CS 2212 B

Introduction to Software Engineering

Project Description

Project 1

Kostas Kontogiannis

1. Introduction

This document discusses a) the structure of the *Warehouse Management System*, b) the use cases to be considered for the first deliverable, and c) the specification and implementation artifacts for the use cases related to the first deliverable.

The *Warehouse Management System* (WMS) is a software system that simulates the operations of the warehouse of a simple company that sells goods to clients, and buys products from suppliers. The model of the warehouse of the company is implemented as a database with one table denoting the ID of each product, the name of the product, its stock quantity, and its unit price. Once a specific quantity of product is sold to a customer, this quantity is taken out of the available stock of this product from the corresponding database entry pertaining to this product. If the stock quantity of a product drops below 10 items or an order cannot be fulfilled the product's state is set as "out of stock" (see packages *ca.uwo.model* and *ca.uwo.model.item.states* for the different states a product can be in). Once a product's state is "out of stock" then a notification is sent so that a purchase (i.e. restock) action can be triggered and the item be restocked from a supplier. Each product is restocked according to a specific *strategy* (e.g. a restocking strategy for a product can be 50 units per restock operation, while for another product a different restocking strategy could be used, leading thus to a different restock quantities to be added). In addition to the restocking strategies, the price of an individual product for a given order can be calculated using different strategies (e.g. orders of apples more than 100 units get a 10% discount for this product), and the total price for the whole order (an order may contain many products with different quantities each) is also calculated using different strategies (e.g. orders of total value more than 1,000 dollars get an additional aggregate 5% discount to any other individual discounts applied for each product in this order).

2. Overall System Structure

The Warehouse Management System is split for modularity into a number of different packages. An outline of each package and its contents is provided below:

- *ca.uwo.banking* package: This package provides the initiation of a payment (to a supplier) and the reception of a payment (from a buyer) after a transaction (buy, or restock) is completed. This package is a placeholder for future system extensions
    - *BankingTransactions.java*: This class implements the operations *receivePayment* and *paySupplier*
- *ca.uwo.client* package: This package implements the buyer and the supplier.
    - *Buyer.java*: This class implements the operations *buy*, and *pay*. The *buy* operation is a client-side process that sends a *placeOrder* invocation to the *WelcomeProxy* of the *Warehouse Management System*. The *pay* operation just prints the generated invoice. The Buyer class has two attributes *userName* and *password* which are used later for authentication purposes using an *authenticate* method in the receiving proxies of the *Warehouse Management System.*
    - *Supplier.java*: This class implements the supply operation which is a client-side process that sends a *restock* invocation to the *WelcomeProxy* of the Warehouse Management System.
    - *Client.java*: This is an abstract class from which the Buyer class and the Supplier class inherit from. The class provides two attributes
- *Ca.uwo.controller*: This package implements the ***Controller*** module in a ***Model-View-Controller*** architecture. It contains all classes that offer operations (i.e. the **Controller**) that change the ***Model*** (i.e. the data).
    - *Controller.java*: This class implements the *depleteStock, replenishStock,* and *createInvoice* operations. This class references all key underlying operations of the warehouse system using the attributes *createInvoiceOp, depleteStockOp,* and *replenishStockOp.* All the referenced operations implement the *perform* method *(*see *OperationInterface class).* This class keeps also a reference to the current order through the *currentOrder* attribute*.* The *Controller* class is implemented as a **singleton** class.
    - *OperationInterface.java*: This class provides the common interface for all system operations (i.e the *CreateInvoiceOperation*, *DepleteStockOperation*, *ReplenishStockOperation* and which all implement the interface (i.e. the *perform* method of the *OperationInterface* interface).
    - *Operation.java*: This is an abstract class form which all operation classes inherit from. It provides three attributes namely, the *itemRepo* attribute that references a model of the database stored in the dynamic memory as products are fetched from the data base, the *items* attribute is a hash map that links the item name to an Item object (see the ca.uwo.model package), and the attribute *dataManager* which keeps a reference to a *DataManager* object that provides the operations to connect with the underlying database and manipulate the items in the underlying data base, providing thus a layer of isolation between the *Warehouse Management System* and the underlying data base technology used.
    - *CreateInvoiceOperation.java*: This class implements through its *perform* method the creation of an invoice.
    - *DepleteStockOperation.java*: This class implements through its *perform* method the reduction of the stock by calling the *depleteItemStock* method on the corresponding *itemRepo*.
    - *ReplenishStockOperation.java*: This class implements through its *perform* method the increase of the stock by calling the *replenishItemStock* method on the corresponding *itemRepo*.

