Online Marketplace Application

By

Enas Alikhashashneh

ealikhas@iupui.edu

Department of Computer information Science

IUPUI

Dr. Rayan Rybarczyk

CSCI 50700 Object Oriented Design and Programming

Spring 2016

<u>Content Table</u>

1. Introduction

The online Marketplace application is a software that was created by using several software design patterns and a framework for the distributed computing systems. A couple of design decisions were decided in order to produce software with high quality, reusability, scalability, reliability, and maintainability through implementing a variety of software design patterns such as: Model-View-Controller, Front Controller, Authorization, Proxy, Reflection, Synchronization, and Concurrency patterns.

In this application one of the most popular Java API, which is the Remote Method Invocation (RMI); was used in order to create a distributed system through creating three basic components: Server, Remote Interface, and the Client, who is either Customer or Administrator. Where the communication between the client and server is done automatically over the Stub (in the Client side) and Skeleton (in the Server side) which are generated automatically. The remainder of this report is organized as follows: section 2 discusses the requirements, section 3 presents the development iteration, and section 4 concludes this report.

2. Requirements

Based on the requirements, which were provided, this application has to sell goods and services to the clients that are categorized into two different groups: Administrators and Customers. Each of the client must have a unique username and password in order to login to the application and perform their functions. The Administrators can add or remove other Administrator or Customer accounts, browse the item in the inventory, remove any item from the system, and update the information about system items. In the other hand, the Customers can browse the system item, add any item to their shopping cart, remove items from the shopping cart, browse the shopping cart items, and purchase the items. To satisfy all of the previous requirements an Online Marketplace application was developed over five main iterations.

3. Developing Iterations

3.1 Iteration # 1

3.1.1 Tasks

A set of tasks was achieved in this iteration as shown below:

- Build the skeleton framework for the online marketplace application.
- Translate the given requirements to a Domain Model for the online marketplace application.
- Create the Model-View-Controller for the online marketplace application.
- Use Java RMI to build the online marketplace application.

3.1.2 Implementation

In this iteration a Domain Model was created. This model is used to describe the problem domain through the model of the real world entities and their relationships, which are extracted from the requirements. In this assignment I used the class diagram to represent the domain model. The importance of the domain model is it gives the software developers a conceptual framework of the things in the problem space in order to allow the developer to focus more on the semantic through defining the structure and the state of the problem domain at any time. Based on the given requirements of the assignment and using the noun based techniques, a number of classes and their relationships are defined. These classes are represented in the fig. 1.

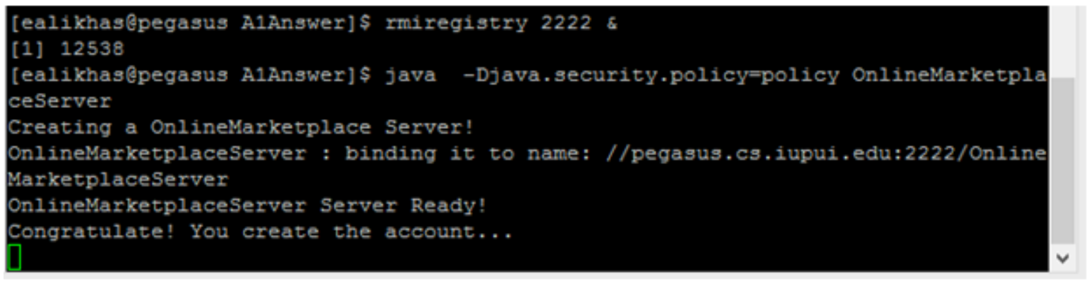
The users that will use the online marketplace application are classified into two major groups or roles: the administrator, who will add, remove, and update the available products/items in this application, also the administrator can create, and remove the user accounts. The second type of user is the customer, who plays the most important role by browsing the products/ items in order to add them into their shopping card to buy. When we look deeply we can observe the information that the customer and administrators need to fill up the form to create a new account or to buy the products such as: username, password, address, and full name allow us to define class is named "USER" as a super type of two other classes, which inherit "USER" attributes and methods, the "ADMINISTRATOR" and the "CUSTOMER". On the other hand, each product/item in this application has a specific attribute such as: description, type, price and quantity; thus I created an item class in order to store these attributes. Additionally, the item on this application is classified into two distant types: the Goods and Services; thus I created two different classes, which inherent the item class attributes and behavior. The last class is the shopping cart that is used in order to add any desired item/product to buy it later, there is a set of relationships between the classes listed following:

1) The relation between the "USER" class and the "CUSTOMER", "ADMINISTRATOR" is-a relationship.
2) The relation between the "CUSTOMER" and "ITEM" is an association where the customer can browse, select, and add the available items/products into the Shopping cart.
3) The relation between the "CUSTOMER" and the "SHOPPINGCART" is an association where the customer adds, modifies, and removes products/items from the "SHOPPINGCART".
4) The relation between "ADMINISTRATOR" and the "ITEM" is an association where the administrator can add, update, and remove the system items/products.

Additionally, the Java RMI was used in order to create a distributed communication between the client/server architecture. Where both of the users (customer and administrator), who act as clients will invoke the methods that implement on the server side, through a remote interface connect between the client and server. Fig.2 shows how the RMI is applied to the online marketplace application.

Lastly, the Model-View-Controller pattern was applied. Where the Client side is represented by building two layers: the view layer; which is written as a set of options and the users can select one of them in order to perform specific functions. The client controller layer, which looks up for the remote object in the registry, and invokes the remote object methods. The server side will contain both of the controller and the model layers, where the function of the controller is to create a remote object to connect the model with the RMI interface and then with the client controller, on the other hand the model layer implements the RMI interface in order to support the online marketplace functionality such as adding items or products to the system.

This design pattern allows the developer to isolate the view changes and reduce their effects on the model layer. Moreover, when the model layer is changed such as: changing the implementation of one of the methods by adding print statement, the view layer will not be change. Thus, in this way we increase the cohesion between the layers and decries the coupling between the layers. Fig.3 and 4 represent some of sample running for this iteration:

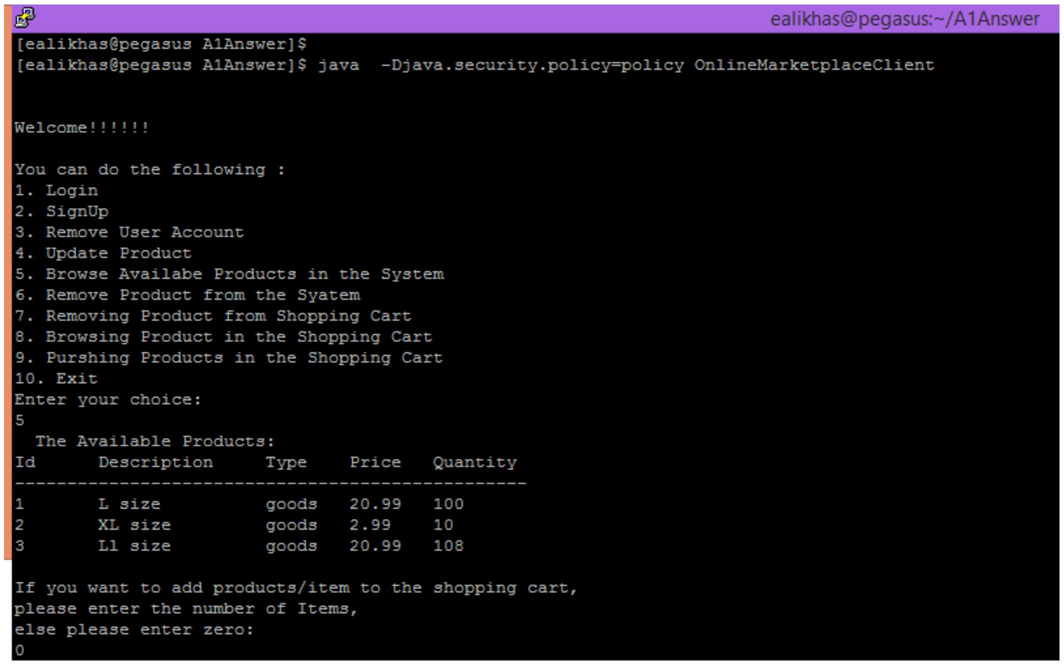


```
[ealikhas@pegasus A1Answer]$ rmiregistry 2222 &
[1] 12538
[ealikhas@pegasus A1Answer]$ java  -Djava.security.policy=policy OnlineMarketpla
ceServer
Creating a OnlineMarketplace Server!
OnlineMarketplaceServer : binding it to name: //pegasus.cs.iupui.edu:2222/Online
MarketplaceServer
OnlineMarketplaceServer Server Ready!
Congratulate! You create the account...
```

Fig. 3: Server side

3.1.3 <u>Feedback</u>

From this iteration I learned how create a distribute framework using Java RMI, and increaser the cohesion and reduce the coupling of the application components in order to increase the maintainability, reusability, and others. Thus; for these reasons we applied MVC patterns.



```
[ealikhas@pegasus A1Answer]$
[ealikhas@pegasus A1Answer]$ java  -Djava.security.policy=policy OnlineMarketplaceClient


Welcome!!!!!!

You can do the following :
1. Login
2. SignUp
3. Remove User Account
4. Update Product
5. Browse Availabe Products in the System
6. Remove Product from the Syatem
7. Removing Product from Shopping Cart
8. Browsing Product in the Shopping Cart
9. Purshing Products in the Shopping Cart
10. Exit
Enter your choice:
5
  The Available Products:
Id      Description     Type    Price   Quantity
-------------------------------------------------
1       L size          goods   20.99   100
2       XL size         goods   2.99    10
3       L1 size         goods   20.99   108

If you want to add products/item to the shopping cart,
please enter the number of Items,
else please enter zero:
0
```

Fig.4: Client View

3.2 <u>Iteration # 2</u>

3.2.1 <u>Tasks</u>

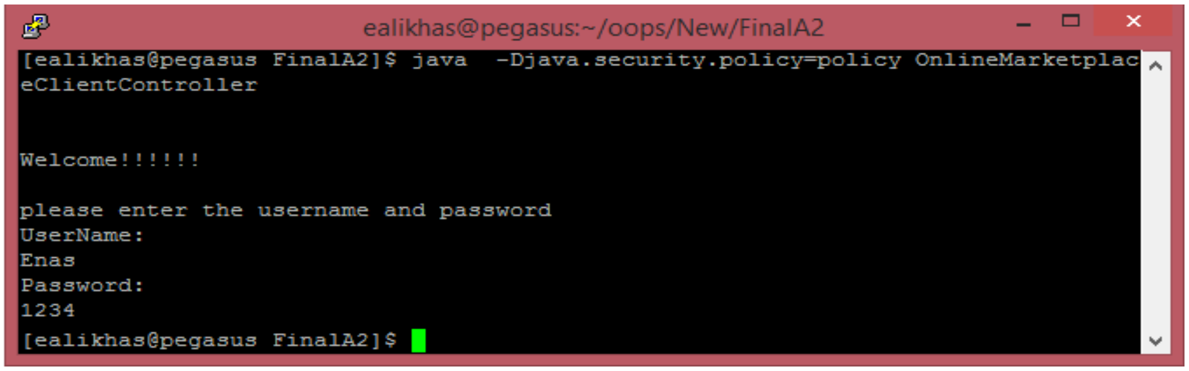
A set of tasks were achieved in this iteration as shown below:

- Implement the Front Controller pattern.
- Implement the Authorization pattern by creating a Session object that can be serialized and sent across the network via Java RMI.

3.2.2 <u>Implementation</u>

In this iteration the users have to enter the login information such as: the user name and password in order to perform their functions. Thus; I used the front controller pattern, which allocates between the client view and client controller, in order to collect this information from the login view, parse it, and send a request to the client controller in order to check the authorization for the user. In this stage, the client controller will create a remote object to invoke the methods in the server side, in other word invokes the login method, which checks the user name and password if it exist or not. If the user name and password valid the login method creates and returns back the session object to the client controller, which sends it to the front controller.

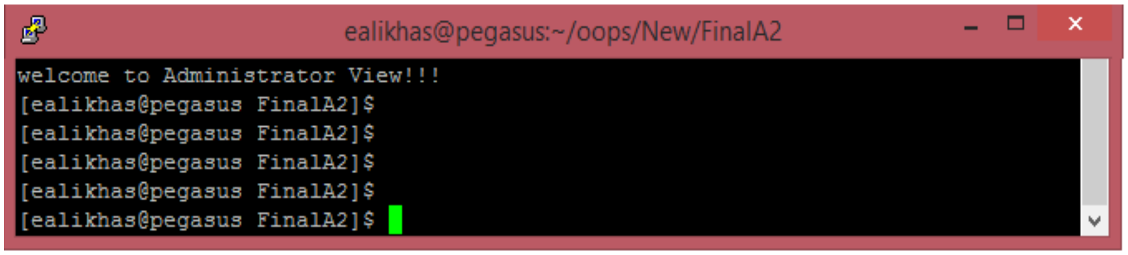
The front controller checks if the object is null or not, where the null object denotes the user has not authorized to browse, add, remove, and buy the products or items from the Online Marketplace application; thus the user have to sign up either administrator account or customer account after got a notification from the front controller in order to perform the actions and events of the application. On the other hand, if the session object is not null this mean the user has authorization and the server create a special session object for his/ her, in this situation the front controller will delegate the dispatcher to decide which flow will be, either administrator or customer flow based on the role data member of the session object. When the session role is administrator the dispatcher will invoke the administrator control in order to show up the administrator view for the user and if the session role is customer the dispatcher will invoke the customer control in order to show up the customer view for the user. There are two related patterns to the front controller pattern: the command pattern, which implement the relation between the front controller and the dispatcher. Another pattern is the strategy pattern, which is represented the relation between the dispatcher and either the administrator or customer flow. Fig.5 represent a sample for login view after the user entered his/her username and password.



```
[ealikhas@pegasus FinalA2]$ java  -Djava.security.policy=policy OnlineMarketplac
eClientController


Welcome!!!!!!

please enter the username and password
UserName:
Enas
Password:
1234
[ealikhas@pegasus FinalA2]$
```

Fig 5. Sample for the Login View

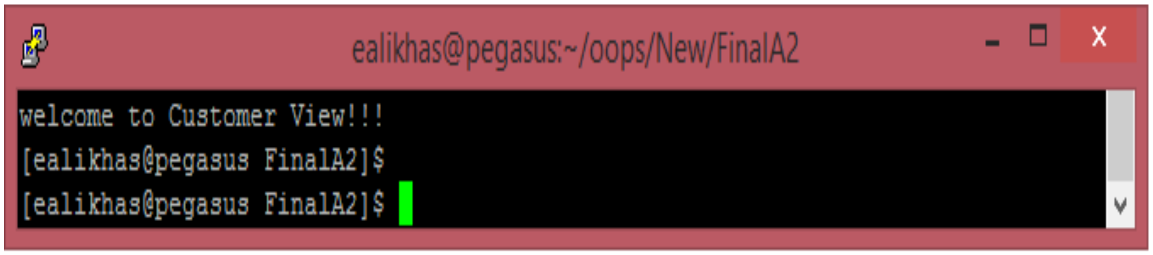
Fig.6 represents the administrator view after the administrator entered the username and password in order to add a new item to the application, remove an item, remove customer or administrator accounts, and update the information of the item.



```
welcome to Administrator View!!!
[ealikhas@pegasus FinalA2]$
[ealikhas@pegasus FinalA2]$
[ealikhas@pegasus FinalA2]$
[ealikhas@pegasus FinalA2]$
[ealikhas@pegasus FinalA2]$
```

Fig 6. Sample for the Administrator View

Fig.7 represents the customer view after the customer entered the username and password in order to add a new item to the shopping cart, remove item from shopping cart, buy the items in the shopping cart, and brows the available item in the application.



```
welcome to Customer View!!!
[ealikhas@pegasus FinalA2]$
[ealikhas@pegasus FinalA2]$
```

Fig 7. Sample for the Customer View

3.2.3 Feedback

From this iteration I learned based on the authorized information of the users I will create a session object which will be used latter to determine which view will be display either the Customer or Administrator view. Another option for design the application in this iteration is creating two remote object classes. The first one is obtained via Naming. Lookup(); it is a singleton; and it contains a login() method, which returns the second remote object, which is not a singleton, not registered in the Registry, and is created anew for every return value. This object contains all the Online Marketplace methods for this user. It probably also implements the Unreferenced interface, which is an interface allows the object to receive notification that there are no users holding remote references to it, so it can detect a dead user. Because it exists once per user, it can hold user state, and because it can only be obtained by a successful login step it increase the security of this application.

