**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**

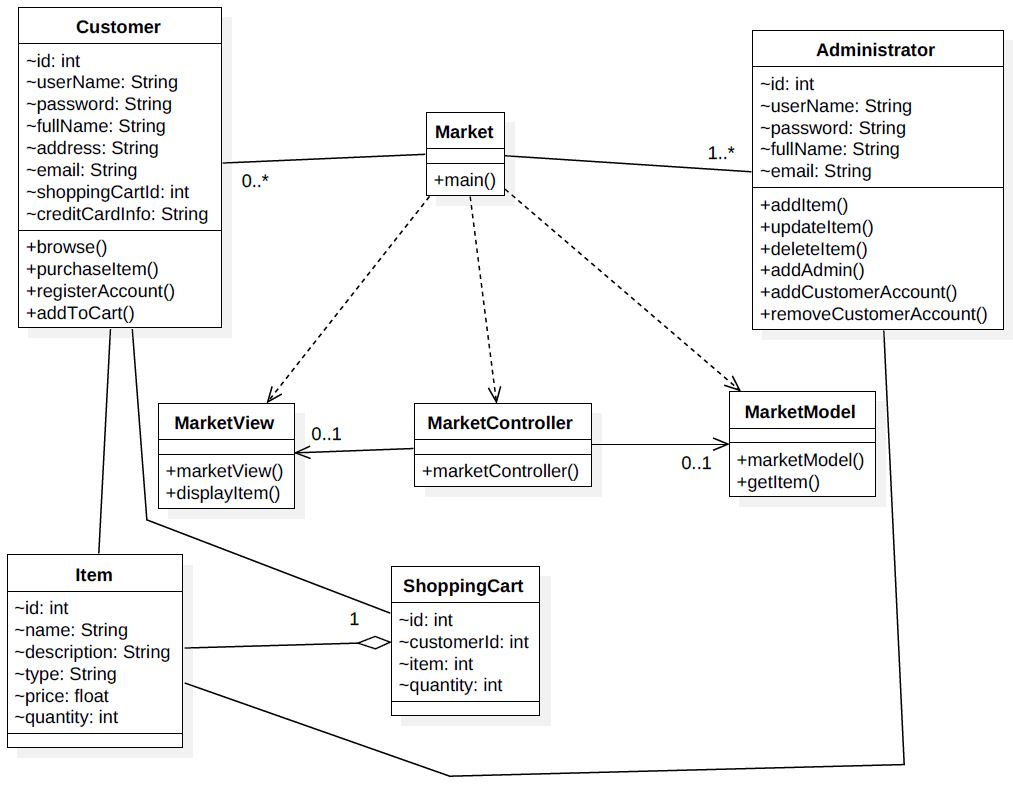


Fig 2. Software Design

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**

No new classes added to the project since assignment #4

# **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:

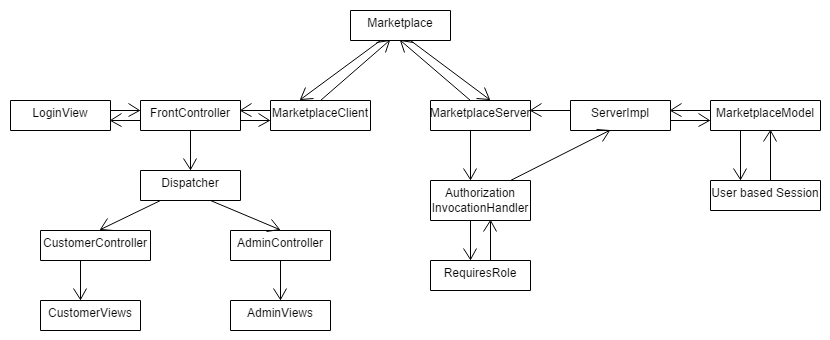


Fig 4. MarketApp Model with Authorization Pattern

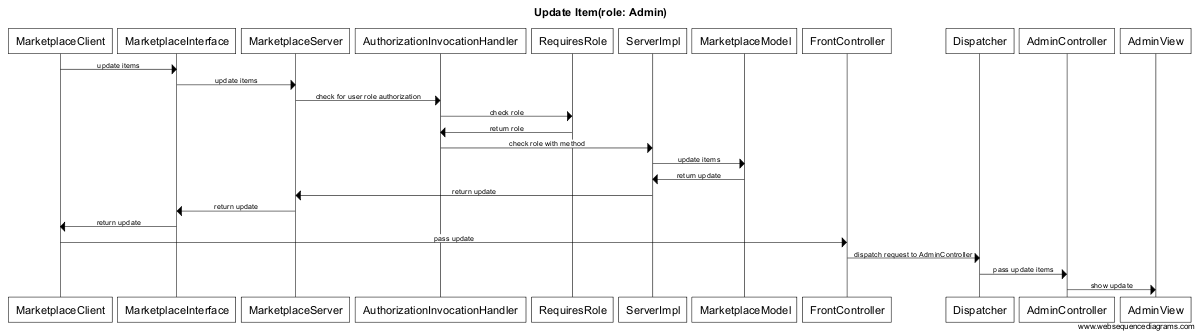


Fig 5. Sequence Diagram to Update Product

# **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.

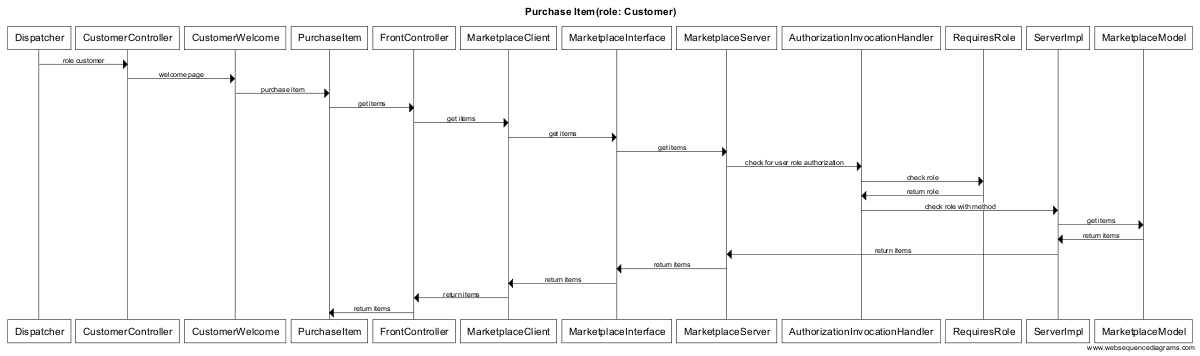


Fig. 7 Sequence Diagram for purchase item

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.

# **Synchronization**

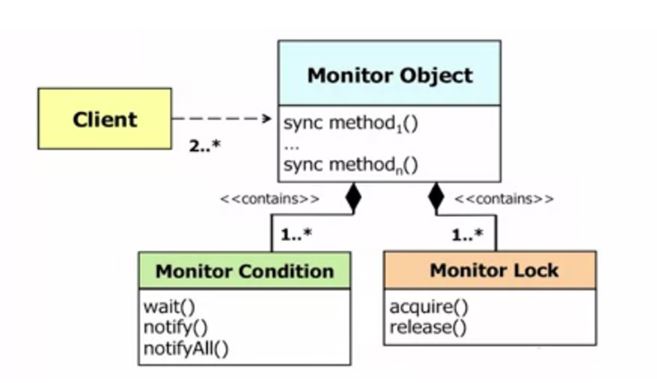
* **Assignment #5 Implementation**

In assignment #5, synchronization patterns were such as Monitor Object, Future Pattern, Guarded Suspension, Thread-safe Interface and Scoped locking were required to be implemented. Below is a brief explanation how I have used the below pattern in my application:

**Monitor Object Pattern:**

Idea behind monitor object pattern is to monitor your object and make sure that it only allows to run one method at a time in that object. So, you should provide such a mechanism that only one thread can pass through that critical section and no other thread can get into that critical section until the current thread completes its execution.

In monitor object, you need to provide mechanism for acquiring a lock when thread comes into that critical section and provide mechanism for releasing the lock, when execution of running thread completes. Following is the diagram of monitor object pattern:



Now that a basic understanding of the monitor object pattern has been established, I want to describe how it has been used in this assignment. To achieve a situation where an object level lock is acquired when the thread is trying to access a critical section and the lock is released as soon as the work is done, I have used Java’s built in synchronized keyword and applied it at method level. This is Java’s built in implementation of the monitor object pattern and can be used very easily by applying the synchronized key word at block or method level. I’ve used it on critical methods (add product, delete product, remove customer etc.) within the MarketModel class where I felt that there is need to ensure synchronization such that two customers or admins do not have to face ambiguous/ dirty data situations.

