**Object Oriented Design and Programming**

**CSCI 50700 Assignment-4**

Love Modi

*Department of Computer Information and Science*

*Indiana University Purdue University Indianapolis, USA*

**TABLE OF CONTENTS**

[**1** **Introduction** 3](#_Toc510460971)

[**1.1** **Requirements** 3](#_Toc510460972)

[**2** **Domain Model** 4](#_Toc510460973)

[**3** **Software Design** 5](#_Toc510460974)

[**4** **Implementation of Java RMI and MVC** 6](#_Toc510460975)

[**5** **Updates to the project** 7](#_Toc510460976)

[**6** **AbstractFactory, FrontController and Command Pattern** 7](#_Toc510460977)

[**7** **Authorization, Proxy and Reflection Pattern** 8](#_Toc510460978)

[**8** **RMI Concurrency Discussion** 10](#_Toc510460979)

[**9** **Sample Runs for Concurrency Tests** 11](#_Toc510460980)

[**10** **Conclusion** 13](#_Toc510460981)

# **Introduction**

The client desires an online marketplace where they can sell goods (and possibly services) to customers geographically dispersed around the world. Think Amazon but on a smaller scale and budget. Their desire is to have a system that is constructed in a portable language (Java) and makes use of their existing network. The system itself should present a view for the customer to interact with as well as a view for the employees or administrators of the company to interface with. For the customer there is a need for them to be able to browse available products – this should present the customer with the type, description and price of the item with the options to add to their shopping cart. If the customer attempts to add a quantity of the item more than the current supply the system should prevent the customer from adding these and prompt them with a message on the availability of the item. The customer should be able to also purchase their items from the shopping cart. This shopping cart should maintain state and be persistent through interactions with the application. The administrators should be able to update an item’s description within the system, update its price, and update its quantity. The administrator should also be able to remove items from the system if so desired. Administrators should be able to add other administrators as well as add/remove customer accounts. On the other hand, a customer should be able to initially register for their account by themselves. The system should handle any faults or unexpected scenarios gracefully.

## **Requirements**

Below are the requirements, segregated from client’s application description

* Separate views for customer and Administrator, so separate interface for customer and admin
* Customer should be able to browse available products
  + type, description and price of the item
  + the options to add to their shopping cart
* If the customer attempts to add a quantity of the item more than the current supply the system should prevent the customer from adding these and prompt them with a message on the availability of the item.
* The customer should be able to also purchase their items from the shopping cart
* Shopping cart should have persistent storage
* The administrators should be able to update an item’s description within the system, update its price, and update its quantity
* The administrator should also be able to remove items from the system if so desired.
* Administrators should be able to add other administrators as well as add/remove customer accounts
* administrators cannot register for accounts.
* customer should be able to initially register for their account by themselves.
* Administrator cannot purchase items in that role
* The system should handle any faults or unexpected scenarios gracefully

# **Domain Model**

Based upon feedback from the previous assignment, updated domain model is as below:

User

Username

Password

Name

Customer

CustumerId

Admin

AdminId

Shopping Cart

CustomerId

ProductId

Quantity

Amount

Product

Id

Name

Descripition

Quantity

Price

Logs in as

Logs in as

Can be registered by

Add, Update, Delete, Browse

Are added to

Purchases

Browse

Fig 1. Domain Model

Our domain will consist of the above-mentioned entities which will interact with each other to successfully fulfil the business requirements.