3.3 Iteration # 3

3.3.1 Tasks

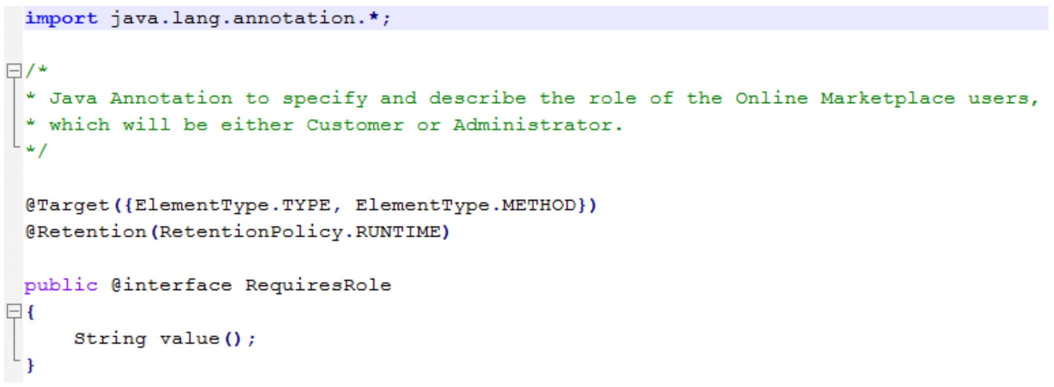
A set of tasks were achieved in this iteration as shown below:

- Implement Role-based Access control (RBAC) approach.
- Implement Java Annotations.
- Implement and explore the Proxy pattern.
- Implement the Reflection pattern.

3.3.2 Implementation

In this iteration the implementation of the Authorization pattern was expanded by creating a Role-based Access control (RBAC) approach by using the Java Annotation, Proxy pattern, and reflection pattern. Where the Java support last two patterns.

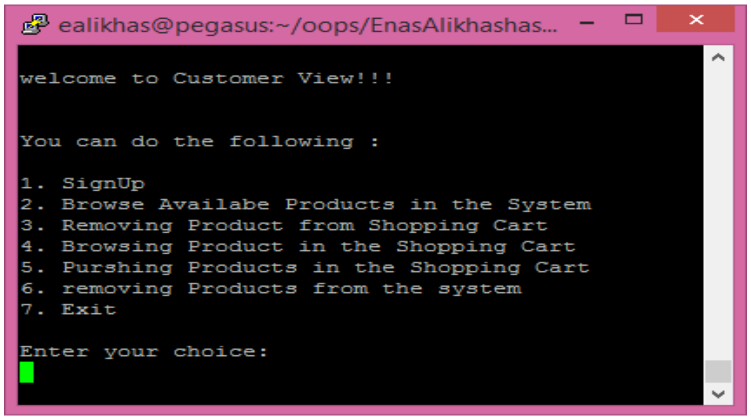
Java Annotation is a form of a metadata that is used in order to provide more information or data about the program. The annotation is not considered as a part of the program thus it is not effect on the execution of the code. In this application I used the annotation in order to specify and describe the role of the Online Marketplace application clients. Which will be either Administrator or Customer. The RequiredRole annotation, which is shown in the fig.8, is applied at the method level during the runtime. Where the value stores the client role.



```
import java.lang.annotation.*;

/*
* Java Annotation to specify and describe the role of the Online Marketplace users,
* which will be either Customer or Administrator.
*/

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)

public @interface RequiresRole
{
    String value();
}
```

Fig.8 Java Annotation

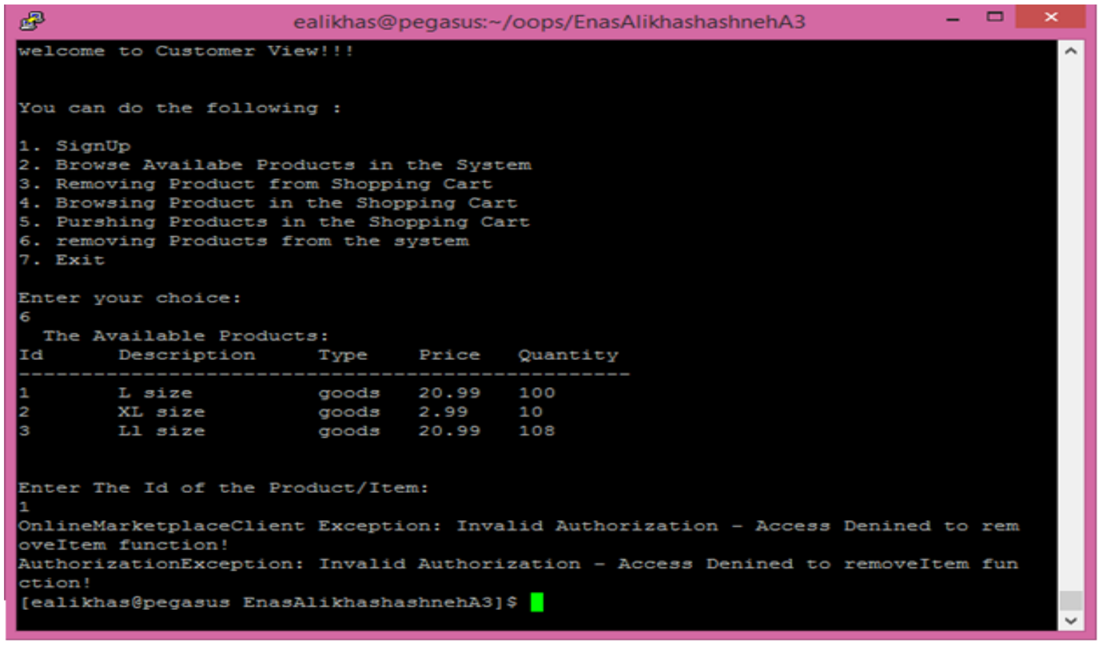
The RMI interface used this annotation in order to determine who can call each method. On the other hand, if the administrator attempt to play the customer role through invokes unauthorized methods such as: purchase items from the shopping cart. The application before invokes and executes the purchase method it will check the role of the client's session with the role, which is specified in the remote interface, if they match the method will be executed and return a value if it is exist, else the application will display the exception message, which is specified in the Authorization Exception. Fig.9 represents the customer view.



```
ealikhas@pegasus:~/oops/EnasAlikhashas...

welcome to Customer View!!!


You can do the following :

1. SignUp
2. Browse Availabe Products in the System
3. Removing Product from Shopping Cart
4. Browsing Product in the Shopping Cart
5. Purshing Products in the Shopping Cart
6. removing Products from the system
7. Exit

Enter your choice:
```

Fig 9. Sample for Customer View

In fig 10. If the customer attempts to remove an item or product from the system, which is allowed for the administrator, the application will display exception message in order to notify the customer he can't access this method and he doesn't have a right to remove any item from the system. Fig 10. Represent the invalid authorization.



```
ealikhas@pegasus:~/oops/EnasAlikhashashnehA3
welcome to Customer View!!!


You can do the following :

1. SignUp
2. Browse Availabe Products in the System
3. Removing Product from Shopping Cart
4. Browsing Product in the Shopping Cart
5. Purshing Products in the Shopping Cart
6. removing Products from the system
7. Exit

Enter your choice:
6
  The Available Products:
Id       Description      Type     Price    Quantity
-------------------------------------------------
1        L size           goods    20.99    100
2        XL size          goods    2.99     10
3        L1 size          goods    20.99    108


Enter The Id of the Product/Item:
1
OnlineMarketplaceClient Exception: Invalid Authorization - Access Denined to rem
oveItem function!
AuthorizationException: Invalid Authorization - Access Denined to removeItem fun
ction!
[ealikhas@pegasus EnasAlikhashashnehA3]$
```

Fig 10. Sample of the Customer View

3.3.3 <u>Feedback</u>

In this iteration based on the authorized information of the users I will create a session object which will be used latter to determine which view will be display either the Customer or Administrator view. Another option for design the application in this iteration is creating two remote object classes. The first one is obtained via Naming. Lookup(); it is a singleton; and it contains a login() method, which returns the second remote object, which is not a singleton, not registered in the Registry, and is created anew for every return value. This object contains all the Online Marketplace methods for this user. It probably also implements the Unreferenced interface, which is an interface allows the object to receive notification that there are no users holding remote references to it, so it can detect a dead user. Because it exists once per user, it can hold user state, and because it can only be obtained by a successful login step it increases the security of this application.

3.4 Iteration #4

3.4.1 Tasks

A set of tasks were achieved in this iteration as shown below:

- Examine the impact the concurrency in Online Marketplace application
- Run the clients on a different machines.
- Implement the purchase item and add item functions.

3.4.2 Implementation

In the previous iteration, the Online Marketplace Application create only one Server and one Client. In this iteration there are multiple Clients, which attempt to invoke same remote object methods concurrently and this will make the Online Marketplace Application works as a multithread application. In multithread application the threads usually communicate by sharing access to fields and the objects reference fields refer to [1]. In such situation a two kind of problems will be occur: thread interference and memory consistency errors.

One of the challenge that was happen in this iteration was the memory consistency errors for instance: when one administrator try to update item with quantity 10 and the system have 10 as quantity for this item. And if the customer concurrently try to add/browse the system items he will see the quantity of this item is 10 not 20 this will be happen because each client represent a spate thread. While if all clients work in same thread the customer will see 20 not 10. To solve this problem we have to create happen-before relationships and one of the action to do that is using the synchronization. One of possible solution is to declare all the remote object methods as synchronized ones to allow mutual exclusive access of critical section to two threads. But the methods synchronized has two effects:

- First, 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 (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

This strategy is effective, but can present problems with liveness such as: deadlock, starvation, and livelock. Based on the java documentation there is no guarantee with respect to mapping the remote object invocations to threads. In other words after the remote object got or exported, the RMI create a thread, and start listen to any client request. Later on when a client does a RMI call, the listening thread got accept (), and spawn a new thread to process this RMI call. So each client call match a new Server side thread. RMI does nothing to make these Server side threads to be thread-safe. So if there are multiple clients making remote calls at the same time, the Remote object methods has to take care of the thread-safe (synchronization). And as I mentioned above from the sample running we can note that each client has a unique thread.

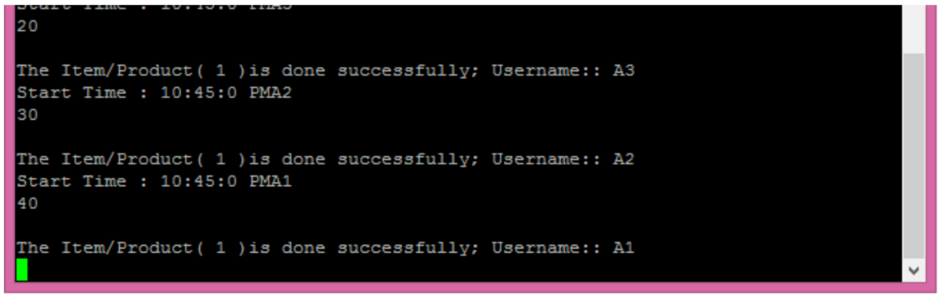
In this iteration the concurrency impact on this application was examined. In order to do that the clients, which were run on different JVM, sent a concurrent requests, which are represented as invocation either same method or different one. These requests need to overlap and be handled concurrently, in the reality each of these request takes on average two seconds to fulfill (from receiving the request at the server to processing it through the application for sending back the response). The flood of requests in the Online Applications can cause a several of problems such as: deadlock and source starvation this may happen because there are many clients shared same resources (the item table and the users table).

One of the common examples of the concurrency in the Online Marketplace Application, when there exist two Clients such as: Customer and Administrator simultaneously attempt to add and buy the same item in the store. This situation could produce undesirable result and may cause the data in the application be inconsistency. Another example, in the Online Marketplace Application is when one of the Customers make a request on the remote server to add a set of items to his/her shopping cart; if the add item to shopping method cart invoke the browser items method, and both of them are synchronized ones; this means all the other Customers will be blocked from adding and browsing the items in the store until the first Customer or thread finish. From the previous examples we can infirm that the requests from distinct Clients (executing in different JVMs) need to coordinate; in other words, we have to define a way to control the individual threads, to coordinate thread activities, to provide thread safety mechanism, and to assign priorities to threads.

