IUPUI

Online Marketplace

Assignment 5

Saurabh Pramod Pandey
2000201934

1. Introduction:

1.1 Marketplace Ecosystem:

The aim of this project was to implement an online marketplace (Like Amazon but smaller in scale). For this, I built an application that presents two portals, one for customers and the other for the administrators of the company to interact with the system. The application provides various functionalities for both customers and administrators.

Functionalities for customers:

- *Registering with the marketplace.*
- *Logging into the system.*
- *Browsing through the products.*
- *Adding products to the cart.*
- *Removing products from the cart.*
- *Purchasing the products.*

Functionalities for administrators:

- *Logging into the system.*
- *Browsing through the products.*
- *Adding new items in the system.*
- *Removing items from the system.*
- *Updating existing items in the system.*

I built the system using Java programming language with the help of various object oriented design paradigm and patterns.

1.2 Scope:

- **Focus:** *Analyzing concurrency and synchronization with respect to Java and Java RMI.*
- **Functionality implemented***: All functionalities mentioned in section 1.1.*
- **Patterns Implemented***:*
  - *Monitor Object.*
  - *Guarded Suspension.*
  - *Future.*
  - *Scoped Locking.*
  - *Thread-safe Interface.*

2. Technical Description:

2.1 Domain Model:

- **Basic Framework diagram:**

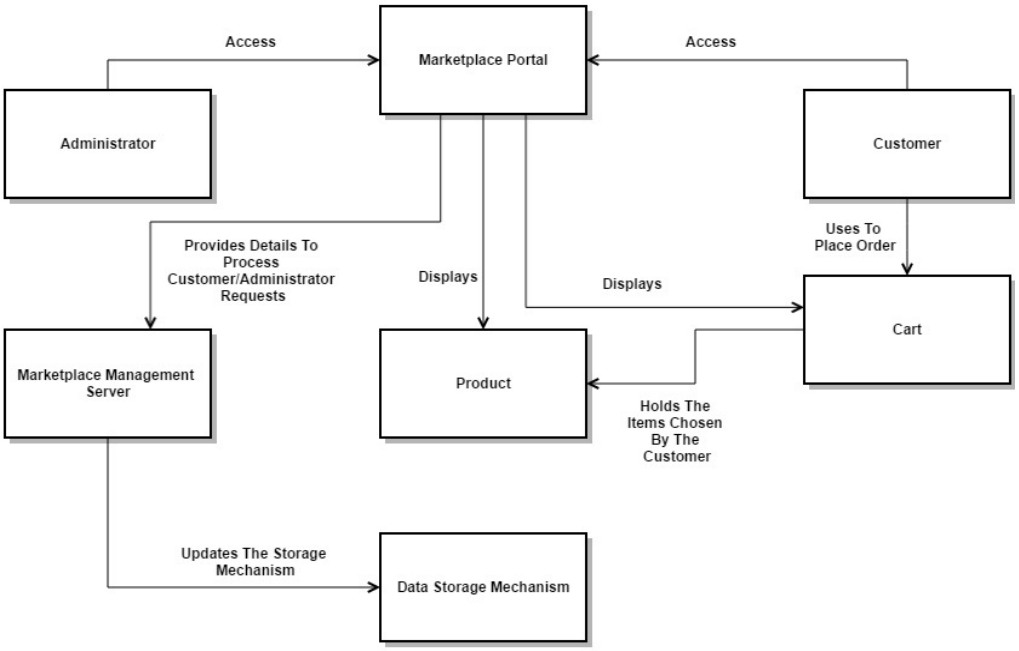


Figure 1

- **UML Class diagram:**

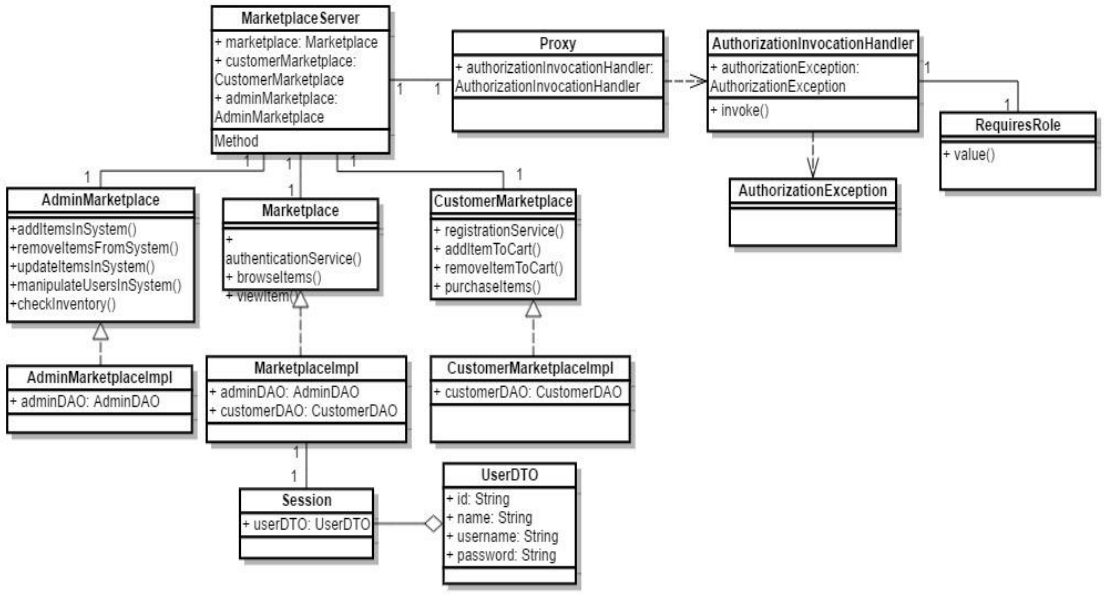
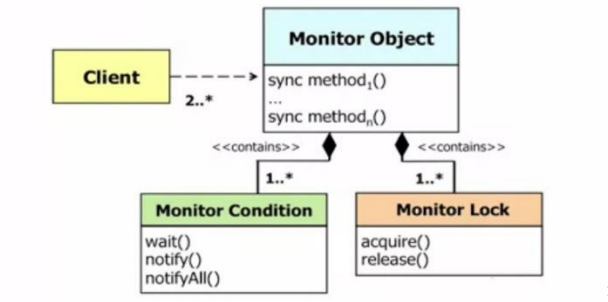


Figure 2

2.2 Patterns Implemented:

2.2.1 Monitor Object.