# **Software Design**

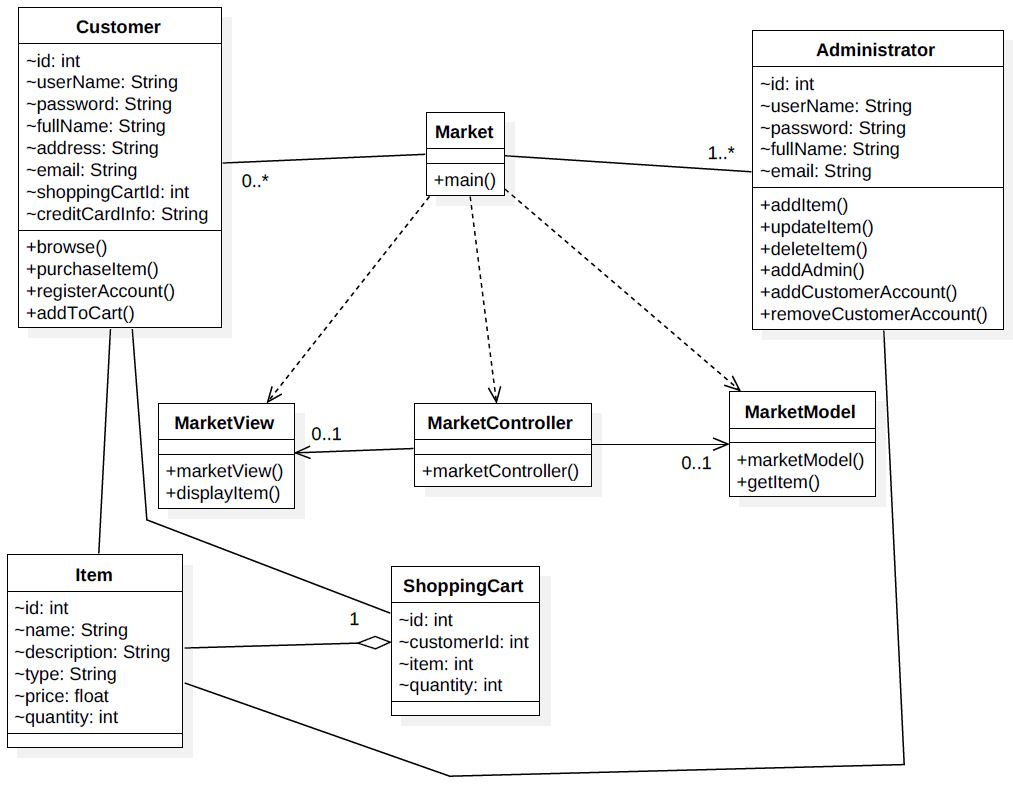


Fig 2. Software Design

The above figure shows that this application will have two types of users:

* Customer
* Administrator

Below are the entities within the domain model derived from the description:

* MarketView (View served on application startup to client)
* MarketModel (Serves as model for our MVC pattern)
* Item
* ShoppingCart
* Market
* MarketClient and MarketServer (serves as MarketController)
* Customer
* Administrator

Below are the actions that a customer would be able to perform:

* browse items
* purchase Items
* register his account
* add items to his cart

Below are the actions that an Administrator would be able to perform:

* Add an administrator
* Add/remove customer account
* Add item
* Update item description
* Delete item

# **Implementation of Java RMI and MVC**

This application is designed and implemented using the Model-View-Controller (MVC) architectural pattern and connects to remote hosts using the Java Remote Method Invocation (RMI) framework.

Java RMI is used in a client/server fashion. The MVC architecture is incorporated in this design by considering the view (MarketView) on the client side, model (MarketModel) on the server side. The MarketClient and the MarketServer act as controllers on the client and server ends, used to relay communication between client and server. The design for controllers is chosen in this way so that the application control can be handled effectively at both client and server ends. The Market class is as an interface used for RMI implementation. This way the Model, View and the Controller remain loosely coupled so that changes in one class do not necessarily affect other.

To implement Java RMI below libraries are imported at client and server end:

* Client
  + import java.rmi.Naming;
* Server
  + import java.rmi.Naming;
  + import java.rmi.RemoteException;
  + import java.rmi.server.UnicastRemoteObject;

The class java.rmi.Naming is used to bind the reference of MarketServer instance with the servers location (name) at the RMI registry. This name is used by the client to lookup the Market interface in the RMI registry to invoke remote methods that the server will execute. This registration happens using the below method:

Naming.rebind(name, market);

Here name is the server’s location and market is a reference to MarketServer class.

# **Updates to the project**

As a part of this assignment, functionalities such as add items, purchase item and browse items have been implemented. The system is connected to a MySQL database now for persistent storage. Tests were performed to test concurrent behavior in the system whose results have been discussed in section 8 of the report. No new files have been added as a part of this assignment.

# **AbstractFactory, FrontController and Command Pattern**

For this assignment abstract factory, front controller and command patterns have been implemented to ensure loose coupling, separation of concerns and high cohesion with our system while implementing the login functionality.

For abstract factory pattern, the client interacts only with the FactoryCreator class to get an MarketFactory instance cast as an AbstractFactory. The AbstractFactory class (abstract class) is implemented by the MarketFactory which in turn provides the required view to the user again cast as an (abstract) MarketView. The client can consume this object and trigger the required methods on it and only deals with the abstract layer that we built over the concrete classes that supply the object instances.

FrontController is the central controller responsible for managing different client-side views. It lies between the client and the application controllers and delegates client commands between them. For this assignment, the FrontController is responsible for delegating client authentication request to the application controllers and receiving their response. If authentication is a successful, it asks the dispatcher to dispatch the relevant view according to the user category (admin or customer). The dispatcher validates the username and requests a view from the FactoryCreator. In case of unsuccessful authentication, the FrontController rejects the client request and appropriate message is shown on the client.