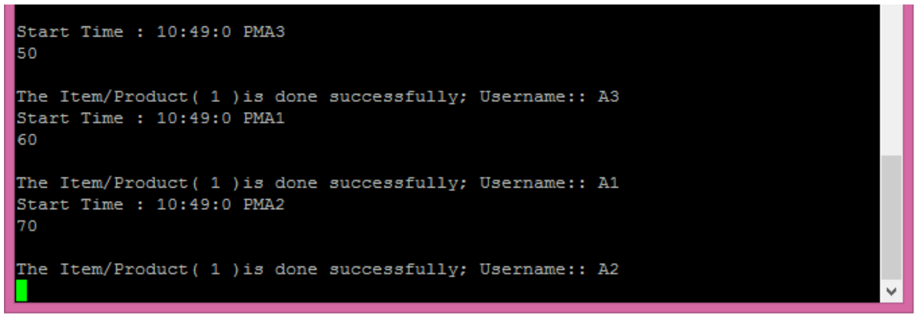
The following running sample reflect how the RMI treats the concurrent requests; where the methods in the remote object are not method synchronized. Also, the Admin and Customer view was modified in order to speed up the testing process for instance in admin view instead the customer enter the information of the item, which he/she want to add it into the shopping cart, the information inserted in the code. In order to generate concurrent requests from different clients at same time, a script of code was written to send all the client requests at same time. To perform this I used specific round method in order to round the invoke method time to the next second.

TEST CASE # 1:

3 admins add same item with same quantity at same time (successfully), but if we repeat the process a random order for the requests will be taken.



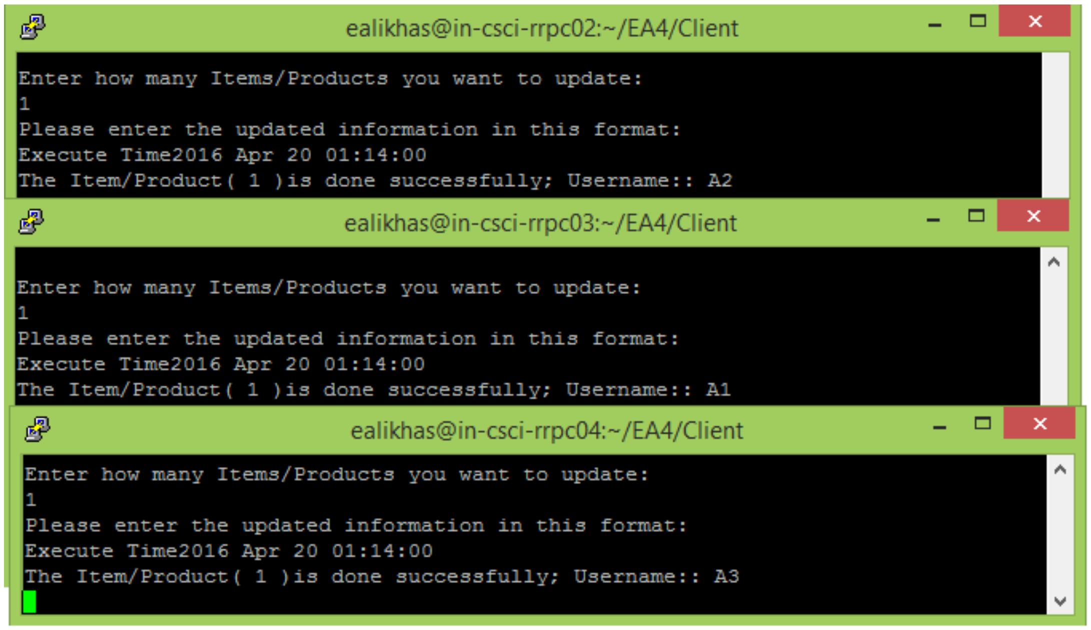
```
20

The Item/Product( 1 )is done successfully; Username:: A3
Start Time : 10:45:0 PMA2
30

The Item/Product( 1 )is done successfully; Username:: A2
Start Time : 10:45:0 PMA1
40

The Item/Product( 1 )is done successfully; Username:: A1
```



```
Start Time : 10:49:0 PMA3
50

The Item/Product( 1 )is done successfully; Username:: A3
Start Time : 10:49:0 PMA1
60

The Item/Product( 1 )is done successfully; Username:: A1
Start Time : 10:49:0 PMA2
70

The Item/Product( 1 )is done successfully; Username:: A2
```

TEST CASE # 2:

3 admins update same item with same quantity at same time. Successfully updated but the order of executing the concurrent requests randomly.



ealikhas@in-csci-rrpc02:~/EA4/Client

```
Enter how many Items/Products you want to update:
1
Please enter the updated information in this format:
Execute Time2016 Apr 20 01:14:00
The Item/Product( 1 )is done successfully; Username:: A2
```

ealikhas@in-csci-rrpc03:~/EA4/Client

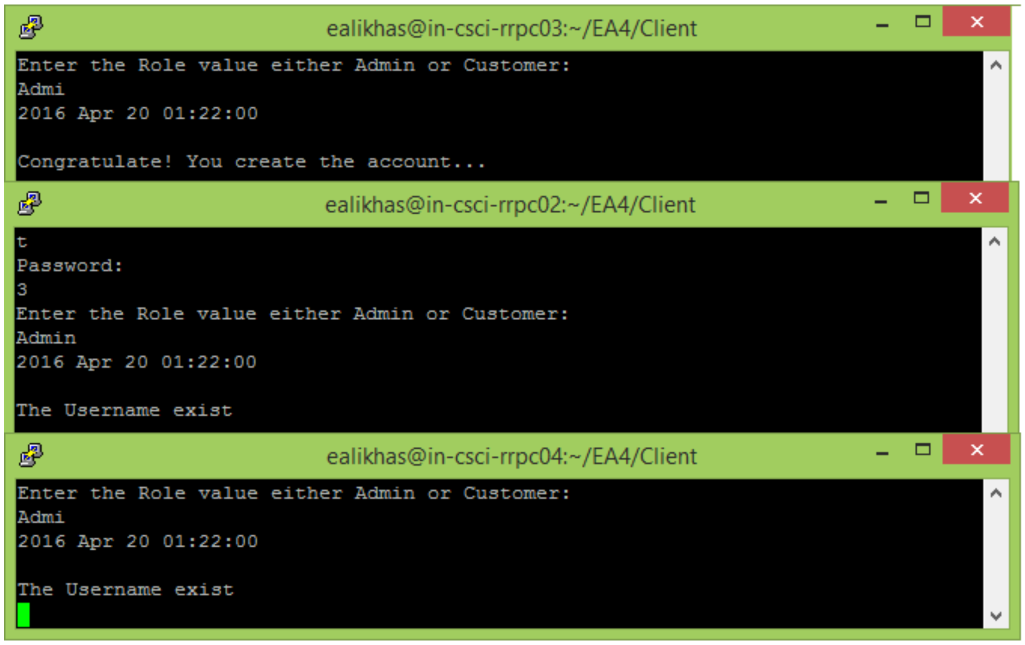
```
Enter how many Items/Products you want to update:
1
Please enter the updated information in this format:
Execute Time2016 Apr 20 01:14:00
The Item/Product( 1 )is done successfully; Username:: A1
```

ealikhas@in-csci-rrpc04:~/EA4/Client

```
Enter how many Items/Products you want to update:
1
Please enter the updated information in this format:
Execute Time2016 Apr 20 01:14:00
The Item/Product( 1 )is done successfully; Username:: A3
```

TEST CASE # 3:

3 admins signup new account with same username, password and same role at same time. Successfully Signup but the order of executing the concurrent requests randomly.



ealikhas@in-csci-rrpc03:~/EA4/Client

```
Enter the Role value either Admin or Customer:
Admi
2016 Apr 20 01:22:00

Congratulate! You create the account...
```

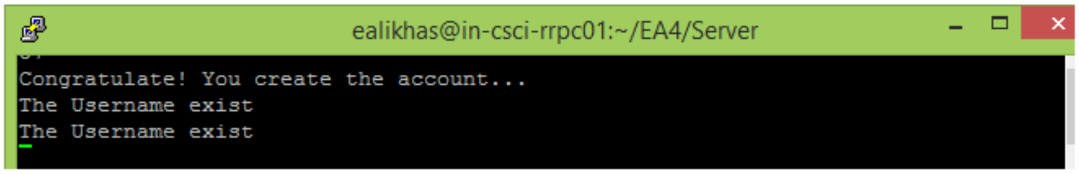
ealikhas@in-csci-rrpc02:~/EA4/Client

```
t
Password:
3
Enter the Role value either Admin or Customer:
Admin
2016 Apr 20 01:22:00

The Username exist
```

ealikhas@in-csci-rrpc04:~/EA4/Client

```
Enter the Role value either Admin or Customer:
Admi
2016 Apr 20 01:22:00

The Username exist
```

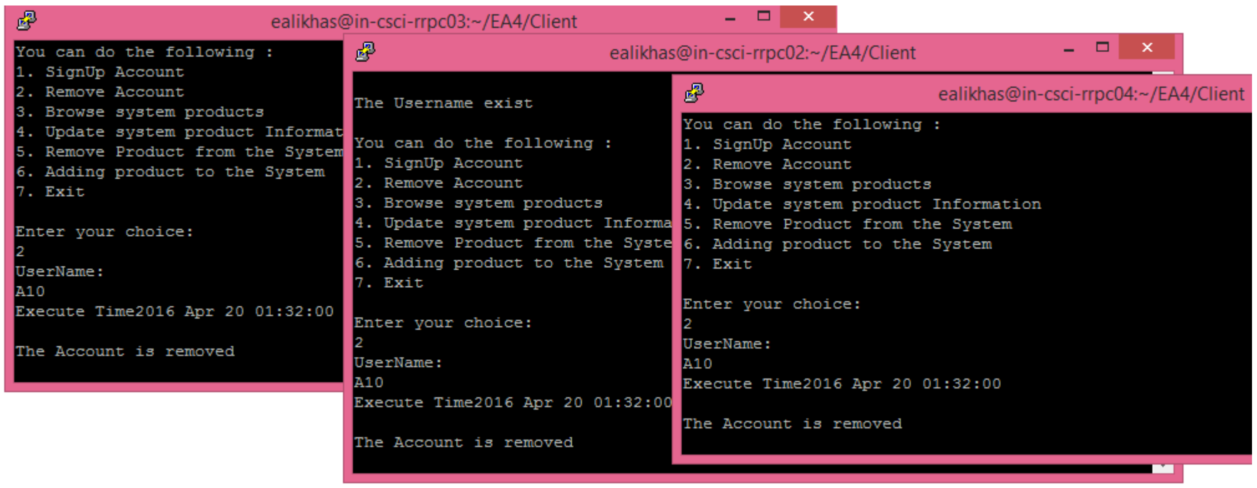
Server state after executing the concurrent requests.



```
Congratulate! You create the account...
The Username exist
The Username exist
```

TEST CASE # 4:

3 admins attempt to remove same account with same username at same time. The server sent message about successfully remove account but the order of executing the concurrent requests randomly. But when I try to sign in by using same username, I sign in successfully!!!!!

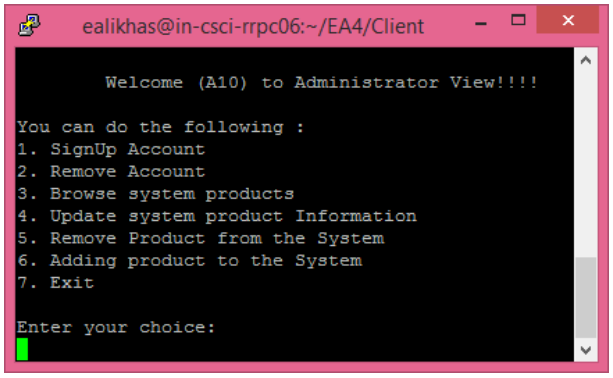


ealikhas@in-csci-rrpc03:~/EA4/Client
ealikhas@in-csci-rrpc02:~/EA4/Client
ealikhas@in-csci-rrpc04:~/EA4/Client

```
You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Informat
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
2
UserName:
A10
Execute Time2016 Apr 20 01:32:00

The Account is removed
```

```
The Username exist

You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Informa
5. Remove Product from the Syste
6. Adding product to the System
7. Exit

Enter your choice:
2
UserName:
A10
Execute Time2016 Apr 20 01:32:00

The Account is removed
```

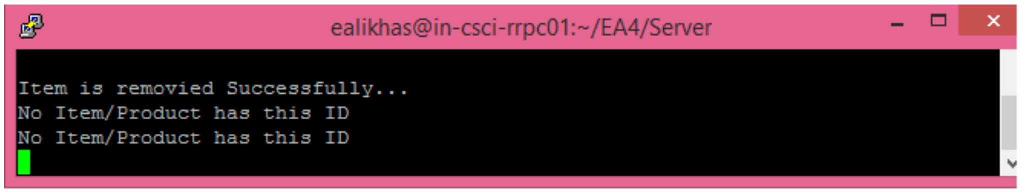
```
You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
2
UserName:
A10
Execute Time2016 Apr 20 01:32:00

The Account is removed
```



```
       Welcome (A10) to Administrator View!!!!

You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
```

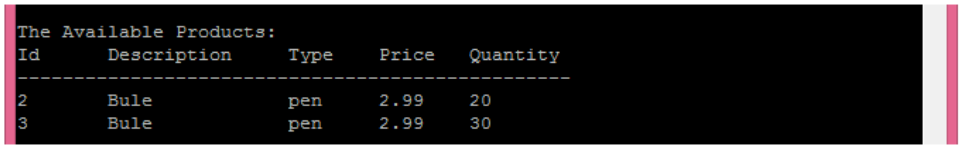
TEST CASE # 5:

3 admins attempt to remove same item from the system at same time. The server sent message about successfully remove item for one of the admin but the others will receive error message the item is not exist in the system. The order of executing the concurrent requests randomly. The following figure shows the server state.



```
Item is removed Successfully...
No Item/Product has this ID
No Item/Product has this ID
```

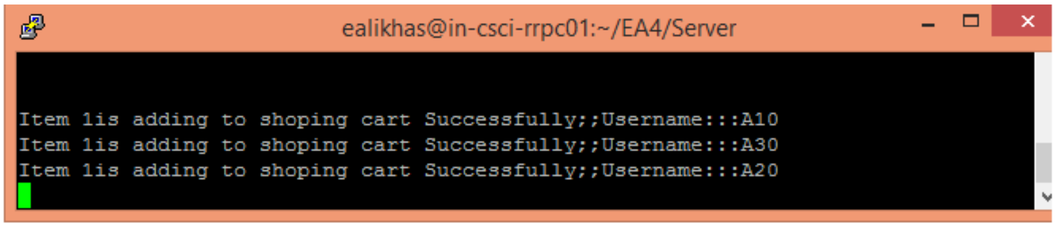
The 3 admins try to remove the item # 1 and as in the following figure the item #1 was successfully removed.



```
The Available Products:
Id      Description     Type    Price   Quantity
-------------------------------------------------
2       Bule            pen     2.99    20
3       Bule            pen     2.99    30
```

TEST CASE # 6:

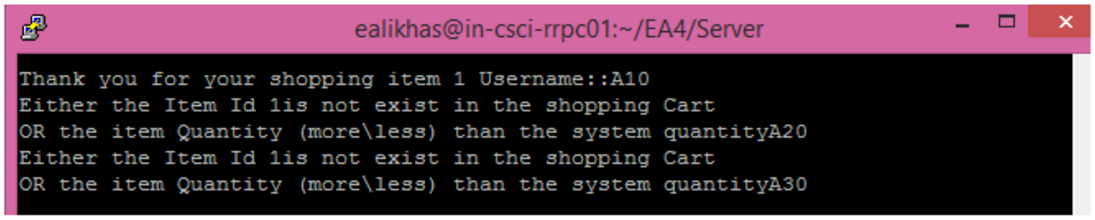
3 Customers add same item with same quantity at same time (successfully) to the shopping cart but if we repeat the process a random order for the requests will be taken. Note the quantity is equal to the item quantity in the system.



```
Item 1is adding to shoping cart Successfully;;Username:::A10
Item 1is adding to shoping cart Successfully;;Username:::A30
Item 1is adding to shoping cart Successfully;;Username:::A20
```

TEST CASE # 7:

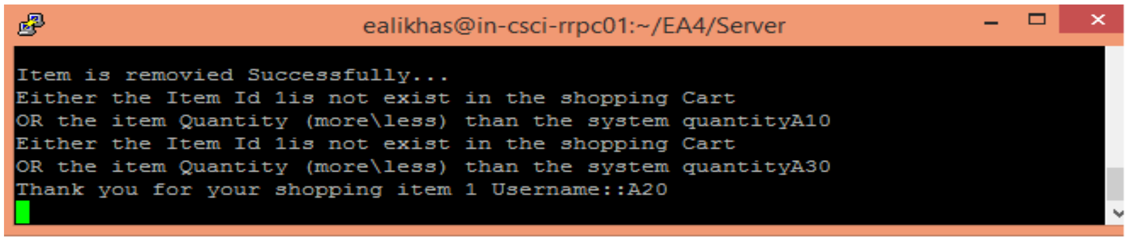
3 Customers purchase same item with different quantities, where the summation of these quantities will be larger than the availability quantity in the system, at same time. One of the customer will buy the item successfully while the others will receive a warning message about the item quantity is larger than the availability quantity in the system. If we repeat the process a random order for the requests will be taken. The quantity in the system will be updated successfully.



```
Thank you for your shopping item 1 Username::A10
Either the Item Id 1is not exist in the shopping Cart
OR the item Quantity (more\less) than the system quantityA20
Either the Item Id 1is not exist in the shopping Cart
OR the item Quantity (more\less) than the system quantityA30
```

TEST CASE # 8:

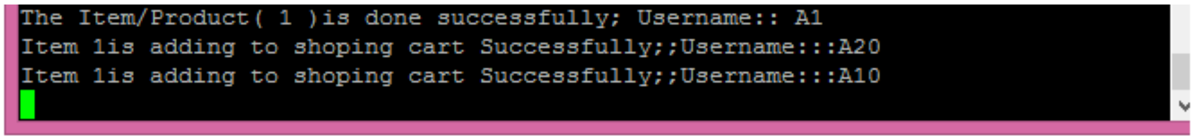
3 Customers purchase same item with same quantity at same time from the shopping cart one of the customer will buy the item successfully while the others will receive a warning message about the item is not exist or the requested quantity is larger than the availability quantity in the system . If we repeat the process a random order for the requests will be taken. Note the quantity is equal to the item quantity in the system.



```
Item is removied Successfully...
Either the Item Id 1is not exist in the shopping Cart
OR the item Quantity (more\less) than the system quantityA10
Either the Item Id 1is not exist in the shopping Cart
OR the item Quantity (more\less) than the system quantityA30
Thank you for your shopping item 1 Username::A20
```

TEST CASE # 9:

2 Customers A20 and A10 attempt to add same item with same quantity at same time. Concurrently admin A1 try to update the quantity of same item to the system. All the Clients added the item successfully without any problems as shown in the following figures. In this situation the Application works correctly, because in this situation the server executed the requests sequentially and select the first one randomly. And because the first one was the Administrator request the other clients request done correctly, but if one of Customer request select first the system will raise warning message for the other customer.



```
The Item/Product( 1 )is done successfully; Username:: A1
Item 1is adding to shoping cart Successfully;;Username:::A20
Item 1is adding to shoping cart Successfully;;Username:::A10
```

3.4.3 Feedback

In the multithread applications performing and executing a set of concurrent requests without using any thread-safety approaches such as: synchronization in order to schedule the access to the critical sections in the code will change the state of the object or the shared resources and may cause two kind of problems: thread interference and memory consistency errors. Thus, to solve this problem a set of synchronization patterns can be used as we will see in the next iteration.

3.5 Iteration #5

3.5.1 Tasks

A set of tasks were achieved in this iteration as shown below:

- Create Database for the Online Marketplace application.
- Connect this Database to the server side.
- Implement the Monitor object pattern.
- Implement the Guarded Suspension pattern.
- Implement the Scoped Locking pattern.
- Implement the Future pattern.
- Implement the Thread-safe pattern.
- Implement the browse item, login, update item, remove item, remove user, purchase item, and add item methods.

3.3.2 Implementation

This section contains a set of different subsections, which each one implements a specific task.

3.3.2.1 DataBase

This section illustrates the schema of the database, which is used in this application in order to hold the important information about the clients and the items, and how to retrieve, update, insert, and delete the information from the database.

3.3.2.1.1 Create the DataBase

Using the database in the Online Marketplace Application is necessary to hold the details about the application users and the products, it's likely to include the user's login information and their roles in this application. In addition to this, specialist information is stored to describe the products in more details such as: the type of product and the available quantity in the inventory. For all the above reasons the MySQL is used to create the application database. The following figure displays the database for the Online Marketplace Application, which contains three tables (Users, item, and ShoppingCart).

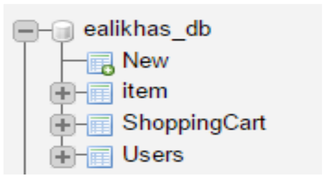


Fig.11 The DB

The following figure shows the item table, which contains the information about each item in the inventory such as: ID, Description, Type, Price, and Quantity. In this table the ID field represents the primary key because each item must have a unique ID to identify this item, also this primary key will speed up the queries in the item table such as: retrive the item price, update, and delete specific item.

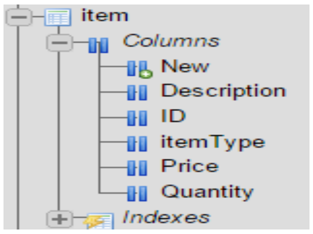
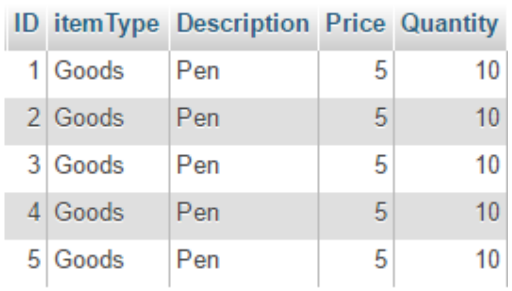


Fig.12 Item table



ID	itemType	Description	Price	Quantity
1	Goods	Pen	5	10
2	Goods	Pen	5	10
3	Goods	Pen	5	10
4	Goods	Pen	5	10
5	Goods	Pen	5	10

Fig.13 Sample for item records

The clients in this application are categorized into two different groups: Admininstrators and Customers, and each of them has a unique functionality in this aplication. For instance, the administrator can add a new item or product to the inventory while the customer can't. In order to store the informaation for these clients the database contains a Users table as shown in the following figure. This table contains the information about each client who uses this application such as: username, passphrase, and role. In this table the username field represents the primary key because each client must have a unique username in order to login sucessfully to the application and perform his/her functionality without intersect between the adminstrator functionality with the customer functionality. Also this primary key will speed up the queries in Users table such as: retrive the user role, update, and delete specific user information. Additionally, the role field plays an important role in this application where based on the value of this field the functionality for each client will be assigned.

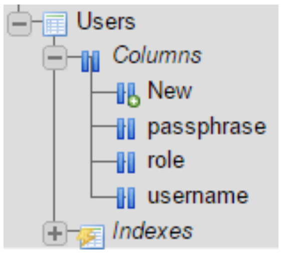
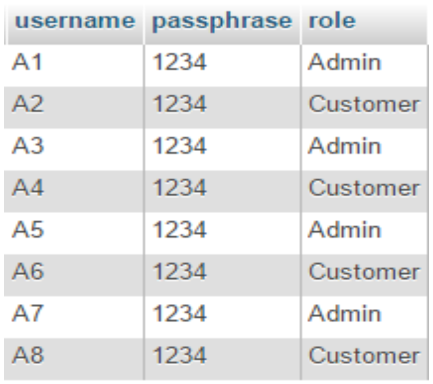


Fig.14 Users table



username	passphrase	role
A1	1234	Admin
A2	1234	Customer
A3	1234	Admin
A4	1234	Customer
A5	1234	Admin
A6	1234	Customer
A7	1234	Admin
A8	1234	Customer

Fig.15 Sample for Users records

I created an additional table called Shopping Cart in order to allow the customer to store the item information such as the item ID and Quantity, which the customer wants to buy from the application. When the customer adds an item to shopping cart with specific quantity this item will be stored in this table and the customer also can delete any item from the shopping cart. After the customer buys all the items the records in this table will be removed automatically. The following figure represents the Shopping Cart table.

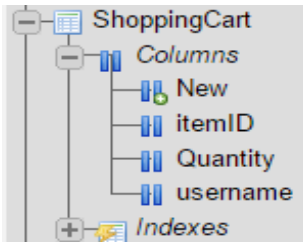
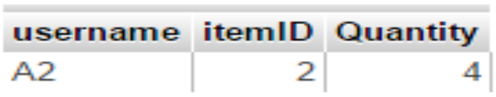


Fig.16 ShoppingCart table



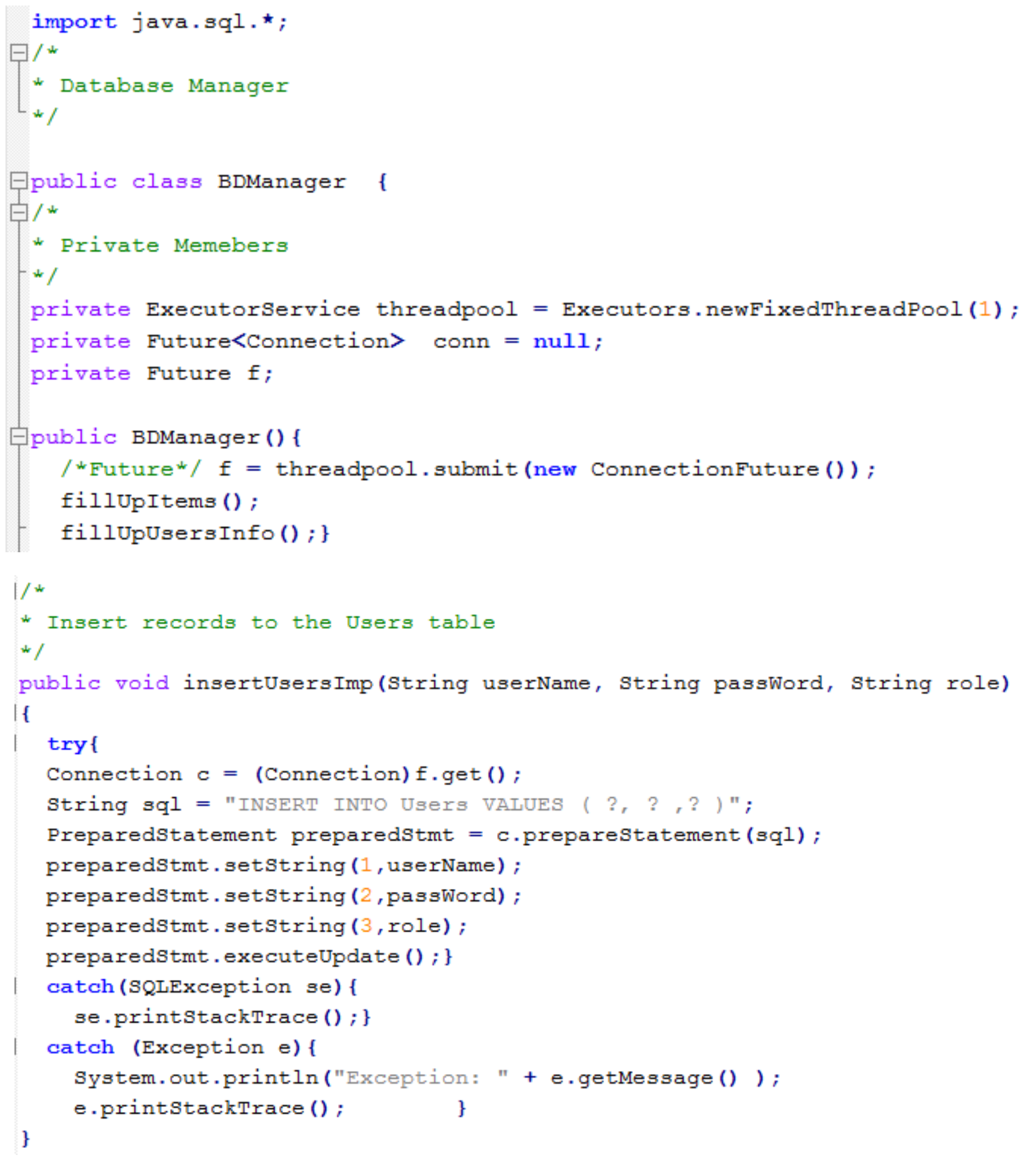
username	itemID	Quantity
A2	2	4

Fig.17 Sample for ShoppingCart records

<u>**DBManager Class**</u>

In this iteration a new class DBManager was created in order to perform any database operations such as insert, update, delete, retrieve data from and to the Online Marketplace application. This class contains a set of methods such as: insertUsersImpl, insertItemImpl, deleteItemImpl, updateItemDescriptionImpl, and updateItemTypeImpl. Because each of these methods represent a critical section where each of them either updates or deletes records in the database we have to ensure that only one method has to be executed at a time to keep the data in the database consistent. For this reason only the Online Marketplace application Model object has to invoke these methods. Additionally, the model methods were synchronized thus when they invoke any method from the DBManager class it will propagate the lock of the caller method. In this way we protect our data in the database.