Method level use of synchronization:

public synchronized void removeItem(List<String[]> items){

//method implementation

}

As can be seen in the above example, we are using the synchronized keyword at the method level. This will put the whole method in the critical section. It automatically establishes a “happens-before” relationship with any subsequent invocation of synchronized method for the same object. This guarantees the change to the state of the object are visible to all threads. It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block execution until the first thread is done with the object. This may sometimes create unnecessary overhead in the situations where there is no need for the other thread to wait for the completion of execution of the other thread. One solution to this is using the statement level synchronization.

Block level use of synchronization:

public void removeItem(List<String[]> items){

List<String[]> catalog = new ArrayList <String[]>();

catalog.addAll(items);

synchronized(this){

removeProducts(catalog);

}

catalog.clear();

}

Unlike method level synchronization we must specify the object that provides the lock. This provides finer synchronization than the synchronized method. This helps synchronizing only the potion of the method that needs to be synchronized leaving the other portion out of the critical section.

If the entire method is part of the critical section, then there is no difference between using the method level or statement level synchronization. If the method has parts that don’t need to be in the synchronized block, then statement level synchronization is more effective to use. The disadvantage to this is that the more the synchronized blocks, the less overall parallelism we get, so it is better to keep the synchronized statement level to the minimum.

* **Future Pattern**:

Idea behind this pattern is to provide the client with a promise or a dummy object whenever the client is trying to access object or on-going computation result is not readily available. This dummy data object keeps the track of the state of running computation and once computation for the object is done, it provides the result to the client.

In my application, I have not used future pattern since I do not have any heavy computation jobs or requests running where my application will get stuck for a long period of time. Implementation of the monitor object pattern has solved my problems wherein I’m able to get a smooth running fast application with 5 clients operating in tandem.

Below is an example class where future pattern can be used:

private static final ExecutorService threadpool =Executors.newFixedThreadPool(1);

DatabaseConnection task = new DatabaseConnection ();

Future<Connection> future = threadpool.submit(task);

while (!future.isDone()) {

System.out.println("Waiting for database connection....");

Thread.sleep(5);

}

Connection conn = (Connection) future.get();

@Override

public Connection call()

{

Connection conn = null;

try {

  conn =  dbconnect();

}

catch (InterruptedException ex)

{

System.out.println(ex);

  }

return conn;

 }

public Connection dbConnect() throws InterruptedException {

Connection conn = null;

String hostname = "in-csci-rrpc01.cs.iupui.edu:3306";

String dbName = "[tablename]\_db";

String url = "jdbc:mysql://" + hostname + "/" + dbName;

String username = "[username]";

String password = "[password]";

System.out.println("Connecting database...");

try

{

conn = (Connection) DriverManager.getConnection(url, username, password);

System.out.println("Database connected!");

}

catch (SQLException e)

{

throw new IllegalStateException("Cannot connect the database!",e);

}

return conn;

}

Until the connection is made we have the future object that was returned immediately when we submitted the task, so we can continue the execution.

We keep check if the actual connection is ready using the while loop. When the connection is ready we use the future.get() to get the result. This is similar to guarded suspension discussed before. Thread is put to sleep for some time instead of aborting the thread execution.

while (!future.isDone()) {

System.out.println("Waiting for database connection....");

Thread.sleep(5);

}

When the connection is ready we use the get method to get the connection.

Connection conn = (Connection) future.get();

We throw the Interrupted Exception if there is an exception in the thread execution. Thus we do not have to wait until the connection is established to continue with the execution.

This pattern is not useful if the return value is needed immediately because the method will have to wait until the actual value is returned. This pattern is useful only if there is something to execute between the time the call is made, and the value is returned. In my implementation of marketplace, I need the connection immediately to execute the query so using future pattern is not useful so I have not used the future pattern.

* **Guarded Suspension**:

For guarded suspension, the client gets access to a thread if some condition becomes true which enables the thread to get access to the object which contains the critical section. If condition is false, then client can’t access to the code and client goes into wait status. Once the condition satisfies, all the clients are notified and the client first in the queue gets an access to that critical section within the object.

This has been achieved by implementing synchronized keyword at places where I require that once the condition is satisfied then the thread should be unblocked otherwise it should stay in wait state. Once the condition is satisfied, notify all the waiting thread to get into ready state. All this has been done using synchronized keyword.