[1]

Figure 3

With distributed computing, it is often seen that the objects are required to be shared among multiple clients. Multiple client threads can concurrently invoke methods of an object. Thus, it becomes essential to synchronize the access to these objects for maintaining the consistency and reliability of the system. One way of achieving this is by using 'synchronized' keyword of java with the methods. It ensures that only one thread can access/execute a method at a time. This allows all the execution sequences of the method to be scheduled thereby maintaining the consistency of state and data within the application. This also helps in writing concurrent software much easily without programmatically being concerned about the need of obtaining and releasing locks. In my application, I have applied this pattern on deleteProduct() method through the use of synchronized keyword. Figure 3 shows various components of the Monitor Object pattern which are as follows.

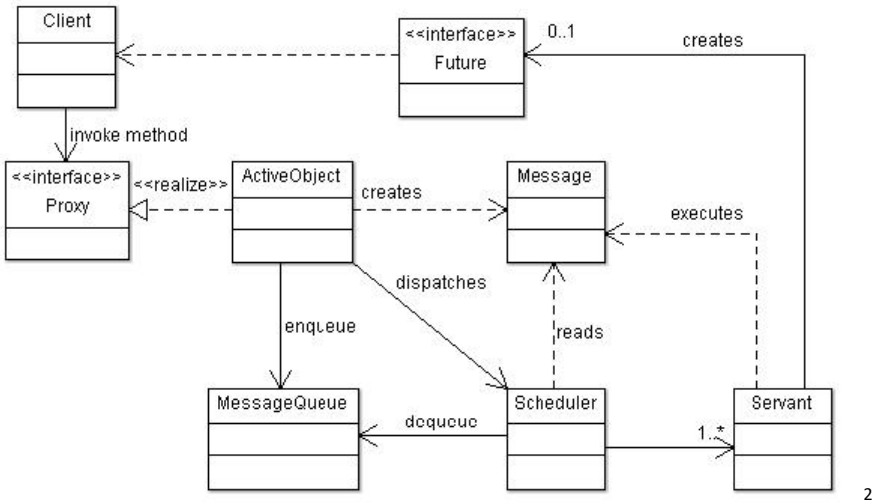
- *Monitor Object*
  It exposes synchronized methods as the only means of accessing the methods for the client.
- *Synchronized Methods*
  They ensure thread-safe access to the methods ensuring that the consistency of the application is maintained.
- *Monitor Lock*
  It is concerned with the monitoring of acquisition and release of the locks. It is used by the synchronized methods for serializing the invocation of method.
- *Monitor Condition*
  It helps in monitoring conditions like wait, notify and notifyAll to ensure proper suspension and resumption of threads. Thus, it ensures proper execution scheduling of the threads.