Command pattern is used to perform authentication at the backend (server-side). Once the request successfully reaches the MarketServerController, it creates a user object instance. The Authenticate command instance is also created. MarketModel acts as an invoker and executes the command as soon as the client requests an event. Command interface is implemented by Authenticate class. This ensures that the MarketModel (invoker) is able to execute the command (subscribed by all commands) without worrying about the command has been implemented.

Figure below captures the entire market app updated software design until now:

*FactoryCreator*

Client

*AbstractFactory*

MarketFactory

*MarketView*

LoginView

AdminView

CustomerView

FrontController

ApplicationController

MarketModel

Authenticate

*Command*

Dispatcher

Database

AdminController

CustomerController

Fig 3. MarketApp Model with 3 design patterns

# **Authorization, Proxy and Reflection Pattern**

Authorization pattern is used to provide the application the right kind of security and access that the system requires. Role based authorization is a type of authorization pattern which uses reflection pattern and proxy pattern for implementation. In Role-based authorization, the user access to a method is determined based on the user’s role. Since we are using Java RMI the RMI interface will work as the proxy for Role-based authorization. A server implementation class was created which implements all the methods that are present in the market interface. An authorization invocation handler was also created which checks whether the user role allows access to a specific method.

An annotation class was also created which contains the annotation that has role type as an argument which will be used to match against the user role to provide access.

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

public @interface RequiresRole {

String value();

}

So, all the methods that need to have access control will have annotation like:

@RequiresRole("admin")

public void updateProduct(Session session, Item item);

Previously where the server was calling the RMI interface it will now call the authorization invocation handler which will check if the user should be provided access to the method or not.

Marketplace assignment = (Marketplace) Proxy.newProxyInstance(Marketplace.class.getClassLoader(), new Class<?>[] {Marketplace.class},new AuthorizationInvocationHandler(new ServerImpl()));

A simplified software design is shown below along with the sequence diagrams:

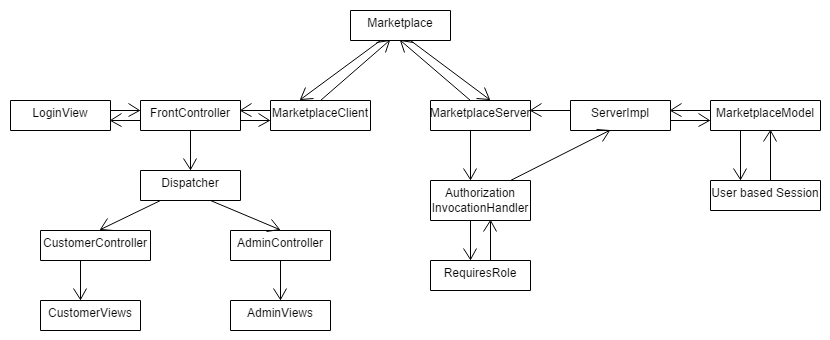


Fig 4. MarketApp Model with Authorization Pattern

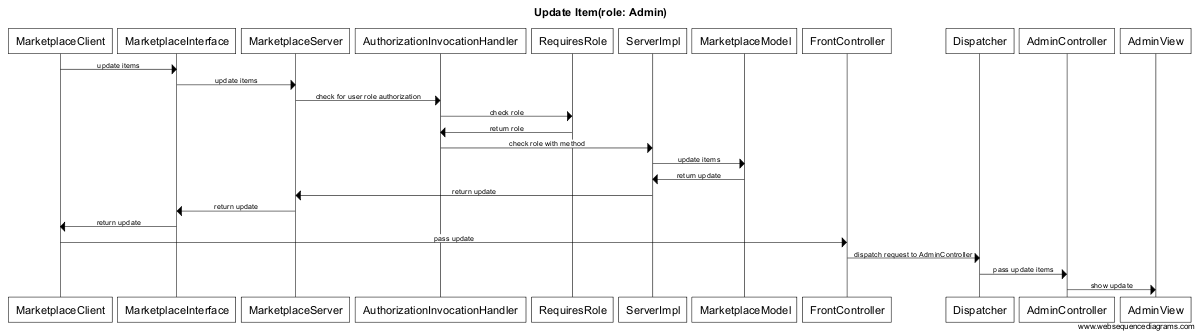


Fig 5. Sequence Diagram to Update Product

# **RMI Concurrency Discussion**

To implement add items, purchase items and browse items functionality and test concurrency, I’ve used a persistent MySQL database here. Below is the sequence diagram to implement the above-mentioned methods.