* **Scoped Locking**

In Scoped locking, one needs to lock the scope of the code for a single thread and release the lock once the execution of that critical section is completed, hence providing thread safe environment and preventing dead lock scenarios.

I have achieved this using synchronized keyword. Once the thread enters in a critical section, the Java prohibits other threads from accessing that object containing the critical section and once the execution is complete, Java releases the acquired lock lock hence other threads can get access to the object.

More discussion about the scoped locking in the coding implementation section.

* **Thread-Safe Interface**

The idea is to put the lock at the interface level rather than the implementation level. The objective of this pattern is to prevent self-deadlock. If the lock is kept at both implementation level and interface level, there is a chance that the lock at implementation level does not get released, which will prevent the interface level lock to release its own lock, hence creating deadlock.

To implement Thread-Safe locking in my application, I used the synchronized keyword at the top of a method and not within the method at block level. This has ensured that synchronized keyword doesn’t appear at both block and method level creating deadlock condition.

Below example describes an application of thread safe interface:

public synchronized String purchaseItem(int item\_index, int item\_quan){

String message = null;

try{

Connection conn = dbConnect();

int item\_quantity = selectProduct(conn, item\_index);

item\_quantity=item\_quantity-item\_quan;

if(item\_quantity >= 0){

updateQuantity(conn, item\_quantity, item\_index);

message = "\nProduct purchase successful!!";

}

else{

message = "\nSorry not enough product left!!";

}

conn.close();

}

catch(SQLException e)

{

System.out.println(e);

}

return message;

}

private int selectProduct(Connection conn, int item\_index){

DatabaseConnect db = new DatabaseConnect();

int item\_quantity = db.selectProduct(conn, item\_index);

return item\_quantity;

}

private void updateQuantity(Connection conn, int item\_quantity, int item\_index){

DatabaseConnect db = new DatabaseConnect();

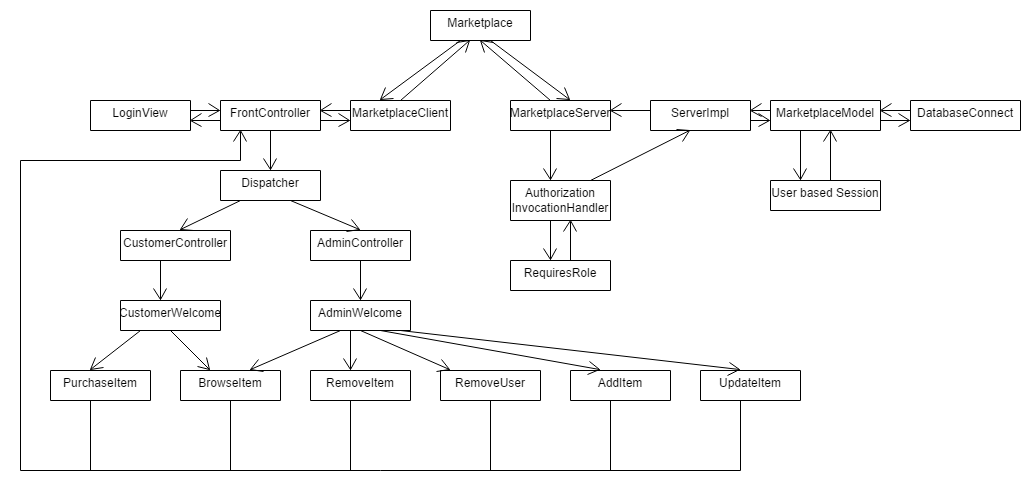
db.updateQuantity(conn, item\_quantity, item\_index);

}

The purchaseItem is a public method which calls two private methods selectProduct and updateQuantity. The synchronization is put on the purchaseItem. It will acquire the lock and implement the private methods. The private methods will not worry about the acquiring locks and perform the execution. The lock will be released by the purchaseItem when both the private methods have completed their execution.

# **Complete Functionality Implementation**

Below is a simplified software design model as of this moment:



All the functions from the product requirement are now implemented.