- *ca.uwo.dataAccess* package: This package provides the isolation layer between the underlying data base technology used and the *Warehouse Management System.*
    - *DataManager.java*: This class provides the operations to connect with the underlying database and manipulate the items in the underlying data base, providing thus a layer of isolation between the *Warehouse Management System* and the underlying data base technology used. It provides the methods *connect*, *createNewDatabase*, *createNewTable*, *getItem*, *insertItem*, and *updateItem*. The *DataManager* is implemented as a **singleton** class.
- *ca.uwo.Driver* package:This package provides the environment to automate the Warehouse Management System by providing a class that reads a number of buy transactions from different buyers.
    - *Driver.java*: This class has the **main** method to run the system. The **main** method reads the for the file *buyer_file* the available buyers and their IDs, and from the driver_file the different buy transactions which are to be performed from the corresponding buyers.
- *ca.uwo.frontend* package: This package provides a façade for the commands issued by the buyers and the suppliers. A Façade implements a workflow for a given command. For example, for the *placeOrder* command (method), the Façade will do a) *createOrder*, b) *depleteStock*, c) *createInvoice*, and d) *receivePayment*.
    - *Facade.java*: This class implements the FacadeCommands interface. **This is also the interface exposed by all *proxies* including the *WelcomeProxy***. (see below). This interface provides the createOrder, restock, and placeOrder methods (API).
- *ca.uwo.frontend.interfaces* package:This package provides a module containing the interfaces exposed by the whole system. In this implementation it is only the *FacadeCommands* interface that is exposed by the *WMS*.
    - *FacadeCommads.java*: This interface specifies the API (interface) exposed by the whole WMS. The Warehouse Management System exposes the *restock* and the *placeOrder* operations.
- *ca.uwo.model* package: This package implements the **Model** module in a **Model-View-Controller** architecture. It contains all classes that offer an **abstraction** of the data that can be changed the **Controller**.
    - *Item.java*: This class provides a runtime abstraction of a product found in the database. It provides the following attributes.
        - *id*: the unique Id of the item (i.e. product) as is in the database
        - *name*: the name of the item as is in the database
        - *availableQuantity*: the available quantity of the item as it currently is
        - *price*: the unit price of the item as is in the database
        - *state*: the current state of the item. The *Item* class references through this attribute an *ItemState* class which denotes the state the item is in at any given moment. An *Item* can reference an *InStockState*, a *LowStockState*, and an *OutOfStockState* class. These classes along with the *Item* class implement the **State** design pattern.
        - *viewers*: this attribute is a list of viewers which need to be notified through the method *notifyViewers*.

        In addition to the accessor and mutator methods, the *Item* class provides the *addViewer*, *removeViewer*, and *notifyViewers* methods. These methods along with the *Viewers* and the Item implement the **Observer** design pattern.

  - *ItemRepository.java*: This class provides a runtime abstraction of a products retrieved so far. It has a hash map called savedItems that maps product names to Item objects, and has a reference to the *DataManager* in the *ca.uwo.dataAccess* package to have access to back-end database operations (see *ca.uwo.dataAccess* package). The class offers the *replenishItemStock* and *depleteItemStock* methods that access the DataManager which in turn accesses the database and performs the database operations (e.g. an update on a database table entry). The *ItemRepository* is implemented as a **singleton**.

- *ca.uwo.model.item.states* package: This package is a module that holds the different states a product can be in. Each item state implements the **State** design pattern. An item state is referenced by the Item.java class (see above).
  - *ItemState.java*: This class provides the interface for all Item States implemented by the *InStockState*, a *LowStockState*, and an *OutOfStockState* classes. It provides the *deplete* and the *replentish* operations that hold the mechanics of removing or adding new products (i.e. changing the stock state, returning the result using the *ItemResult* class etc.).
  - *ItemStateFactory.java*: This class implements the **Factory** design pattern. It provides a method *create* which uses the value of its parameter to create an object of type *ItemState*.
  - *InStockState.java*: Provides the logic of the *deplete* and *replenish* methods when the item is considered as "in stock" (i.e. has available quantities in the warehouse).
  - *LowStockState.java*: Provides the logic of the *deplete* and *replenish* methods when the item is considered as "low stock" (i.e. there are less than 10 available items in the warehouse).
  - *OutOfStockState.java*: Provides the logic of the *deplete* and *replenish* methods when the item is considered as "out of stock" (i.e. the quantity of the available items is 0) (relate to Item.java constructor).