[1] Lecture Slides – 20 of Dr. Ryan Rybarczyk

2.2.2 Guarded Suspension

Guarded suspension is another synchronization pattern that helps in maintaining the consistency of the system. It helps in controlling access to a region of the code and ensures that the access is granted only when certain conditions are met. It is similar to a crossing guard that allows passage only when certain conditions are met. As discussed in section 2.2.1, it is important to ensure proper execution scheduling of the threads accessing a common method. Guarded suspension helps in achieving this by ensuring that a method invoked is executed only when certain conditions are met. It ensures that a method call is not aborted right away if the precondition is not met. It blocks the execution of the thread by invoking wait() method on the thread. This allows other threads to safely access the common shared resource and change the state of the method's guard condition. Now, if the condition is satisfied, the blocked thread can resume it execution and invoke the method. Resumption of threads can be achieved through notify() and notifyAll() methods. Thus, this pattern works closely with monitor object pattern to ensure synchronization. With Java, these can simply be realized by using a 'synchronized' keyword with the method or a block.

2.2.3 Future



[2]

Figure 4

Future pattern helps in optimizing runtime performance and efficiency of the system. While running an application in a single-threaded environment, a function call is returned only when the function has finished its execution. In case of multi-threaded environment, we may need to perform other tasks while the corresponding invoked function is being

[2] Lecture Slides – 22 of Dr. Ryan Rybarczyk

executed. This is especially required if the called function is too large or requires large computation time. Also, later we should be able to access the resources that were requested to be worked out by the called function. Future pattern helps in achieving this by returning a virtual object (called future) to the caller as soon as the request is invoked as shown in figure 2. Through the future object, the caller can keep a track of the called function's execution. A value is made available to the caller only once the computation of the called method is completed. In short, the future object can be seen as a virtual proxy through which the execution of the called method can be monitored.

For the scope of my application, considering a holistic view of my design, I did not find this pattern applicable to any scenario. I implemented this pattern with the browse items functionality to ensure that the user is displayed with a message ("Fetching the products") when the user invoked the functionality. However, looking at the complexity and the intent of the future pattern, this implementation of the pattern looked trivial and thus I decided to drop the implementation of this pattern from my application.

2.2.4 Scoped Locking

Scoped locking pattern ensures that the locks are acquired and released automatically as desired. As with synchronization, it is known that the locks must be acquired when a critical section of the code is being executed and released when the critical section ends. It is possible to let the programmers decide when to acquire and release locks. However, it is often hard to predict all the scenarios through which the locks must be released. This can result in a resource getting locked forever which can ultimately lead to situations like deadlock. Java helps in proper acquisition and release of locks automatically through the use of keyword 'synchronized'. Java ensures that locks are acquired as soon as the code enters the critical section. Also, that the locks are released as soon as the code exits the critical section from any path. This helps in minimizing the errors caused by humans when dealing with locks. In addition, Java also helps in establishing a 'happens-before' relationship at the end of the critical section. These together help in maintaining consistency within the system. As programmers, it is essential to use the 'synchronized' keyword within the application. It is necessary to correctly the identify the scope of the code on which the locks must be acquired. Locking has a significant cost associated with it. Applying lock on sections which should not a be part of the critical section can significantly impact the performance and efficiency of the system. Thus, it might be advisable to use synchronized statements instead of synchronized methods wherever possible within the application. I have implemented scoped locking through use of 'synchronized' statements within the addProduct() method of the ProductDAOImpl.java class.