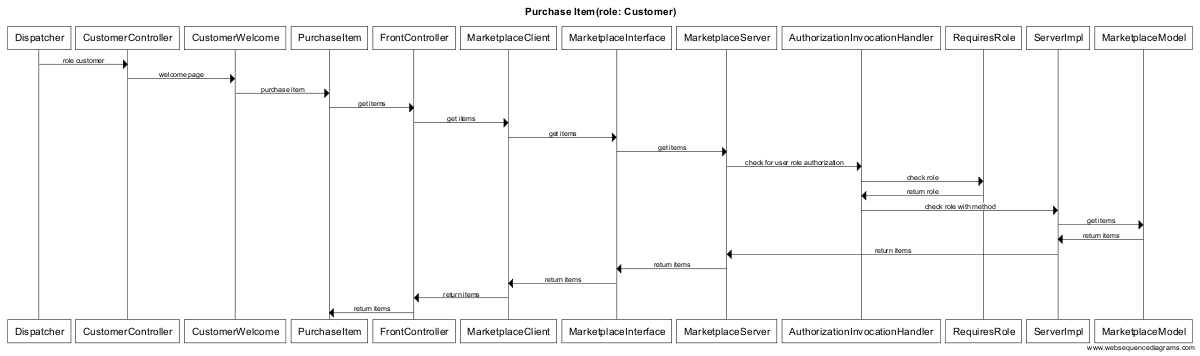


Fig. 7 Sequence Diagram for purchase item

To test concurrency, I’ve used Machine01 (10.234.136.55) as my server and the rest as my clients. Upon launching 5 clients and 1 server, RMI was able to handle the application and maintain distributed connections with all the clients. Hence it proved that RMI is a multi-threaded framework. Next, the application was tested for synchronization and concurrency.

I found that when only a single quantity of a product was left, and two users were trying to access it at the same time, the system allowed both the users to purchase the product, which is wrong. This proved that RMI does not guarantee concurrency. One solution can be to allow only one user to access the method while the other user needs to wait. This is possible using the keyword synchronized for critical methods like the purchase items, add items, browse items etc. which directly interact with the database. Snapshots for my experiments have been attached in the sample tests section.

One other problem that might occur is when two admins try to add the same product with the same price and description at the same time then two different entries will be created instead of one. Solution can be to allow only one admin to go ahead and then check the other entry if the product already exists then prompt for confirmation. Another solution can be to allow only one admin login at any time.

We need to use synchronization on the methods to let only one client access the read or write in the file. This way multiple client writing to the same file can be prevented and we may not have the above-mentioned issues.

# **Sample Runs for Concurrency Tests**

Below are sample runs that show all three methods purchaseItems, addItems and browseItems have been implemented successfully. Also, we see that when two customers try to purchase items simultaneously they are both able to buy an item although quantity in stock is only 1 for that item.

The customer can execute methods associated to customer only. Same is true for the admin.