This class is related with another new one which is called ConnectionFuture, which implemented the Future pattern. The DBManager submits a task for the ConnectionFuture class in order to perform the connection to the database, which is a long run task, through returning a lightweight Connection object. This object is a promise that the Connection will be available in the future. We don’t know when, but once it’s there, we will be able to retrieve it using the get ( ) method. Additionally, when a new DBManager instance is created the DBManager constructor invokes two private methods (fillUpItems() and fillUpUsersInfo()) which insert a set of initial records in the database such as: item information and users information in order to test this application through allowing clients to login into the server using this information. The following source code represents a part of the DBManager.



```
import java.sql.*;
/*
* Database Manager
*/

public class BDManager  {
/*
* Private Memebers
*/
private ExecutorService threadpool = Executors.newFixedThreadPool(1);
private Future<Connection>  conn = null;
private Future f;

public BDManager(){
  /*Future*/ f = threadpool.submit(new ConnectionFuture());
  fillUpItems();
  fillUpUsersInfo();}

/*
* Insert records to the Users table
*/
public void insertUsersImp(String userName, String passWord, String role)
{
  try{
  Connection c = (Connection)f.get();
  String sql = "INSERT INTO Users VALUES ( ?, ? ,? )";
  PreparedStatement preparedStmt = c.prepareStatement(sql);
  preparedStmt.setString(1,userName);
  preparedStmt.setString(2,passWord);
  preparedStmt.setString(3,role);
  preparedStmt.executeUpdate();}
  catch(SQLException se){
    se.printStackTrace();}
  catch (Exception e){
    System.out.println("Exception: " + e.getMessage() );
    e.printStackTrace();         }
}
```

Fig.18 Sample code for DBManager.

<u>Future Pattern</u>

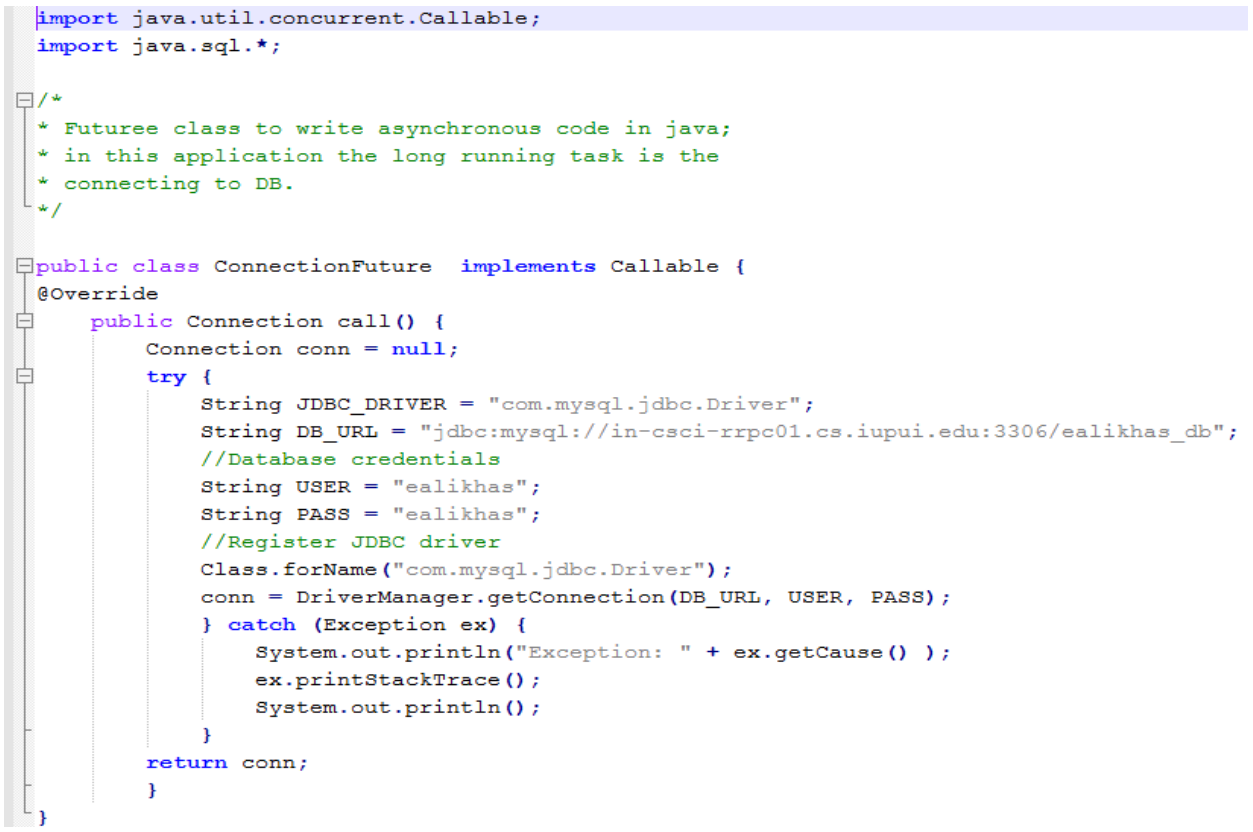
This pattern is also called promise or delays pattern, which is important for the asynchronous, event-driven, parallel, and scalable system. The Java programming language implements this pattern by using the Future<T> as a parametric interface and FutureTask as an implementation or RunnableFuture. Both of them are available in Java.util.concurrent package.

One situation where we can apply this pattern is if we have two methods A and B, where the B performs a long time process such as: downloading files or creating a connection to the database. And method A invokes B in order to reduce the latency instead B returns immediately, it returns a lightweight Future object, which is a promise that the object will be available in the future. Thus; the method A can perform some independent processing in the meantime while waiting for results. In this way we increase and optimize the performance of the software.

Java provides a set of basic methods that can be used for this interface:

1. **Future.get()** is the most important method. It blocks and waits until the promised result is available. So if we really need the object, we just call get () and wait.
2. **Future.isDone()** is used to peek on the Future and continue if the result is not yet available. It returns a Boolean value the true represents the object is available while the false represents wait.
3. **Future.cancle(true)** Cancelling future.

In the Online Marketplace application, a new class named **ConnectFuture** is declared in order to implement this pattern. This class attempts to create a connection to the database, which is considered as a long running task. The following code represents the **ConnectFuture**.

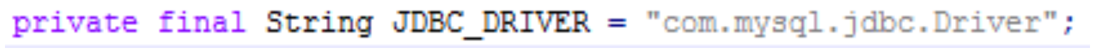


```
import java.util.concurrent.Callable;
import java.sql.*;

/*
* Futuree class to write asynchronous code in java;
* in this application the long running task is the
* connecting to DB.
*/

public class ConnectionFuture  implements Callable {
@Override
    public Connection call() {
        Connection conn = null;
        try {
            String JDBC_DRIVER = "com.mysql.jdbc.Driver";
            String DB_URL = "jdbc:mysql://in-csci-rrpc01.cs.iupui.edu:3306/ealikhas_db";
            //Database credentials
            String USER = "ealikhas";
            String PASS = "ealikhas";
            //Register JDBC driver
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            } catch (Exception ex) {
                System.out.println("Exception: " + ex.getCause() );
                ex.printStackTrace();
                System.out.println();
            }
        return conn;
        }
}
```

Fig.19 Sample code for ConnectionFuture class.

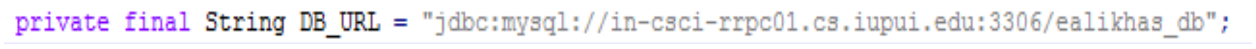
As shown in the above figure this class implements the Callable interface and overrides the call method, which attempts to create the connection through a set of steps after importing the package that includes the JDBC classes needed for database programming (**import java.sql.***):

1. **Specify the name of JDBC driver:** in this step the name of the driver, which handles the communications with the database server, will be determined in order to show that we use the MySQL driver.



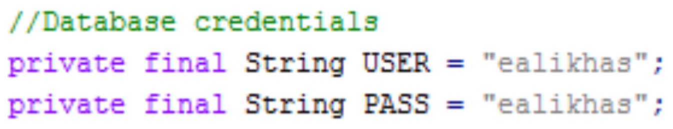
```
private final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
```

2. **Determine the Database URL:** includes the server IP address, port number, and the database name.



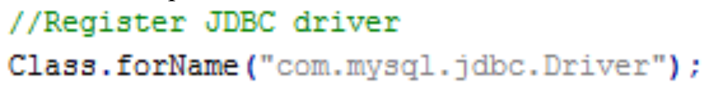
```
private final String DB_URL = "jdbc:mysql://in-csci-rrpc01.cs.iupui.edu:3306/ealikhas_db";
```

3. **Specify the Database credentials:** includes the username and password in order to access the ealikhas_db.



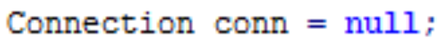
```
//Database credentials
private final String USER = "ealikhas";
private final String PASS = "ealikhas";
```

4. **Register JDBC driver:** in this step the driver will be initialized in order to connect the database.



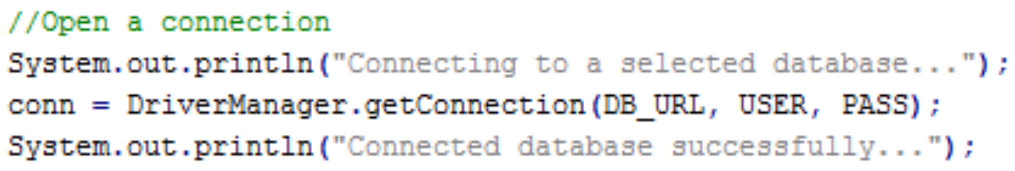
```
//Register JDBC driver
Class.forName("com.mysql.jdbc.Driver");
```

5. **Create the connection object:** which represents communication context and all the communication with the database is done through it only. This object will be the future object value.



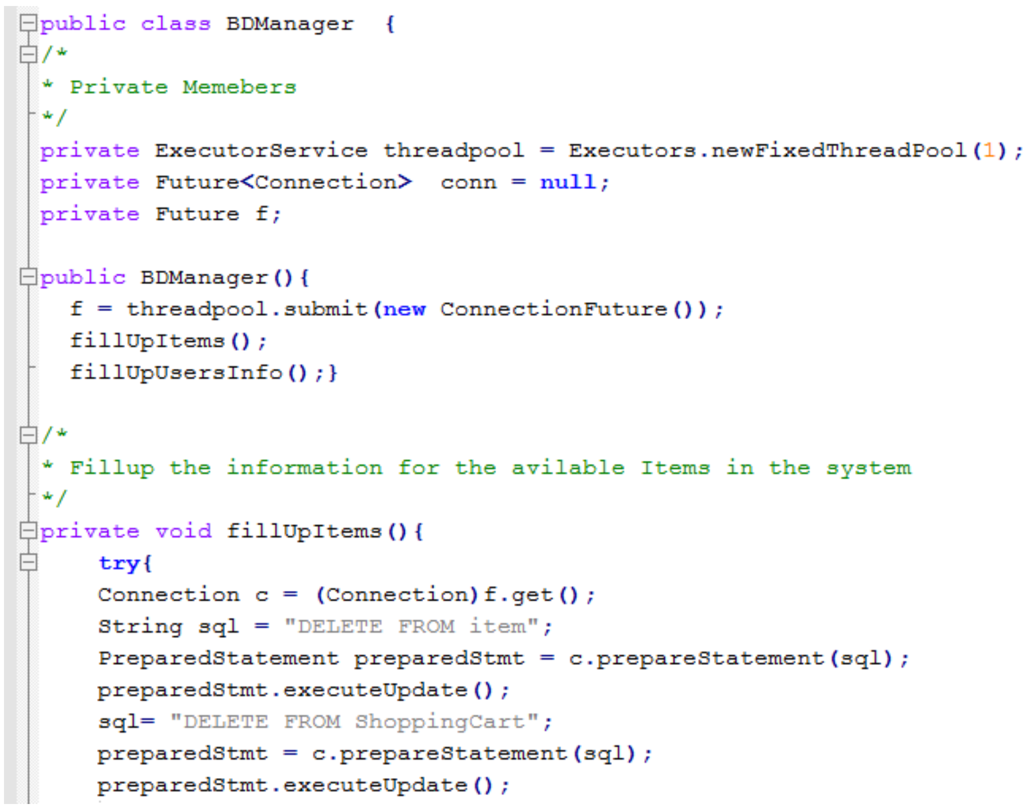
```
Connection conn = null;
```

6. **Open a connection:** in this step I used a specific method from the **DriverManager** class, which manages the database drivers and matches the connection request from the java code with the suitable database driver, this method is **getConnection()** to create a physical connection with the ealikhas_db.



```
//Open a connection
System.out.println("Connecting to a selected database...");
conn = DriverManager.getConnection(DB_URL, USER, PASS);
System.out.println("Connected database successfully...");
```

In the DBManager class a threadpool was declared in order to submit a task, which creates a connection to the database and returns a connection object, also a Future instance was declared as a connection object named conn. The submitting invokes the call method in the ConnectionFuture class, which returns a lightweight connection object as a promise object. And when the connection object is ready this means the connection is created and we can get it by using the get () method. After we get the connection object all the database queries will be performed by using it. The **TimeOutExecption** is thrown if the DBManager instance is waiting for a long time. Fig.20 Shows the DBManager code, which does all the above.



```
public class BDManager  {
/*
* Private Memebers
*/
private ExecutorService threadpool = Executors.newFixedThreadPool(1);
private Future<Connection>  conn = null;
private Future f;

public BDManager(){
  f = threadpool.submit(new ConnectionFuture());
  fillUpItems();
  fillUpUsersInfo();}

/*
* Fillup the information for the avilable Items in the system
*/
private void fillUpItems(){
    try{
    Connection c = (Connection)f.get();
    String sql = "DELETE FROM item";
    PreparedStatement preparedStmt = c.prepareStatement(sql);
    preparedStmt.executeUpdate();
    sql= "DELETE FROM ShoppingCart";
    preparedStmt = c.prepareStatement(sql);
    preparedStmt.executeUpdate();
```

Fig.20 submitting a task for ConnectionFuture.