- *ca.uwo.pricingStrategies.aggregate* package: This package is a module that contains the strategies for computing the final price for the whole order (all products and their quantities included in the order).
  - *AggregatePricingStrategy.java*: Provides the interface for all aggregate pricing strategies. Specifies the calculateTotal method.
  - *AggregatePricingStrategyFactory.java*: This class implements the **Factory** design pattern which creates the appropriate aggregate pricing strategy depending the parameter passed on the *create* method defined in this factory class.
  - *AggregateDefaultPricingStrategy.java*: This class implements the default aggregate pricing strategy which just adds the price for each ordered product that is included in the whole order (note an order may be composed of orders of individual products).
  - *TestAggregatePricingStrategy.java*: This class implements a test aggregate pricing strategy which just provides a 10% discount on the price of the whole order (note an order may be composed of orders of individual products).
  - *AggregatePricingStrategyRepo.java*: This class provides an abstraction for all available aggregate pricing strategies as these are specified in the **aggr_pricing_strategy_file** file (see the constructor of the *AggregatePricingStrategyRepo* class.

The aggregate pricing strategy is used by the ***calculateInvoiceTotal*** method in the *Invoice.java* class

- *ca.uwo.pricingStrategies.individual* package: This package is a module that contains the strategies for computing the price for the individual product in an order. It has the dual structure as the aggregate pricing package above.

- *ca.uwo.proxies package*: This package is a module that contains the different Proxy classes of the system. The Proxy classes implement the ***FacadeCommands*** interface and inherit from the Proxy abstract class. All proxies are **singletons** (i.e. implement the **Singleton** design pattern).
  - *Proxy.java*: This is an abstract class from which all other proxy classes (see below) inherit from.
  - *WelcomeProxy.java*: This is the first proxy that accepts external client-side requests (see also the *ca.uwo.client* package). The *WelcomeProxy* is the first element of a **sequence of chained proxies**. These chained proxies collectively implement **the Chain of Responsibility** design pattern. The proxy which is next to the *WelcomeProxy* in the sequence of chained proxies is the *SupplierProxy* which is referenced by the *next* attribute of the *WelcomeProxy*.
  - *SupplierProxy*.java: This class implements the logic of the *restock* operation by forwarding the request to the *restock* operation of the Facade. If the request is a *placeOrder* operation request then it forwards the request to the next proxy in the sequence (i.e. the *LowQuantityProxy*) .
  - *LowQuantityProxy.java*. This class implements the logic of the *placeOrder* operation by forwarding the request to the *placeOrder* operation of the Facade. If the request is a *placeOrder* operation request and is more than 10 items then it forwards the request to the next proxy in the sequence (i.e. the *HighQuantityProxy*). The operation *placeOrder* can procced if the user is **authenticated**, so the LowQuantityProxy is this way is implementing the **Proxy** design pattern.
  - *HighQuantityProxy.java*. This class implements the logic of the *placeOrder* operation by forwarding the request to the *placeOrder* operation of the Facade. The operation *placeOrder* can procced if the user is **authenticated**, so the *HighQuantityProxy* is this way is implementing the **Proxy** design pattern. There is no other proxy in the sequence after the HighQuantityProxy. You are free to change this order and implement your own proxies.
- *ca.uwo.utils* package: This package is a module that hosts a number of utility classes.
  - *Invoice.java*: This class holds the details of an invoice (amount, break down) and also has a reference to the *AggregatePricingStrategy* to be used for calculating the final price for the whole order. This class also implements the *calculateInvoiceTotal* method. The class along with the *AggregatePricingStrategy* class implement the **Strategy** design pattern.
  - *Order.java*: This class holds the details of an order, that is the client's name, the list of items being included in this order, and a reference to the invoice.
  - *OrderItem.java*: This class holds the details of an individual item that is part of an order. It holds details such as the item name, the quantity of the item in the order, and the price that is to be computed by the associated to this class *IndividualPricingStrategy* referenced by the *pricingStrategy* field. In this respect, the class OrderItem along with the class *IndividualPricingStrategy* implement the **Strategy** design pattern. This class references an ***ItemResult*** object via its *itemResult* attribute.
  - *ItemResult.java*: This class represents holds the information about the result of a ***deplete*** or ***replenish*** operation as these are implemented in the *InStockState*, *OutOfStockState*, and *LowStockState* classes. This class holds also the **response code** of the operation as ***ResponseCode.Not_Completed*** or ***ResponseCode.Completed*** (see *InStockState*, *OutOfStockState*, and *LowStockState* classes) and a **message** to be displayed.
  - *ResponseCode.java*. This is an enumeration of possible response codes that can be produced as a result of the deplete or replenish operations as these are implemented in the *InStockState*, *OutOfStockState*, and *LowStockState* classes.