Customer Role:

-Login

-BrowseItem

-ViewShoppingCart

-AddToShoppingCart

-PurchaseItem

Admin Role

-Login

-BrowseItem

-AddItem

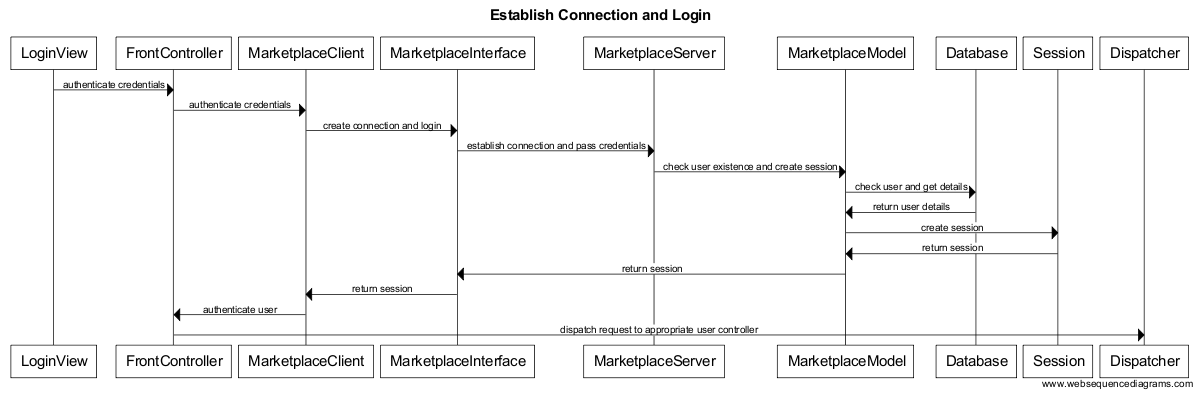
-UpdateItem

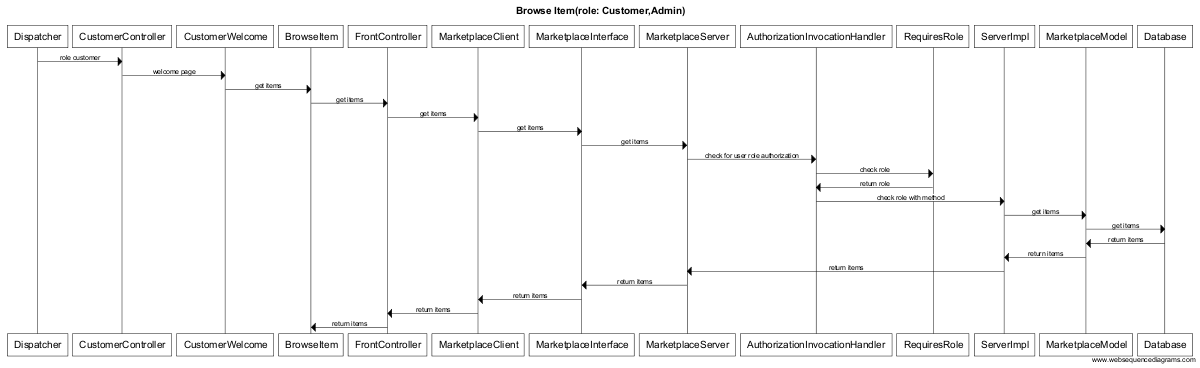
-RemoveItem