2.2.4 Thread-Safe Interface

Within any application, it is often observed that there are many intra-component calls. It is necessary for each of the components to acquire locks for protecting their critical sections. However, it becomes too expensive if each of the related components sequentially acquire and release locks. Also, there may occur scenarios where the system can enter a self-deadlock, waiting for a lock on a resource already locked by itself. Thread-safe pattern helps in dealing with such scenarios by exposing a publicly accessible interface. The methods in the implementation class of the thread-safe interface are responsible for acquiring and releasing the locks. Also, these methods are responsible for invoking other methods which implement the desired functionality. This ensures that none of the internally invoked methods are required to deal with the locking mechanism. The internally invoked methods assume that the lock is already acquired. It is desirable to declare these methods as private so that they can only be accessed through a common public method which is invoked by the method of the thread-safe implementation class. Thus, with thread-safe interface pattern, locks are acquired only at the border of the component. This helps in increasing the robustness and enhancing the performance of the system. However, with this pattern, there are possibilities of misuse which may lead to deadlocks. Also, there may be chances of potential overhead associated with it.

For the scope of this application, I implemented the thread-safe interface pattern for the ‘update items’ functionality associated with the administrators. In this, AdminMarketplaceImpl.java class is the implementation class for an interface AdminMarketplace.java (Thread-safe). The updateItemsInSystem() method of the thread-safe implementation class deals with the locking the mechanism. Other function invoked by this method, updateProduct() (of ProductDAOImpl.java class) which in turn invokes getProduct(), need not acquire and release locks. Ideally, the method invoked by the thread-safe implementation class should all be private. However, in with my design the methods invoked are the methods of an interface (DAO pattern implementation) and hence cannot be made private. Thus, I decided to keep the invoked methods as public. This helps in reducing the number of occasions where locks are acquired and released thereby improving the efficiency of the system. Also, as evident, it ensures that none of the methods enter a state of self-deadlock where it is waiting to acquire a lock already acquired by itself.

2.3 Implemented Functionalities:

As part of assignment 5, I have implemented the following functionalities in the marketplace application.

- **Remove Item From Cart.**

  Removing items from the cart is a functionality specifically associated with the customers. As discussed in the report of assignment 4, before purchasing items, the customers should add the items into the cart. Similarly, the customers can also remove the items from the cart if they wish to do so. It is important to note that the item must be present in the customer's cart before he/she attempts to remove it. Otherwise, the system will display of message that the requested item cannot be removed.

- **Remove Items (From System).**

  Removing items from the system is a functionality that is specifically associated with the administrators. Therefore, to remove products in the system, the role of the user must be that of administrators. When an administrator logs into the system, he/she is presented with an option to remove items from the system. Selecting to do so, the administrator is prompted to enter ID of the product to be removed. Entering it correctly, administrator can successfully remove the item from the system. Administrators can then browse items in the system to verify that the product was correctly removed from the system.

- **Update Items (In the system).**

  Updating items in the system is a functionality that is specifically associated with the administrators. Therefore, to update products in the system, the role of the user must be that of administrators. When an administrator logs into the system, he/she is presented with an option to update items in the system. Selecting to do so, the administrator is prompted to enter the product parameter that needs to be updated: Description, Quantity or Price. Selecting an option and entering the new value for the corresponding parameter, administrator can successfully update the item in the system. Administrators can then browse items in the system to verify that the product was correctly updated in the system.

- **Database Integration.**

  With this assignment, a persistent backend storage mechanism has been integrated with the application. The database used was MySQL. As per the design of my application, I had to persist two types of records: Users and Products information. Thus, these two tables were created with some dummy values to demo the functionalities of the application.

3. Phases of Development:

3.1 Assignment 1

In assignment 1, the scope was to build the domain model for developing the marketplace application. Along with it, Java RMI was implemented that provided the basic infrastructure to enable client-server communication.

3.2 Assignment 2

With assignment 2, the goal was to improve the design of the system by implementing some key application control patterns which included the Front Controller pattern along with the Command and Abstract Factory patterns. This provided the basis for implementing the login functionality.

3.3 Assignment 3

In assignment 3, the goal was to add some security mechanisms in the application through the implementation of Role Based Access Control, Authorization pattern. This was achieved through the use Java Annotations along with some other patterns like Dynamic Proxy and the Reflection pattern.

