**Object Oriented Design and Programming**

**CSCI 50700 Assignment-1**

Love Modi

*Department of Computer Information and Science*

*Indiana University Purdue University Indianapolis, USA*

**TABLE OF CONTENTS**

[**1** **Introduction** 3](#_Toc505093155)

[**1.1** **Requirements** 3](#_Toc505093156)

[**2** **Domain Model** 4](#_Toc505093157)

[**3** **Implementation of Java RMI and MVC** 5](#_Toc505093158)

[**4** **Sample Runs** 6](#_Toc505093159)

[**5** **Conclusion** 6](#_Toc505093160)

# **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**

Using the above requirements, following domain model was designed

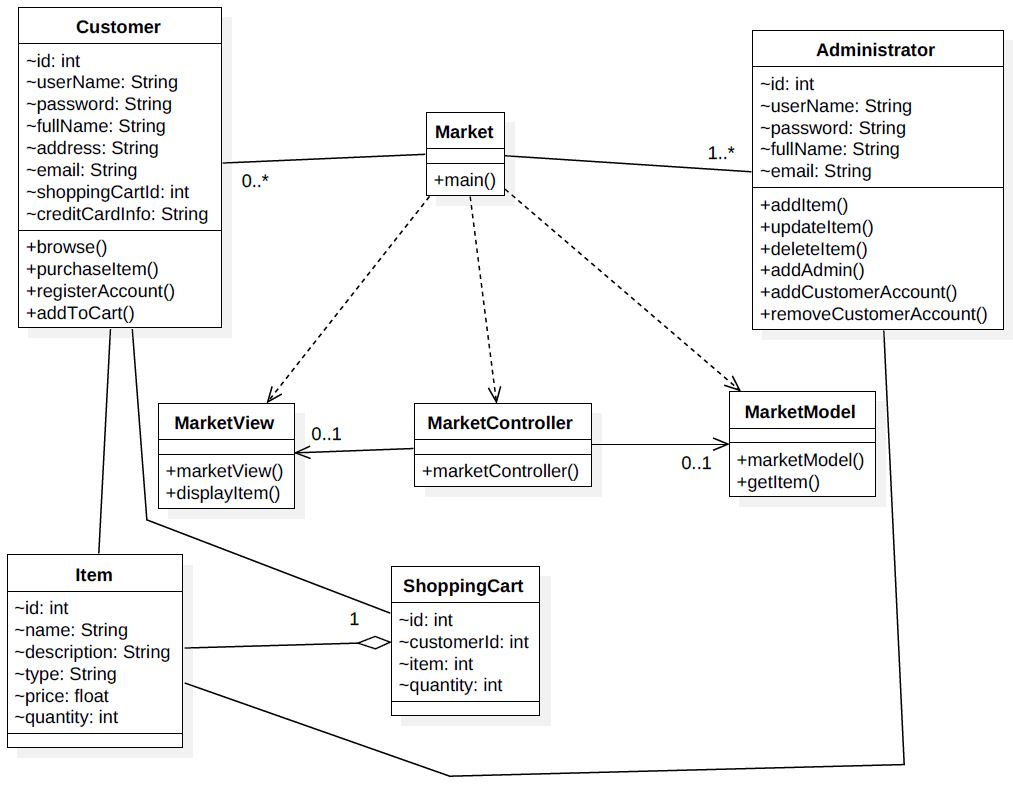


Fig 1. Domain Model for Market Application

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.

# **Sample Runs**

Below are sample runs to show that Java RMI has been implemented and runs successfully within the application

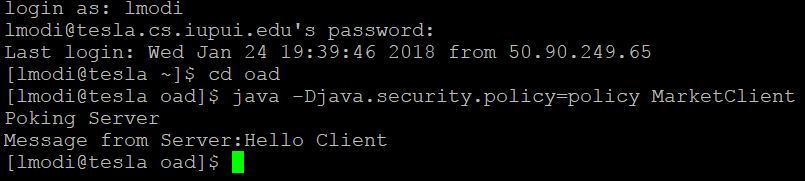


Fig 2. Application at Client

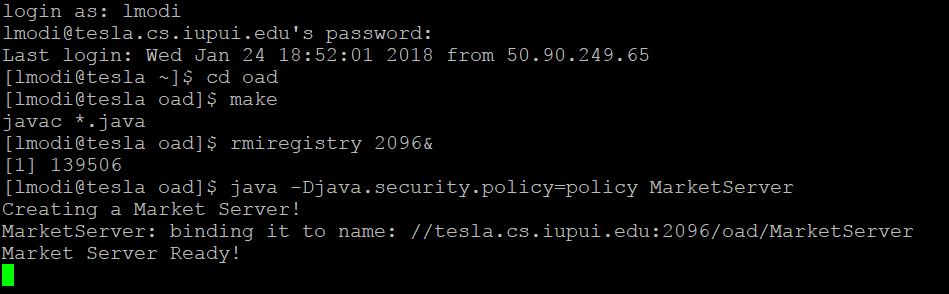


Fig 3. Application at Server

As it can be seen in Figure 3, the RMI registry is started at port 2096. The server (MarketServer) is created and bound to the server address on tesla. Once the binding is complete, Server Ready message is displayed.

In Figure 2, the client pokes the server by invoking a remote method which is executed at the server and the response is returned to the client. This along with the other entities described in section 2, form the skeleton for our application. The application will be built by modifying, adding or removing classes as per changes in the requirement or inclusion of existing requirements.

# **Conclusion**

This assignment has resulted in the learning and implementation of many concepts. Starting with a big ball of mud (client requirements) to the built up of a domain model and using it to generate a skeleton for the Marketplace application. I learned how a distributed application (built using Java RMI) could be developed using a MVC architecture. Domain model used concepts of Class diagram, hence concepts regarding Unified Modelling Language were refreshed. With an understanding of Java RMI along with MVC, the requirements make much more sense than initial phase. The application is now ready to start incorporating the client requirements by modifying and building upon this model.