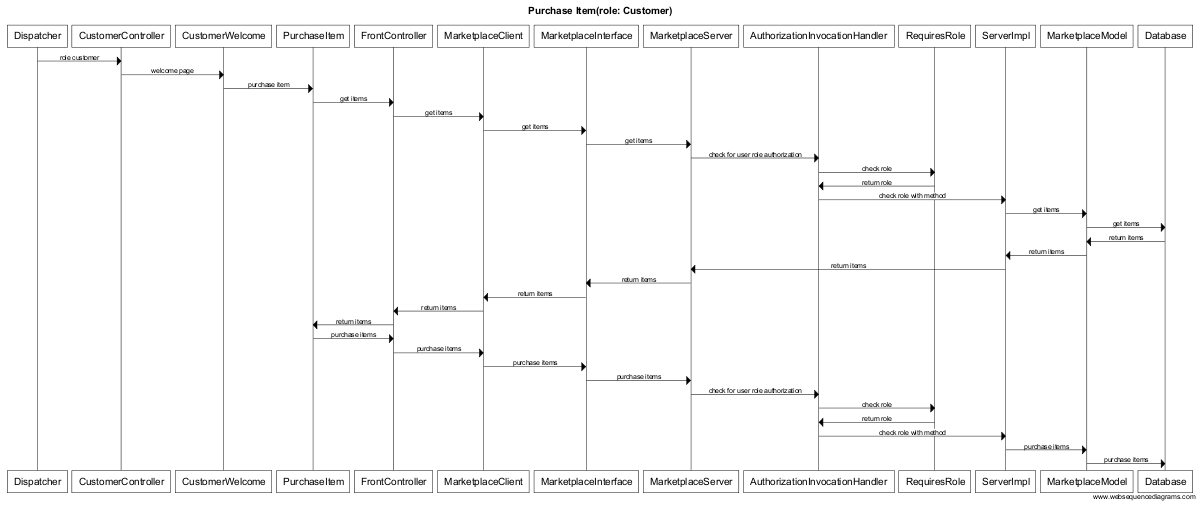
-RemoveUser

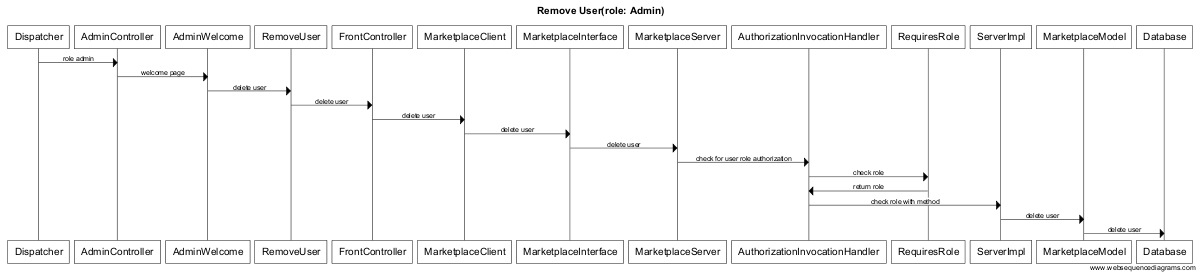
DatabaseConnect is the class used to implement all the queries required to implement the functions.

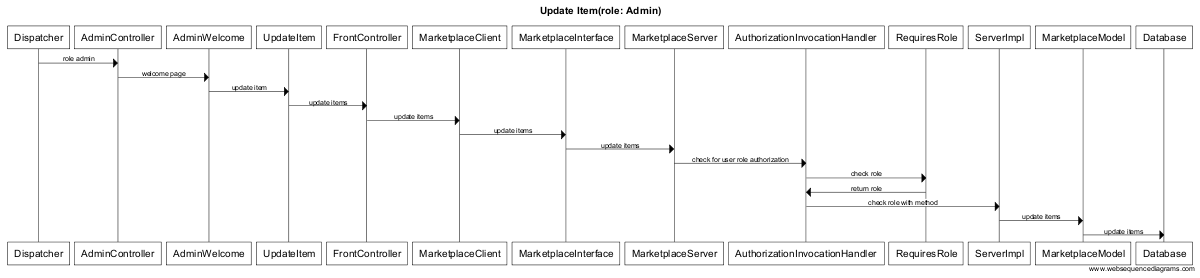
Sequence Diagrams:

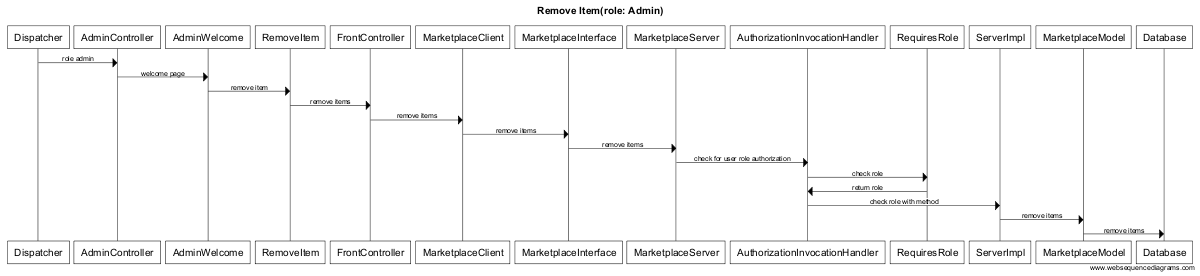


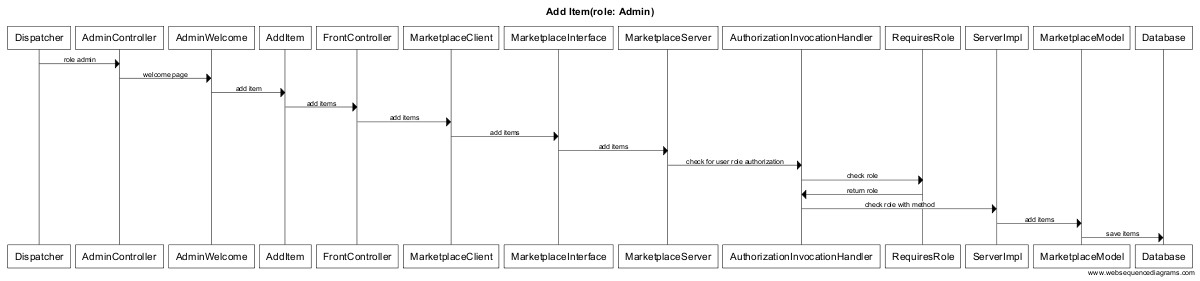












# **Sample Runs**

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

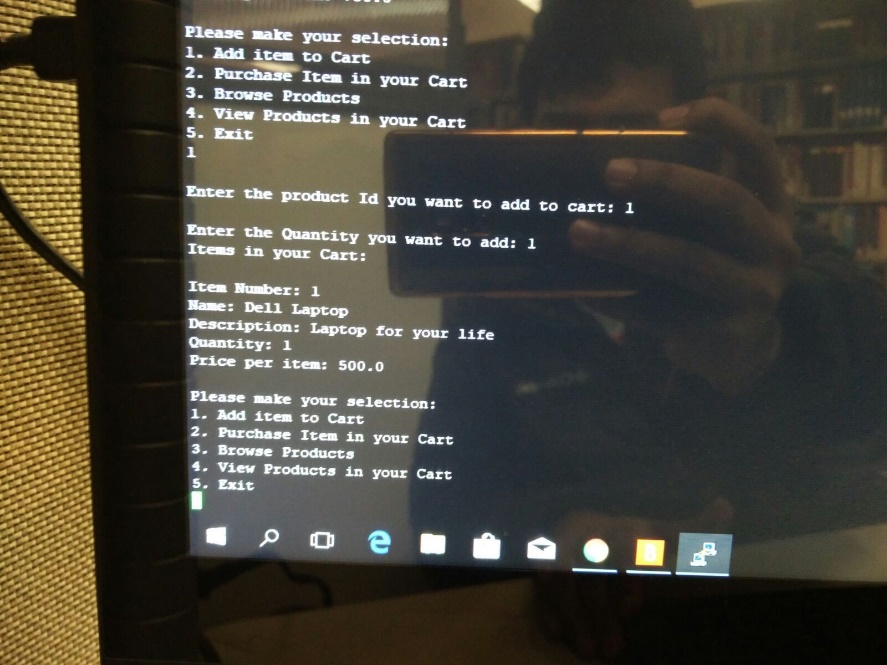


Fig 10. Add Items

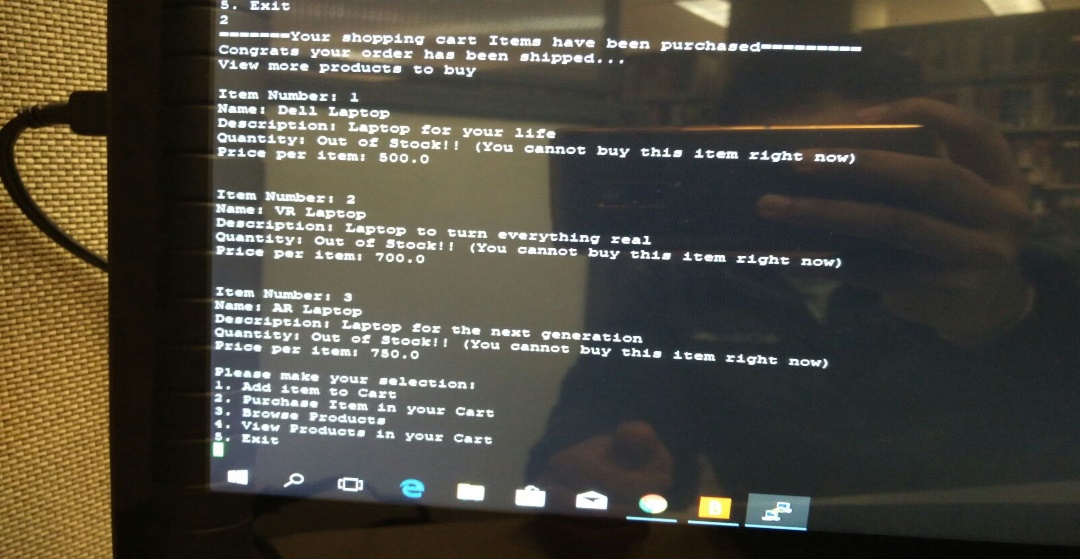


Fig 11. Browse Items

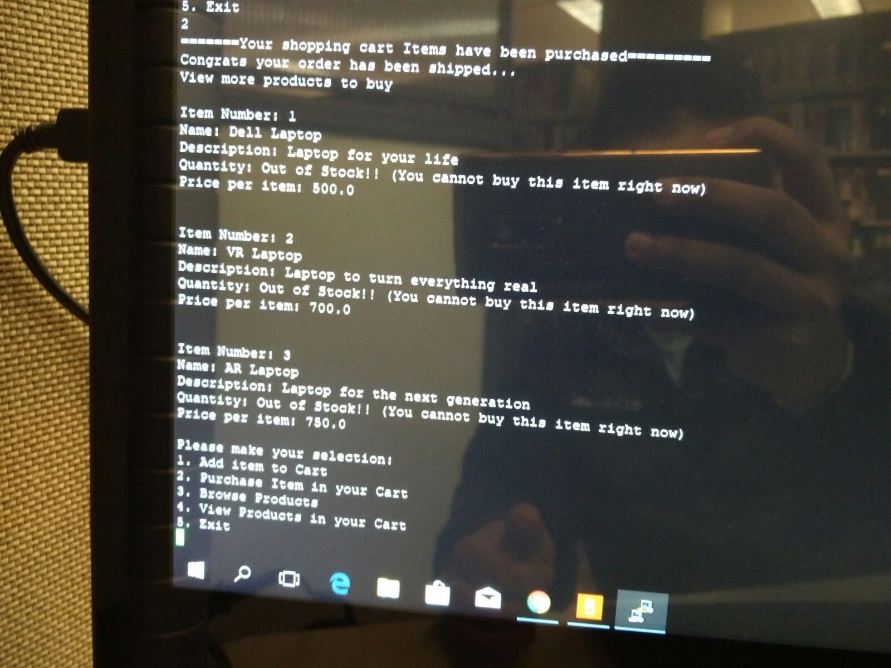


Fig 12. Purchase Items

# **Conclusion**

In this application we used different design patterns to make the application design scalable. For the client server architecture, we used Java RMI (Remote method invocation). RMI provided an interface to connect the client to the server.

Next, we decided to use the MVC architecture for the application. MVC stands for Model View Controller architecture. This helps us maintain a decoupled system. Separating the page views from the method implementation helps keep the client-side light weight. To combine MVC with the RMI, I have considered view on client side and model on server side. The RMI client is considered as client controller and RMI server as server controller. Also, an interface is implemented for the RMI. This ensures that if any changes are made in the view will not affect the implementation of that method. This is like keeping the business logic separate from the presentation layer.

Next, we implemented front controller pattern. Front controller pattern provides a centralized entry point for handling server request calls. We implemented dispatcher as an interface between front controller and other admin/customer controller. All the requests from the views will go to client controller only through the front controller. Front controller will receive the return from the server and pass to dispatcher which will then dispatch the request to appropriate user controller. Using front controller, it becomes easy to control the navigation of the system.

Next, we implemented authorization pattern- role based authorization. Role based authorization guarantees that a method will be accessed by only the user who has the permission to access that method. No other user can access it. This provides security against malicious users. Authorization invocation handler is used for this purpose. Authorization invocation handler which checks whether the user role allows access to a specific method. RMI will be used as an interface for this.

Next, we implemented synchronization patterns. Marketplace application will be accessed by multiple clients on a single server. This makes the necessity to make sure that all the calls performed by different clients do not override each other. All the clients’ request should be performed such that there is only that client. To ensure that it is important to implement synchronization patterns. In my application as per my design I have used thread-safe synchronization and monitor object synchronization. There was no need for future pattern in my application, so it was not implemented.

The implementation of these patterns has helped make the system easily scalable and easy to test. Changes in view will not affect the implementation. Even in a distributed environment where there are multiple client calls the application will be easily able to handle it. The implementation on client side is light weight. All the processing is done on the server side making it easy at the presentation layer. The system has less coupling and a fair amount of cohesion. From the security point of view also the application will not provide access to unauthorized users and will not let user access methods they are not allowed to access. This case study has also help helped understand when and how to use design pattern and most importantly when not to use what pattern.