|  |  |
| --- | --- |
|  |  |

Fig 8. Without Synchronization both customers able to buy 1 item left in stock

|  |  |
| --- | --- |
|  |  |

Fig. 9 With synchronization only one customer able to buy 1 item left in stock

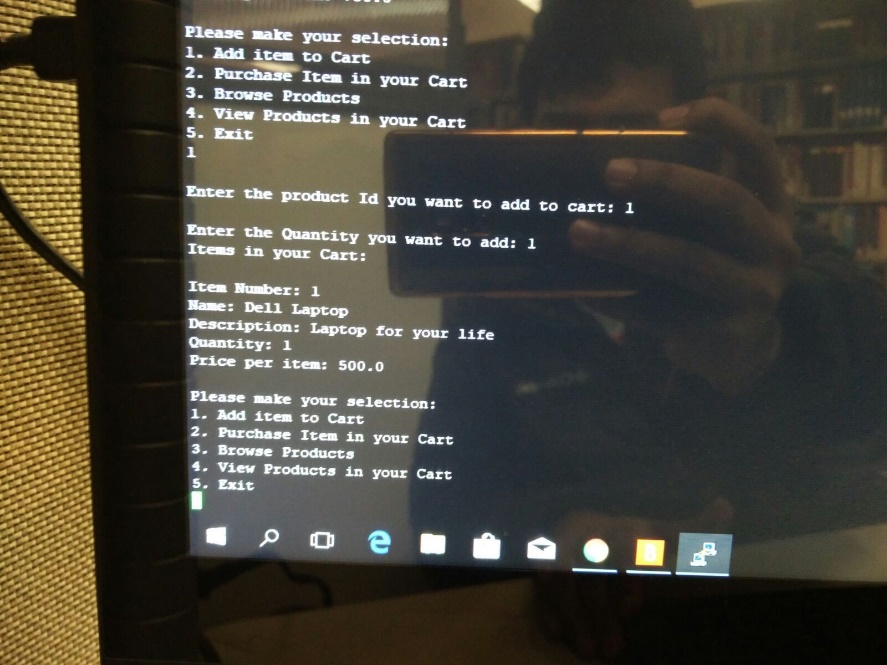


Fig 10. Add Items

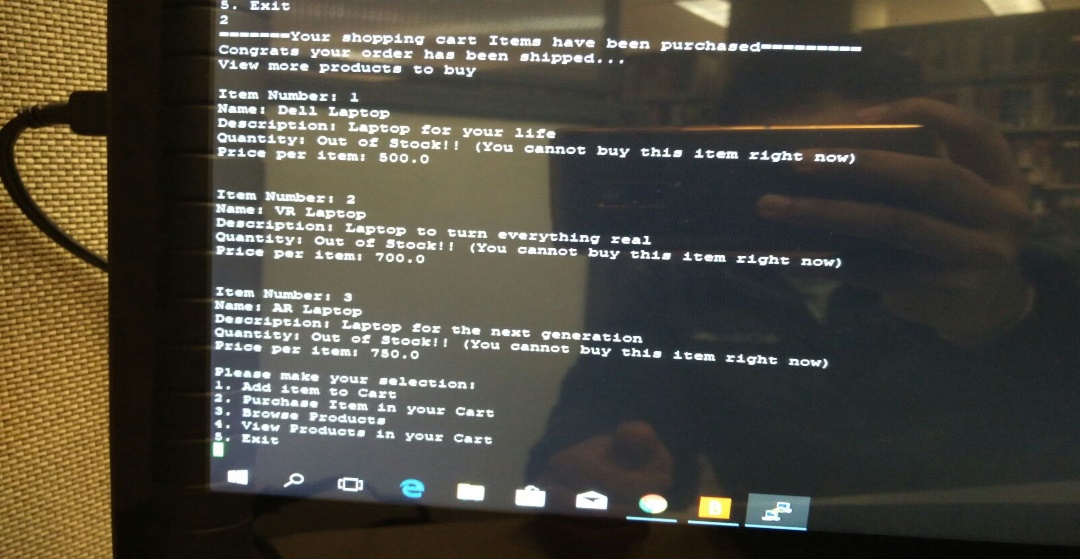


Fig 11. Browse Items

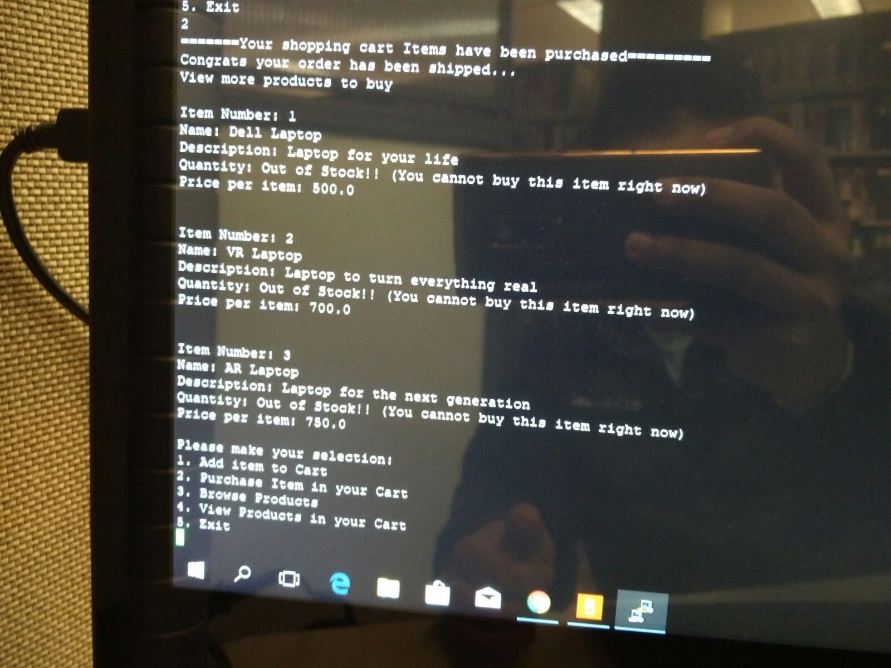


Fig 12. Purchase Items

# **Conclusion**

Three important design patterns were successfully implemented within this assignment. The most challenging part about this assignment was to understand how the different patterns would fit into our application. This involved in depth study of each pattern and eventual application. These design pattern have resulted have helped in maintaining loosely coupled classes, high cohesion and separation of concerns.