<u>Thread-safe Interface</u>

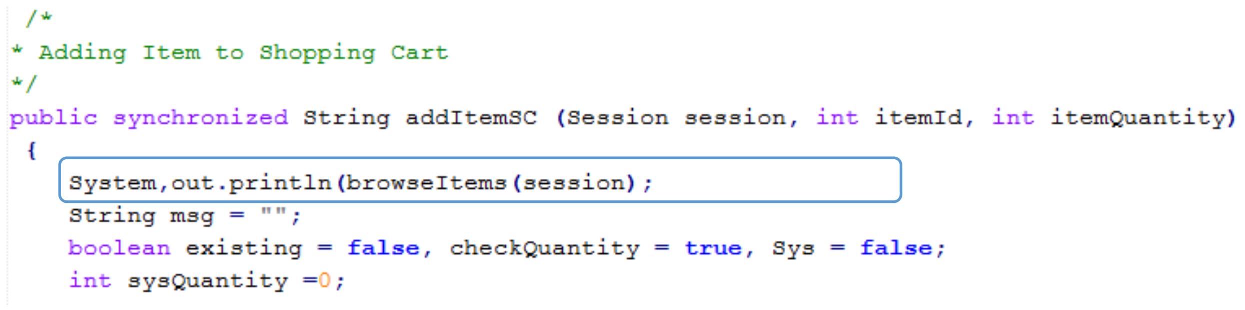
This design pattern is used in the software in order to reduce the locking overhead, ensure thread-safe access, and ensures that the intra-component methods calls do not cause a self-deadlock by trying to reacquire a lock that is held by the component already. The most common error that causes deadlock is self-deadlock or recursive deadlock. In a self-deadlock, a thread tries to acquire a lock already held by the thread. In another words, if an acquired lock is non-recursive and the method calls another method in the component that tries to acquire the same lock.

The solution which is proposed by this pattern is to structure all the components that process intra-component method invocation according to the following design:

1. A public accessible interface for the methods, where the interface performs synchronization checks at the border of component.
2. The corresponding implementation for the methods must be either private or protected, which only perform the task of the method when invoked by interface method. In this situation we make sure they will be invoked with the necessary lock(s) held and never acquire or release the lock.

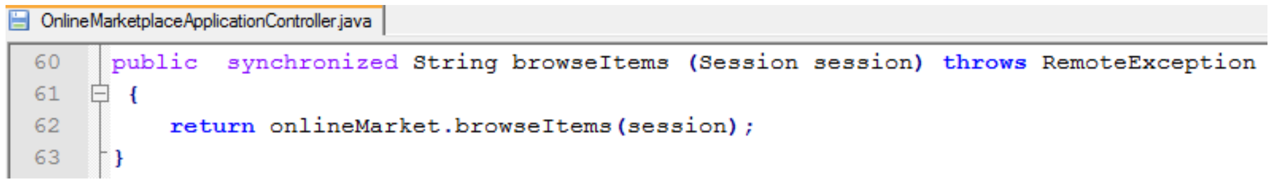
In the Online Marketplace Model layer there is a **ddItemSC** method, which is used in order to add item to the shopping cart, this method will invoke the **browseItem** method as intra-component. Thus, if I apply the Thread-safe Interface pattern to this situation by creating a public synchronized interface and convert the **addItemSC** and the **browseItem** methods into private ones. Additionally, in the interface I will invoke the **addItemSC** in order to reduce the locking overhead. In such situation when the Administrator attempts to browse the available item in the system through invoking the **browseItem** method at the Application Controller layer he can’t because the **browseItem** method is private method in the model layer, so the

ApplicationController object can't invoke it. The following figure shows that the **addItemSC** method invokes the browseItems method at the model layer.



```
/*
* Adding Item to Shopping Cart
*/
public synchronized String addItemSC (Session session, int itemId, int itemQuantity)
 {
    System,out.println(browseItems(session);
    String msg = "";
    boolean existing = false, checkQuantity = true, Sys = false;
    int sysQuantity =0;
```

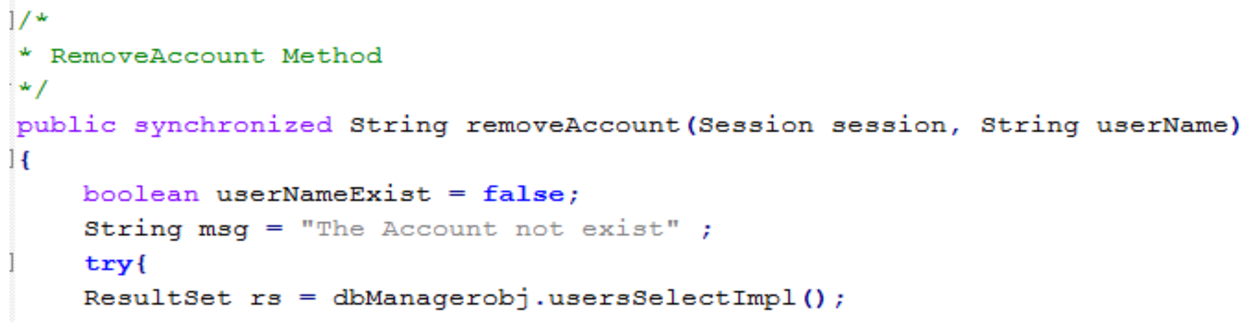
The following figure represents the source code, which invoke the browseItems method at the application controller layer.



OnlineMarketplaceApplicationController.java

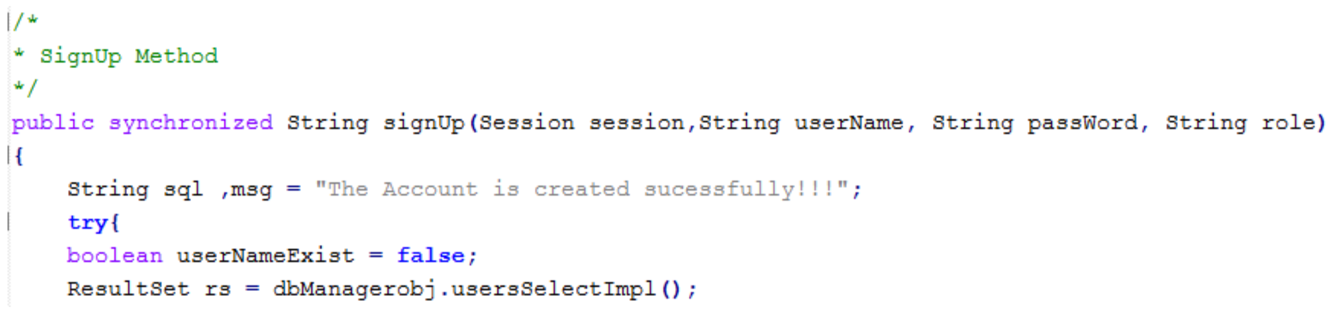
```
public  synchronized String browseItems (Session session) throws RemoteException
 {
     return onlineMarket.browseItems(session);
}
```

Thus as shown in the above figures I can't apply this pattern to my design. On the other hand the each of the methods in the model layer such as purchase method invokes one or more of the DBManager methods as an intra-component; thus, in this situation I can't also apply this pattern to the DBManager method and the Model layer methods, because all the DBManager methods have to be public and if I declare all of them as a private methods I will not able to access them. For instance the **remveAccount** method invoke one of the DBManager method which is the **usersSelectImpl()** in order to check if the user account that we want to remove it exist in the database or not. As shown below.



```
/*
 * RemoveAccount Method
 */
public synchronized String removeAccount(Session session, String userName)
{
     boolean userNameExist = false;
     String msg = "The Account not exist" ;
     try{
     ResultSet rs = dbManagerobj.usersSelectImpl();
```

On the other hand, the **signUp** method also invokes the same **DBManager** method as shown in the following figure. Thus, if I apply this pattern in this situation one of the methods (**singUp** and the **removeAccount**) will not be able to invoke the **usersSelectImpl**.



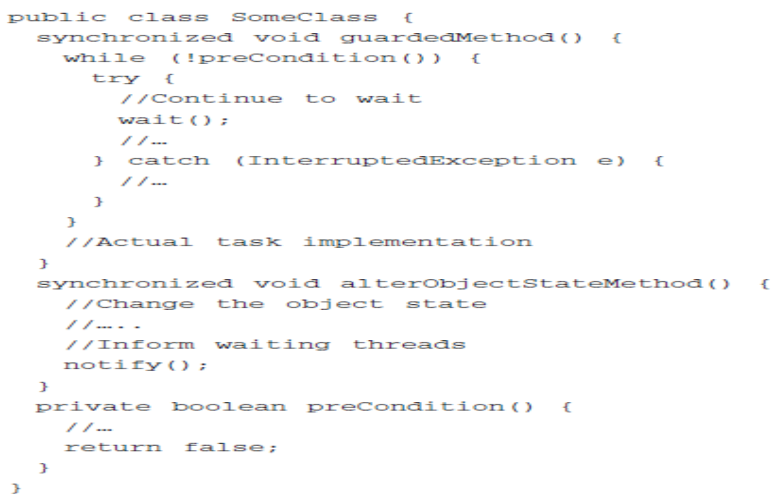
```
/*
* SignUp Method
*/
public synchronized String signUp(Session session,String userName, String passWord, String role)
{
    String sql ,msg = "The Account is created sucessfully!!!";
    try{
    boolean userNameExist = false;
    ResultSet rs = dbManagerobj.usersSelectImpl();
```

Moreover, all the client threads in the RMI server side will have only one OnlineMarketplaceModel instance which will be lock when one of them attempts to access any OnlineMarketplaceModel instance method; the synchronized ones; in addition to lock the DBManager instance, which was declared as a private class member and it will be used to invoke the methods from the DBManager class. In this situation I will be sure that only one thread will access the DBManager methods at a time.

<u>Guarded Suspension</u>

It is a software design pattern, which is applied on the concurrent software and application in order to manage the threads, which require both a lock to be acquired and a precondition to be satisfied before the invoked method can be executed. This pattern involves suspending the method call until the precondition, which acting as a guard is satisfied. The method invocation and the calling thread will be suspended for a finite and reasonable period of time. If a method call is suspended for too long, then the overall program will slow down or stop.

In general each method in the class instance is designed to execute a specific task and sometimes, when a method is invoked on the class instance, the instance need to be in a certain state, which is logically necessary for a method in order to perform its task. For this reason the guarder suspension pattern is used in order to suspend the method execution until such a precondition becomes true. The precondition here represents a particular instance state. The Java programming language uses the built-in wait, notify and **notifyAll** methods, which every class and object in the Java inherent them from the **java.lang.Object** class to implement this pattern. This methods can only be invoked from within a synchronized method or statement. In other words, the lock associated with an object must already be acquired before any of these methods are invoked. The following figure shows a simple example for the Guarded Suspension pattern.



```
public class SomeClass {
  synchronized void guardedMethod() {
    while (!preCondition()) {
      try {
        //Continue to wait
        wait();
        //…
      } catch (InterruptedException e) {
        //…
      }
    }
    //Actual task implementation
  }
  synchronized void alterObjectStateMethod() {
    //Change the object state
    //…..
    //Inform waiting threads
    notify();
  }
  private boolean preCondition() {
    //…
    return false;
  }
}
```

Fig.21 sample code for Guarded Suspension pattern

The class in this example contains two synchronized methods in order to prevent the race conditions: the **guardedMethod** and **alterObjectStateMethod**. When thread attempts to execute the **guardedMethod** they have to wait until the precondition become true else it will wait to use the wait method. additionally, **alterObjectStateMethod** method allow the threads to change the state of the class instance, which may change the value of the precondition into true and this will cause to notify the waiting threads that is waiting inside the **guardedMethod** to move one of them to ready state and then executes the **guardedMethod**. Instead of using the while loop in the above example the **object.wait()** method can be used.

This pattern is implement the monitor condition component in the monitor object pattern. The Guarded Suspension act as a blocking queue or shared queue for the monitor object pattern in order to synchronized and schedule the threads access to the synchronized block. When thread invokes a synchronized method, it will be acquire the lock first (and this is done by the Scoping locking for the monitor object pattern). In this situation the monitor condition which is the Guarded Suspension pattern will be checked if the lock is acquired by another thread this means the condition not hold and the wait method will be invoked in order to suspend the thread and the invoked method. The monitor condition or the Guarded Suspension pattern will be hold when the lock will be released. In this situation the notify or **notifyAll** method will be invoked in order to notify the suspended or the waiting threads to access and execute the method.

Scoped Locking

The Scoped Locking pattern ensures that the shared resource which is in our situation is the remote object methods, typically a lock such as a mutex or some other synchronization primitive, is acquired and held for the duration of the critical section of a block of code and is automatically released when that critical section ends even if it is terminated through thrown an exception. And in this way we make sure no a substantial problems for the other threads will be happen.

This pattern is acting as a part of the monitor object pattern where it implements the monitor lock component of that pattern. Thus; the Java programming language makes use of this pattern via implementation the synchronization block, which form the monitor object pattern.

There are two different mechanism to implement the Scoping Locking pattern in Java programming language. The first one, if we identify the monitor object pattern in our code using the synchronized statements through using the synchronized keyword with an expression that evaluates to an object reference. In this situation the compiler will generates two instructions the **monitorenter** and **monitorexit**, which are used for synchronization blocks within methods. When the **monitorenter** is encountered it acquires the lock for the object referred to the **objectref** on the stack. But if the thread already own that object, the count which is associated with the lock will be incremented. On the other hand, every time the **monitorexit** is executed for the thread on the object, the count will be decreased and when its value becomes zero the lock or the monitor will be released.

The second mechanism is when we identify the monitor object pattern in our code using the synchronized method. There are no any special instructions have to use in order to invoke or return from the synchronized methods. If the method is determined as a synchronized one, the virtual machine acquires a lock before invoking the method. For an instance method, the virtual machine acquires the lock associated with the object upon which the method is being invoked. For a class method, it acquires the lock associated with the class to which the method belongs (it locks a Class object). After a synchronized method completes, whether it completes by returning or by throwing an exception, the virtual machine releases the lock. In the Online Marketplace application the synchronized methods were used to identify the critical section the code and to implement the monitor object pattern; thus the **monitorenter** and **monitorexit** instructions will not be used.

Monitor Object pattern

This concurrency pattern is used to design the multi-threaded software that share resources among multiple threads. In multi-thread applications there are multiple threads invoking methods on an object which modify its internal state. In such cases by using the monitor object pattern the access to these methods can be synchronized and scheduled [POSA2]. The main components of his pattern are: monitor object, monitor lock, and monitor condition.

The monitor object pattern is implemented in the Java programming language by using the synchronized keyword. Java provides two type of synchronization; at statement level and at method level (static synchronized method and non-static synchronized method). Synchronized keyword with a variable is illegal and will result in compilation error, instead of synchronized variable in java the java volatile variable is used.