3.4 Assignment 4

In assignment 4, the aim was to analyze the concurrency mechanisms available with distributed systems. The focus was on understanding how concurrency is handled in Java and Java RMI. Also, what could be the potential challenges that might cause concurrency and synchronization issues with the online marketplace application.

3.5 Assignment 5

In the final assignment, the aim was to fully implement all the functionalities associated with the customers and administrators of the system. Also, to integrate the system with a persistent storage mechanism: MySQL. The key focus was on implementing patterns related to synchronization within the application which included patterns like: Monitor Object, Future, Guarded Suspension, Scoped Locking, and Thread-Safe Interface.

4. Code Snapshots:

- **deleteProduct():**

```
    // Method to delete a product from the system
    @Override
    public synchronized String deleteProduct(String productID) {
        ProductDTO existingProduct = getProduct(productID);

        if (existingProduct != null) {

            query = "DELETE from Product WHERE ID = '" + productID + "'";

            try {
                stmt.executeUpdate(query);
            } catch (SQLException e) {
                e.printStackTrace();
            }

            return "Product Deleted Successfully!";
        }
        else
            return  "Product Not Found!";
    }
}
```

- **DatabaseConnector.java:**

```
public Statement getStatement() {

    String hostname = "localhost:3306";
    String dbName = "pandey_db";
    String url = "jdbc:mysql://" + hostname + "/" + dbName;
    String username = "pandey";
    String password = "****";
    Connection conn = null;

    try {
        Class.forName("com.mysql.jdbc.Driver");
        conn = (Connection) DriverManager.getConnection(url, username, password);

    } catch (SQLException e) {
        throw new IllegalStateException("Cannot connect the database!", e);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    if (conn != null) {
        Statement stmt = null;

        try {
            return (Statement) conn.createStatement();

        } catch (SQLException e1) {
            System.err.println("Unable to create SQL statement!");
            e1.printStackTrace();
            return null;
```

- **removeItemFromCart():**

```
public Object[] removeItemFromCart(Session session, String productID, int quantity) throws RemoteException {

    String response = "Cannot Remove Requested Item!";
    Object o[] = new Object[2];

    synchronized (this){
        CustomerDTO customerDTO = (CustomerDTO) session.userDTO;
        CartDTO cart = customerDTO.getCart();
        ProductDAO productDAO = ProductDAOImpl.getInstanceOf();
        ProductDTO product = productDAO.getProduct(productID);
        if (cart.getCurrentItems() != null) {
            Iterator i = cart.getCurrentItems().entrySet().iterator();
            while (i.hasNext()) {
                Map.Entry val = (Map.Entry) i.next();
                String pid = (String) val.getKey();
                int quantity_in = (Integer) val.getValue();

                if (pid.equals(productID) && quantity_in >= quantity) {

                    // System.out.println("Before in stock: "+productID+" : "+product.getQuantity());
                    //product.setQuantity(product.getQuantity() + quantity);
                    String value = String.valueOf(product.getQuantity() + quantity);
                    productDAO.updateProduct(productID, property: "Quantity", value);
                    // System.out.println("After in stock: "+productID+" : "+product.getQuantity());

                    cart.setCurrentItems(productID, -quantity);
                    customerDTO.setCart(cart);
                    session.userDTO = customerDTO;
```

- **updateItemsInSystem():**

```
public String updateItemsInSystem(Session session, String productID, int property, String value) {
    String p = null;

    synchronized (this){
        ProductDAO productDAO = ProductDAOImpl.getInstanceOf();
        ProductDTO productDTO = productDAO.getProduct(productID);
        if(productDTO != null){
            if(property == 1)
                p = "Description";
            if(property == 2)
                p = "Quantity";
            if(property == 3)
                p = "Price";
            return productDAO.updateProduct(productID, p, value);
        }else
            return "Product Not Found!";
    }
}
```

