantity using the appropriate strategy which is attached to the `StockManager`, make sure you update the `restockDetails` map for this item name accordingly. Make sure you use the `setRestockStrategy(RestockStrategy restockStrategy)` in order to change when needed the `restockStrategy` field value of the `Stockmanager`. The method `setRestockStrategy` is called when in the `driver_file` you encounter the `StrategyChange` line (e.g. `StrategyChange strategy1` meaning that the restock strategy has to be set to `RestockStrategy1` restock strategy. You will use the factory passing it as a parameter the string (say) “`strategy1`” to create the right RestockStrategy object and set it on the `StockManager` using its `setRestockStrategy` method. Make sure if in the driver file you have “`strategy1`” as the strategy name, you use the same string in the factory to determine the object type of the restock strategy to be created.
In order to make use of the `StrategyChange` line on the `driver_file` as mentioned above, you need to uncomment the following section on the `driver.java` file

```
// Uncomment the following lines when restock strategies are implemented

//RestockStrategy strategy = RestockStrategyFactory.create(lineTokens[1]);
//StockManager.getInstance().setRestockStrategy(strategy);
```

So it looks like:

```
if (lineTokens[0].equals("StrategyChange")) {

  RestockStrategy strategy = RestockStrategyFactory.create(lineTokens[1]);
  StockManager.getInstance().setRestockStrategy(strategy);
}
else…..
```

Notes:

a. The `indiv_pricing_strategy_file` contains information on the pricing strategy for a single item in the format <itemName, Strategyname>. The `indiv_pricing_strategy_file` is used by the constructor of the `IndividualPricingStrategyRepo` class (see `IndividualPricingStrategyRepo.java`). For a complete example of how it should work see the constructor of its dual counterpart, namely the `AggregatePricingStrategyRepo` class, which is fully provided to you. Again make sure the strategy name you have on the `indiv_pricing_strategy_file` is the same name that is used in the factory to create the right `InvididualPricingStrategy` (see how it is done for the `AggregatePricingStrategy` and the use of the `aggr_pricing_strategy_file` where the final pricing strategy is set for a buyer).
b. Make sure when you set the the `pricingStrategy` attribute of the `OrderItem` in its contructor using the `itemName` parameter you use the `IndividualPricingStrategyRepo` class. So

your code at this part of the `OrderItem` constructor would look like:

```
this.pricingStrategy = IndividualPricingStrategyRepo.getInstance().getStrategy(itemName);
```

As a complete example you cam also see how the `AggregatePricingStrategyRepo` is used in the constructor of the `Invoice` class.

I3. The rest of the factories. In Project 2 you have implemented a Factory Method Design Pattern for `ItemState`. In Project 3 you will implement the rest two factories that is the Factory Method Design Pattern for the `IndividualPricingStrategy` class and for the `RestockStrategy` class. You can use as an example the factory for AggregatePricingStrategy class which creates any one of the two subclasses `AggregateDefaultPricingStrategy` or the `TestAggregatePricingStrategy`.

3. Deliverables for Project 3

Submit the following:

1. An archive file with your project in a form that can be imported in Eclipse and tested using the input data files you should also include in the archive (i.e. the driver file, buyer_file, etc.).

Submit your deliverables at OWL by Friday April 3, 17:00:00.