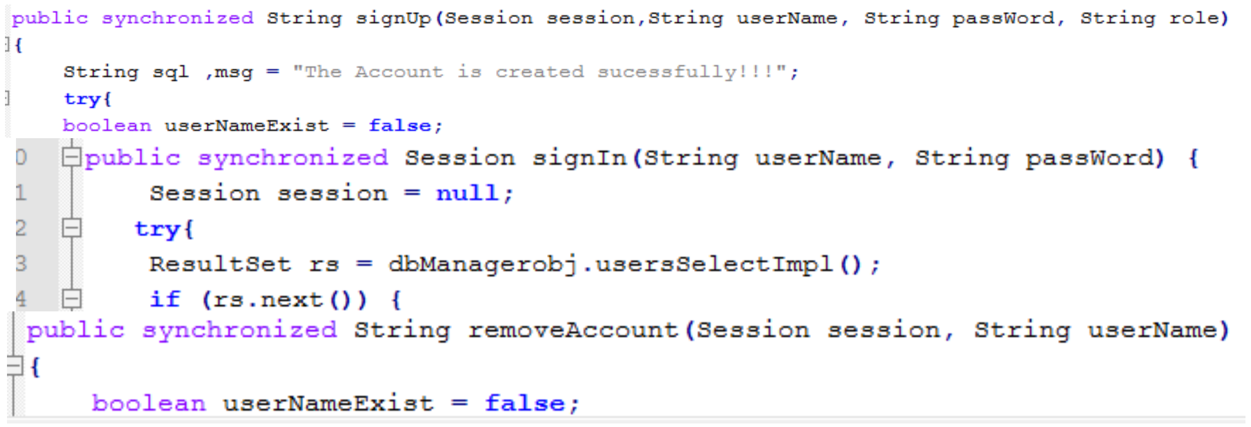
The first and important step to implement the monitor object pattern is to specify or determine the critical section of the code, which may be called sometime monitor region. This section of code has to execute only once at a time for instance the **updatItem, removeItem, removeAccount**, and **purchaseItem** methods have to specify as a monitor region because each of them modify the data in the database. After determining the monitor region either the synchronized method or statement will be used to create a synchronized code and monitor object.

Synchronization does not address the problem of which thread executes the statements first: it is first come, first served. The mechanism that the Java uses to support the synchronization is known as the monitor lock, intrinsic lock, or monitor. Java's monitor supports two kinds of thread synchronization: mutual exclusion and cooperation. Mutual exclusion, which is supported in the Java virtual machine via object locks, enables multiple threads to independently work on shared data without interfering with each other and prevent the data race. Cooperation, which is supported in the Java virtual machine via the wait and notify methods of class Object, enables threads to work together towards a common goal. In general because the java programming language support the multithreading, so each object in the java will automatically have a monitor or intrinsic lock object associated with it; this monitor object is a synchronization mechanism placed at the class hierarchy root (**java.lang.Object**).

Before a synchronized block can be entered, a thread needs to gain ownership of the monitor for that block. Once the thread has gained ownership of the monitor, no other thread synchronized on the same monitor can gain entry to that block (or any other block or method synchronized on the same monitor). The thread owning the monitor gets to execute all the statements in the block, and then automatically releases ownership of the monitor on exiting the block. At that point, another thread waiting to enter the block can acquire ownership of the monitor. However, threads synchronized on different monitors can gain entry to the same block at any time. For example, a block defined with a synchronized (this) expression is synchronized on the monitor of this object. If this is an object that is different for two different threads, both threads can gain ownership of their own monitor for that block, and both can execute the block at the same time. This won't matter if the block affects only variables specific to its thread (such as instance variables of this), but can lead to corrupt states if the block alters variables that are shared between the threads.

A thread can acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables reentrant synchronization. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block. Finally, the Scope-locking pattern will implements the monitor lock and the Guarded Suspension pattern will implement the monitor condition.

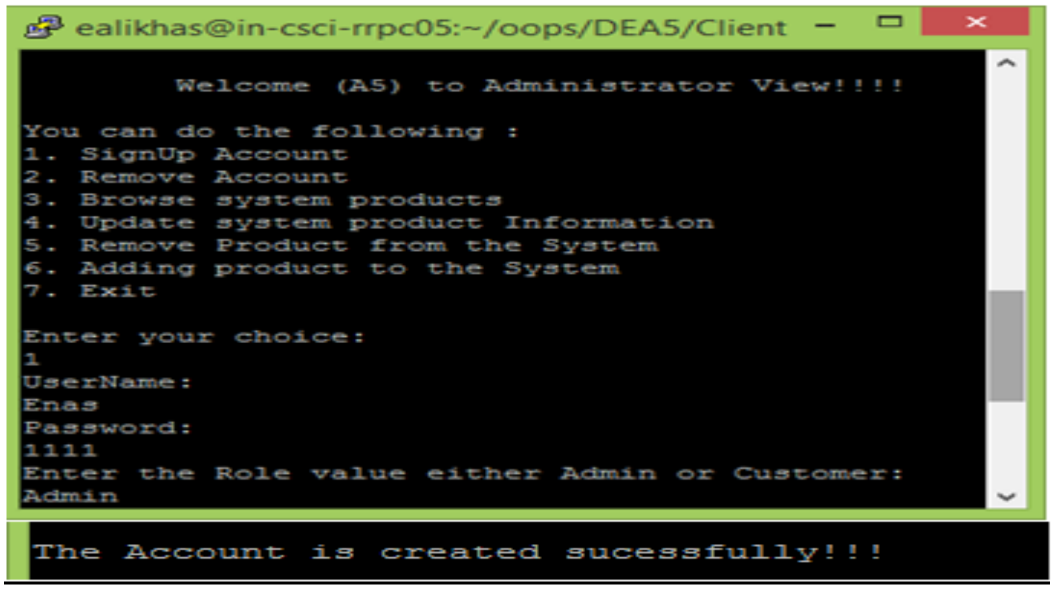
The following figure shows a sample of code which a monitor object pattern was applied to it. I synchronized all the remote object methods to lock the write operation to the data and to allow the read operation to read a consistence data from DB.



```
public synchronized String signUp(Session session,String userName, String passWord, String role)
{
    String sql ,msg = "The Account is created sucessfully!!!";
    try{
    boolean userNameExist = false;
public synchronized Session signIn(String userName, String passWord) {
        Session session = null;
      try{
        ResultSet rs = dbManagerobj.usersSelectImpl();
        if (rs.next()) {
public synchronized String removeAccount(Session session, String userName)
{
     boolean userNameExist = false;
```

4 Sample Running

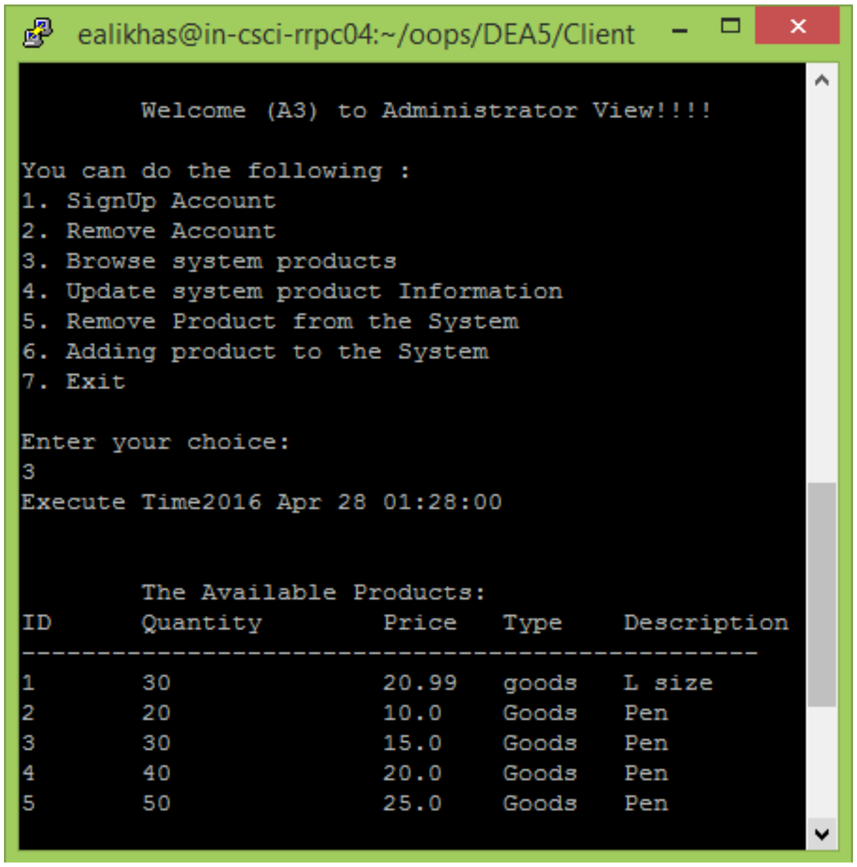
The following figures shows how the Online Marketplace application works for both the Administrators and Customers. And how this application act as a multithread application. The following represent the Administrator view.



```
        Welcome (A5) to Administrator View!!!!

You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
1
UserName:
Enas
Password:
1111
Enter the Role value either Admin or Customer:
Admin
```

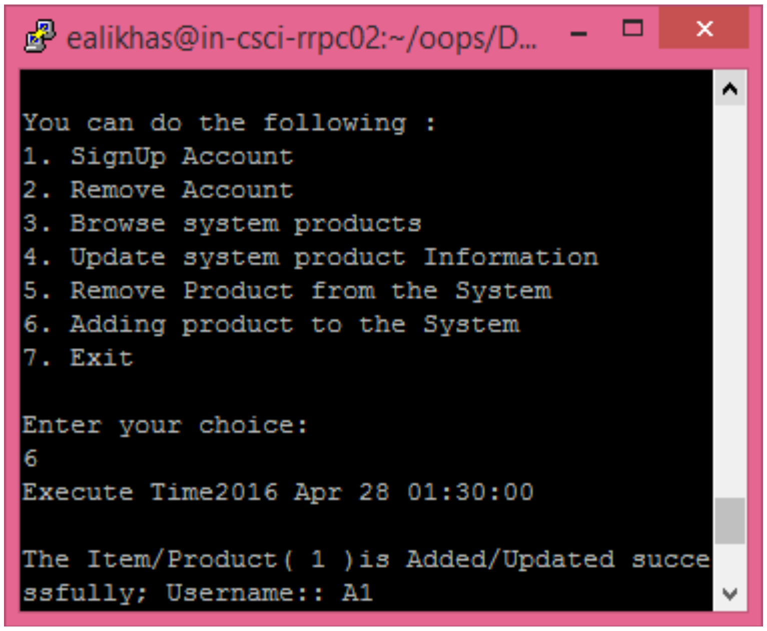
```
The Account is created sucessfully!!!
```

This figure shows the browse method in the Administrator view.


ealikhas@in-csci-rrpc04:~/oops/DEA5/Client

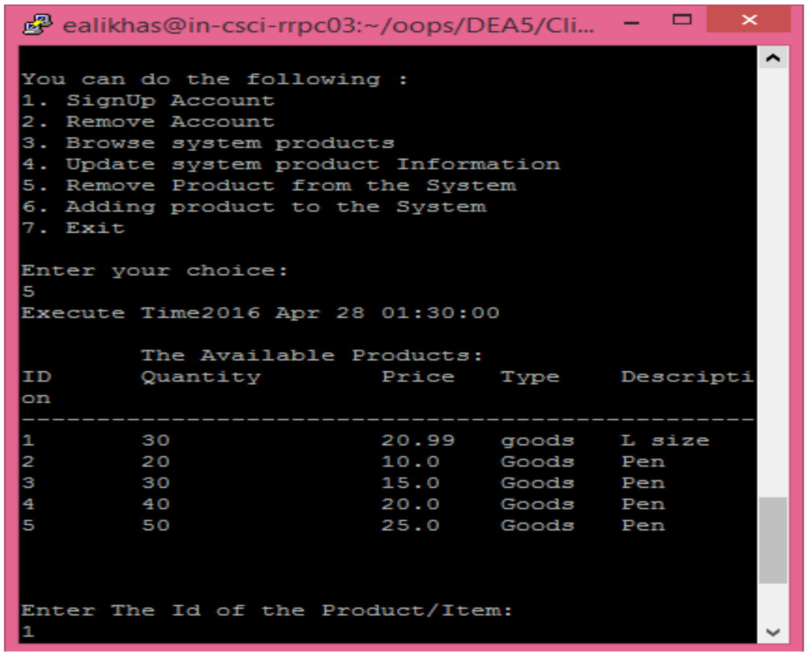
```
        Welcome (A3) to Administrator View!!!!

You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
3
Execute Time2016 Apr 28 01:28:00


        The Available Products:
ID      Quantity        Price   Type    Description
-------------------------------------------------
1       30              20.99   goods   L size
2       20              10.0    Goods   Pen
3       30              15.0    Goods   Pen
4       40              20.0    Goods   Pen
5       50              25.0    Goods   Pen
```

This figure represents the add new item to the system.,


ealikhas@in-csci-rrpc02:~/oops/D...