- **getCustomer():**

```
public CustomerDTO getCustomer(String username) {
    //return customers.get(username);
    query = "SELECT * FROM User WHERE ROLE = 'CUSTOMER' AND USERNAME = '"+username+"'";

    try {
        rs = stmt.executeQuery(query);
        if(rs.next()){
            String id = rs.getString( columnLabel: "ID");
            String name = rs.getString( columnLabel: "NAME");
            String uname = rs.getString( columnLabel: "USERNAME");
            String pwd = rs.getString( columnLabel: "PASSWORD");
            String role = rs.getString( columnLabel: "ROLE");
            return new CustomerDTO(id, name, uname, pwd, role);
        }
    } catch (SQLException e) {
        e.printStackTrace();
        return null;
    }

    return null;
}
```

- **updateProduct():**

```
public String updateProduct(String productID, String property, String value) {
    ProductDTO existingProduct = getProduct(productID);

    if(existingProduct != null) {
        double newValue = 0;

        if(property.equals("Quantity"))
            newValue = Integer.parseInt(value);
        else if(property.equals("Price"))
            newValue = Float.parseFloat(value);

        if (property.equals("Description"))
            query = "UPDATE Product SET " + property + " = '" + value + "' WHERE ID = '" + productID + "'";
        else
            query = "UPDATE Product SET " + property + " = " + newValue + " WHERE ID = '" + productID + "'";

        try {
            stmt.executeUpdate(query);
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return "Product Updated Successfully!";
    }
    else
        return  "Product Not Found!";
}
```

5. Sample Run Snapshots:

- **Compiling Classes:**

```
[pandey@in-csci-rrpc01 csci50700_spring2017_marketplace]$ make -f makefileServer
cd main/storage; make
make[1]: Entering directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/storage'
javac -d ./../../out -classpath ./../../out/mysql-connector.jar DatabaseConnector.java
make[1]: Leaving directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/storage'
cd main/model/data/transfer; make
make[1]: Entering directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/model/data/transfer'
javac -d ./../../../../out -classpath ./../../../../out UserDTO.java
javac -d ./../../../../out -classpath ./../../../../out ProductDTO.java
javac -d ./../../../../out -classpath ./../../../../out CartDTO.java
javac -d ./../../../../out -classpath ./../../../../out AdminDTO.java
javac -d ./../../../../out -classpath ./../../../../out CustomerDTO.java
make[1]: Leaving directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/model/data/transfer'
cd main/model/data/access; make
make[1]: Entering directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/model/data/access'
javac -d ./../../../../out -classpath ./../../../../out ProductDAO.java
javac -d ./../../../../out -classpath ./../../../../out AdminDAO.java
javac -d ./../../../../out -classpath ./../../../../out CustomerDAO.java
javac -d ./../../../../out -classpath ./../../../../out UserDAO.java
javac -d ./../../../../out -classpath ./../../../../out:./../../../../out/* ProductDAOImpl.java
Note: ProductDAOImpl.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
javac -d ./../../../../out -classpath ./../../../../out:./../../../../out/* CustomerDAOImpl.java
javac -d ./../../../../out -classpath ./../../../../out CartDAOImpl.java
javac -d ./../../../../out -classpath ./../../../../out:./../../../../out/* AdminDAOImpl.java
javac -d ./../../../../out -classpath ./../../../../out UserDAOImpl.java
make[1]: Leaving directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/model/data/access'
cd main/model/authorize; make
make[1]: Entering directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/model/authorize'
javac -d ./../../../out -classpath ./../../../out AuthorizationException.java
javac -d ./../../../out -classpath ./../../../out RequiresRole.java
javac -d ./../../../out -classpath ./../../../out Session.java
javac -d ./../../../out -classpath ./../../../out AuthorizationInvocationHandler.java
make[1]: Leaving directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/model/authorize'
cd main/model/logic; make
make[1]: Entering directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/model/logic'
javac -d ./../../../out -classpath ./../../../out Marketplace.java
javac -d ./../../../out -classpath ./../../../out AdminMarketplace.java
javac -d ./../../../out -classpath ./../../../out CustomerMarketplace.java
javac -d ./../../../out -classpath ./../../../out MarketplaceImpl.java
javac -d ./../../../out -classpath ./../../../out AdminMarketplaceImpl.java
javac -d ./../../../out -classpath ./../../../out CustomerMarketplaceImpl.java
javac -d ./../../../out -classpath ./../../../out MarketplaceServer.java
make[1]: Leaving directory `/home/pandey/v1/csci50700_spring2017_marketplace/main/model/logic'
```