- *ca.uwo.viewer* package: This package holds the classes that need be notified when the model changes implementing thus the **View** part of the **Model-View-Controller** architectural style.
    - *Viewer.java*: This is an abstract class from which all viewers will inherit. In specifies the abstract method *inform*. The viewers are **informed** (i.e. notified) by the **Model** component.
    - StockManager.java: This class accepts is informed by the Model (see method ***notifyViewers*** in the ***Item.java*** class which calls the ***inform*** method in each viewer associated with the Item). The *inform* method here calculates the quantity of the items to be restocked using the appropriate strategy, and attaches it to a HashMap that associates the item name with the restock quantity. In a separate thread the class runs in an infinite loop the method *order*.
    - *Messenger.java*: This is a test class for which its inform method just prints a message.
- *ca.uwo.viewer.restockstrategies* package: This package holds all classes that relate to the restocking strategies.
    - *RestockStrategy.java*: This is an interface that specifies the method *calculateQuantity* that is implemented in the other classes of the package.
    - *Units50RestockStrategy.java*: This is a strategy that always restocks an item with 50 new units by implementing the *calculateQuantity* method.
    - *WeirdRestockStrategy.java*: This is a strategy that always restocks apples with 500 new items, otherwise with so many items as twice the product of remaining quantity+1 multiplied by the unit price of the item!
    - *RestockStrategyFactory.java*: This is a factory that generates *RestockStrategy* objects using the Factory design pattern. It uses the method create to generate different types of *RestockStrategy* objects depending the parameter passed in the *create* method.

driver_file (structure)

```
<buyerID> <item> <quantity> <item> <quantity> …

…..

StrategyChange <restock strategy name>

….

<buyerID> <item> <quantity> <item> <quantity> …
```

buyer_file (structure)

```
<buyer name> <buyerID>

<buyer name> <buyerID>

…
```

indiv_pricing_strategy_file (structure)

```
<item name> <individual pricing strategy name>

<item name> <individual pricing strategy name>

…
```

aggr_pricing_strategy_file (structure)

```
<Buyer name> <aggregate pricing strategy name>

<Buyer name> <aggregate pricing strategy name>

…
```

3. How to Run the System

Run the Driver class (its main) as a java application from Eclipse.

The system will execute the orders specified in the driver_file

Once the system starts printing constantly in the console the message

```
Stockmanager looking for potential orders...
Wait for orders to accumulate...
Stockmanager looking for potential orders...
Wait for orders to accumulate...
```

You can terminate the process.

4. Use Cases for Project 1

For the first deliverable you can consider the following use cases:

UC1: A user sends a *placeOrder* request to the system (i.e. the Welcome Proxy). The user needs to be authenticated. If the user is authenticated then his/her request is handled by the back-end system. Consider that there is a database with user credentials. The users can be authenticated using user name and password or just a PIN. In special cases can be authenticated by calling an agent on the phone.

UC2: *High quantity orders* (i.e. > 10 items) and *Low quantity orders* (i.e. less or equal to 10 items) should be handled in different jurisdictions (i.e. by different components). It has to be transparent to the user which component handles the request. In order to service a placeOrder request the system has a) create a new Order, b) reduce appropriately the stock for this item, c) create an invoice, and d) receive payment. You can consider that there is an external banking system on the back-end and corresponding data bases that hold the orders and keep received payment details. In exceptional cases, payment should be done in advance before the item is sent to the buyer.

UC3: Once the final price of an order is ready to be processed there may be special discounts for specific users so that the total price may not be just the sum of the prices of the individual items in the order. There are three types of buyers, the Gold status (they get a 10% discount for all orders more than 100 dollars), the Silver status (they get a 2% discount for all orders more than 500 dollars), and the Bronze status buyers (they get a discount of 2% for discount for all orders more than 1000 dollars). In exceptional cases, there may discounts of 10% for buyers approved by the manager (management decision process). Consider that there is a database that keeps the status of the buyers.

5. Tasks for Project 1

Part A.

For each use case above, draw the corresponding Use Case Diagram and the corresponding Sequence Diagram. Use operands in the sequence diagrams and stereotypes in the use case diagrams as you deem appropriate.

Part B.

Implement the following:

I1. The appropriate proxies. Make sure you implement the Chain of Responsibility design pattern and the Proxy design pattern.

I2. The placeOrder and the restock methods in the Facade.java implementing thus the Façade design pattern

I3. The authenticate method in the appropriate proxies, upon receiving a placeOrder request implementing thus the Proxy design pattern.

6. Deliverables for Project 1

Submit the following:

1. A complete report with the use case diagrams and the sequence diagrams for UC1, UC2, and UC3. The report has to have a cover page with your group number, your names, and your emails. There should be short but clear explanations of the diagrams. For example there should be short explanations of what the participants are in the sequence diagrams, what the messages mean, and what parameters in the messages mean.
2. 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 Wednesday February 12, 23:59:00.