- **Running Registry, Server & Database (Machine: in-csci-rrpc01.cs.iupui.edu - 10.234.136.55):**

```
make -f makefileServer run-registry
make[1]: Entering directory `/home/pandey/v1/csci50700_spring2017_marketplace'
rmiregistry 5000&
make -f makefileServer run-server
make[2]: Entering directory `/home/pandey/v1/csci50700_spring2017_marketplace'
java -Djava.security.policy=policy logic.MarketplaceServer
Creating a Marketplace Server!
MarketplaceServer: binding it to name: //in-csci-rrpc01.cs.iupui.edu:5000/MarketplaceServer
CustomerMarketplaceServer: binding it to name: //in-csci-rrpc01.cs.iupui.edu:5000/CustomerMarketplaceServer
AdminMarketplaceServer: binding it to name: //in-csci-rrpc01.cs.iupui.edu:5000/AdminMarketplaceServer
Marketplace Server Ready!
```

- **Running Client – Customer, Browsing Items (Machine: in-csci-rrpc03.cs.iupui.edu - 10.234.136.57):**

```
[pandey@in-csci-rrpc03 csci50700_spring2017_marketplace]$ make -f makefileClient
java -Djava.security.policy=policy entry/MarketplaceView
Please Enter Your Choice!
 Press 1 For Login
 Press 2 For Registering as a Customer
1
Choice Entered: 1
Please Enter Your Username and Password!
Saurabh.6
customer6
Username: Saurabh.6      &       Password: *****
Page requested: CUSTOMER
Welcome: Saurabh
Menu Options:
 1. Browse Item
 2. Add Item To Cart
 3. Remove Item From Cart
 4. Purchase Item
 5. Exit
 Please select an option to continue.
1
Product Details:
Name: Nike Trainers             ID: P1                  Type: Footwear                  Description: Sports shoes               Quantity Available: 100
Price: $100.0
Name: Columbia WW               ID: P2                  Type: Jacket            Description: Winter wear                Quantity Available: 75
Price: $135.0
Name: iPhone 7S                 ID: P3                  Type: Electronic                Description: Apple smartphone           Quantity Available: 200
Price: $800.0
Name: Sony cybershot            ID: P4                  Type: Electronic                Description: Digital camera             Quantity Available: 50
Price: $250.0
Menu Options:
 1. Browse Item
 2. Add Item To Cart
 3. Remove Item From Cart
 4. Purchase Item
 5. Exit
 Please select an option to continue.
```

- **Running Client – Administrator, Adding Items in System (Machine: in-csci-rrpc03.cs.iupui.edu - 10.234.136.57):**

```
[pandey@in-csci-rrpc03 csci50700_spring2017_marketplace]$ make -f makefileClient
java -Djava.security.policy=policy entry/MarketplaceView
Please Enter Your Choice!
 Press 1 For Login
 Press 2 For Registering as a Customer
1
Choice Entered: 1
Please Enter Your Username and Password!
Smith.1
admin1
Username: Smith.1       &       Password: *****
Page requested: ADMIN
Welcome: Smith
Menu Options:
 1. Browse Item
 2. Add Item In System
 3. Remove Item From System
 4. Update Item In System
 5. Exit
 Please select an option to continue.
2
Please Enter The Product Details:
 Enter ProductID:
P5
Enter Product Name:
Fastrack555
Enter Product Type:
Electronic
Enter Product Description:
Watch
Enter Product Quantity:
100
Enter Product Price:
300
Product Added Successfully!
Menu Options:
 1. Browse Item
 2. Add Item In System
 3. Remove Item From System
 4. Update Item In System
 5. Exit
 Please select an option to continue.
```

6. References:

- Frank Buschmann, Kevlin Henney, Douglas C. Schmidt, “Pattern Oriented Software Architecture”. Vol. 4.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vissides, “Design Patterns”.
- Lecture Slides (20 – 25) of Dr. Ryan Rybarczyk
- http://www.tomaszezula.com/2014/08/30/concurrency-patterns-monitor-object/
- https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
- http://javarevisited.blogspot.com/2015/01/how-to-use-future-and-futuretask-in-Java.html
- http://www.cs.wustl.edu/~schmidt/PDF/ScopedLocking.pdf
- https://www.tutorialspoint.com/jdbc/index.htm