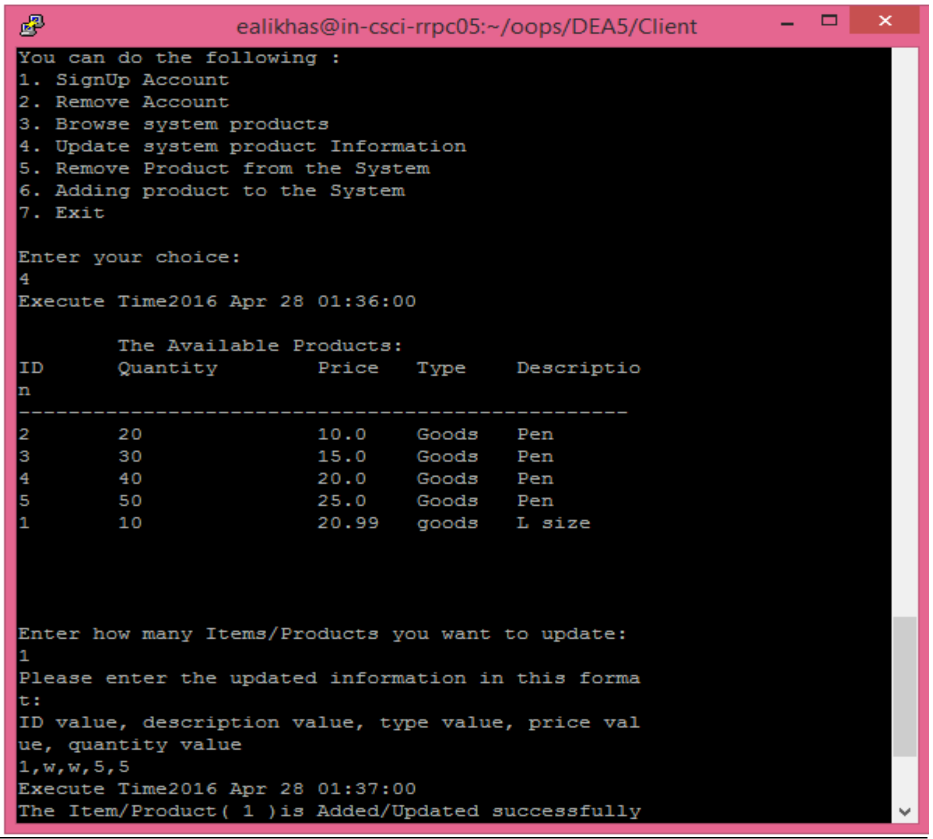
```
You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
6
Execute Time2016 Apr 28 01:30:00

The Item/Product( 1 )is Added/Updated succe
ssfully; Username:: A1
```

This figure shows the removing item from the system



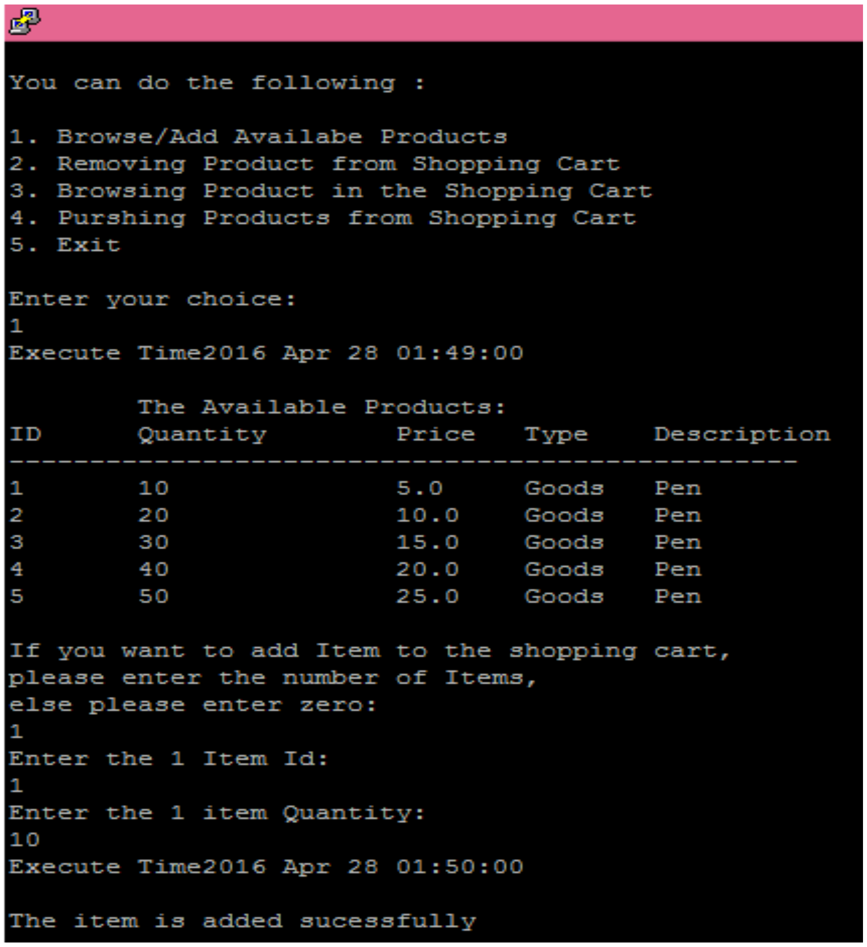
```
You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
5
Execute Time2016 Apr 28 01:30:00

        The Available Products:
ID      Quantity        Price    Type     Descripti
on
-------------------------------------------------
1       30              20.99    goods    L size
2       20              10.0     Goods    Pen
3       30              15.0     Goods    Pen
4       40              20.0     Goods    Pen
5       50              25.0     Goods    Pen


Enter The Id of the Product/Item:
1
```

This figure represents the update item method



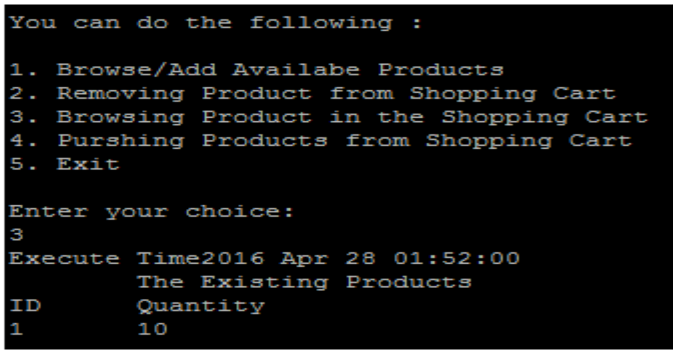
```
You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
4
Execute Time2016 Apr 28 01:36:00

        The Available Products:
ID      Quantity        Price    Type     Descriptio
n
-------------------------------------------------
2       20              10.0     Goods    Pen
3       30              15.0     Goods    Pen
4       40              20.0     Goods    Pen
5       50              25.0     Goods    Pen
1       10              20.99    goods    L size



Enter how many Items/Products you want to update:
1
Please enter the updated information in this forma
t:
ID value, description value, type value, price val
ue, quantity value
1,w,w,5,5
Execute Time2016 Apr 28 01:37:00
The Item/Product( 1 )is Added/Updated successfully
```

The following figures show the Customer methods. In the following one the Customer can browse the system items and add any number of them to the shopping cart.



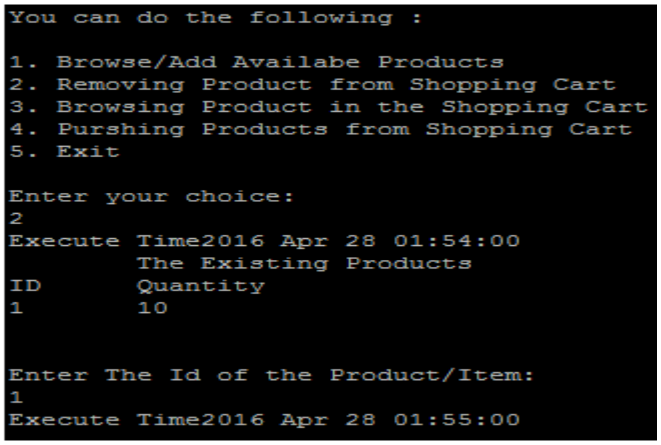
```
You can do the following :

1. Browse/Add Availabe Products
2. Removing Product from Shopping Cart
3. Browsing Product in the Shopping Cart
4. Purshing Products from Shopping Cart
5. Exit

Enter your choice:
1
Execute Time2016 Apr 28 01:49:00

        The Available Products:
ID      Quantity        Price   Type    Description
-------------------------------------------------
1       10              5.0     Goods   Pen
2       20              10.0    Goods   Pen
3       30              15.0    Goods   Pen
4       40              20.0    Goods   Pen
5       50              25.0    Goods   Pen

If you want to add Item to the shopping cart,
please enter the number of Items,
else please enter zero:
1
Enter the 1 Item Id:
1
Enter the 1 item Quantity:
10
Execute Time2016 Apr 28 01:50:00

The item is added sucessfully
```

While in the following one the Customer can browse the item in the Shopping cart.



```
You can do the following :

1. Browse/Add Availabe Products
2. Removing Product from Shopping Cart
3. Browsing Product in the Shopping Cart
4. Purshing Products from Shopping Cart
5. Exit

Enter your choice:
3
Execute Time2016 Apr 28 01:52:00
        The Existing Products
ID      Quantity
1       10
```

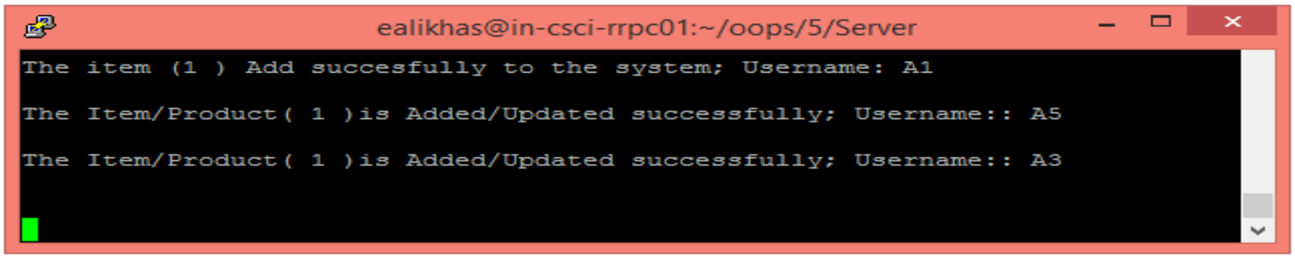
In the following figure the Customer remove the item1 from the shopping cart.



```
You can do the following :

1. Browse/Add Availabe Products
2. Removing Product from Shopping Cart
3. Browsing Product in the Shopping Cart
4. Purshing Products from Shopping Cart
5. Exit

Enter your choice:
2
Execute Time2016 Apr 28 01:54:00
        The Existing Products
ID      Quantity
1       10


Enter The Id of the Product/Item:
1
Execute Time2016 Apr 28 01:55:00
```

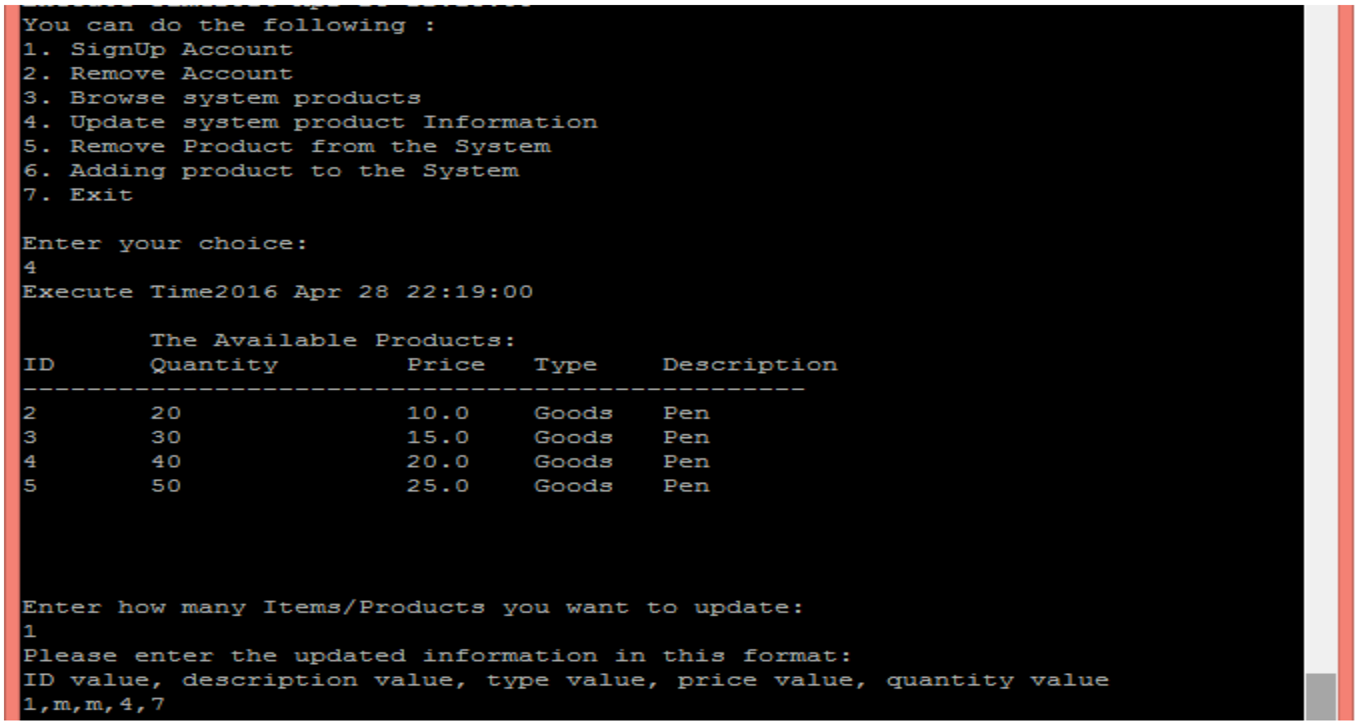
Concurrent Sample running

3 Admins attempts to add same item with same quantity concurrently. It is don successfully.

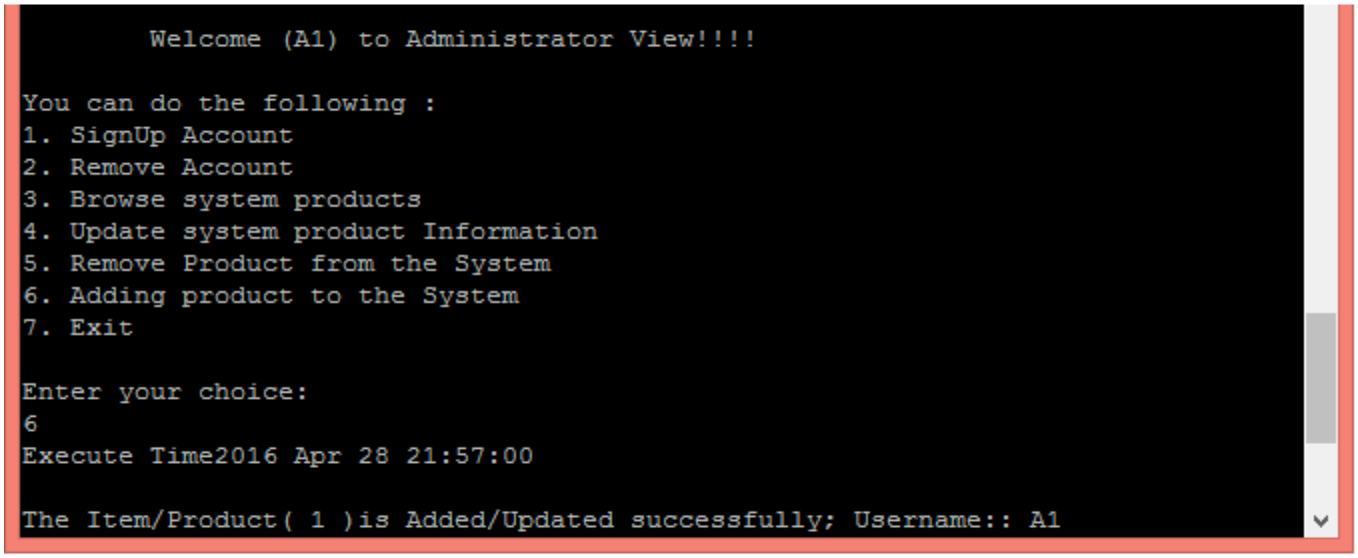


```
The item (1 ) Add succesfully to the system; Username: A1

The Item/Product( 1 )is Added/Updated successfully; Username:: A5

The Item/Product( 1 )is Added/Updated successfully; Username:: A3
```

One Admin adds item1 with quantity 10 and another one update the same item successfully.

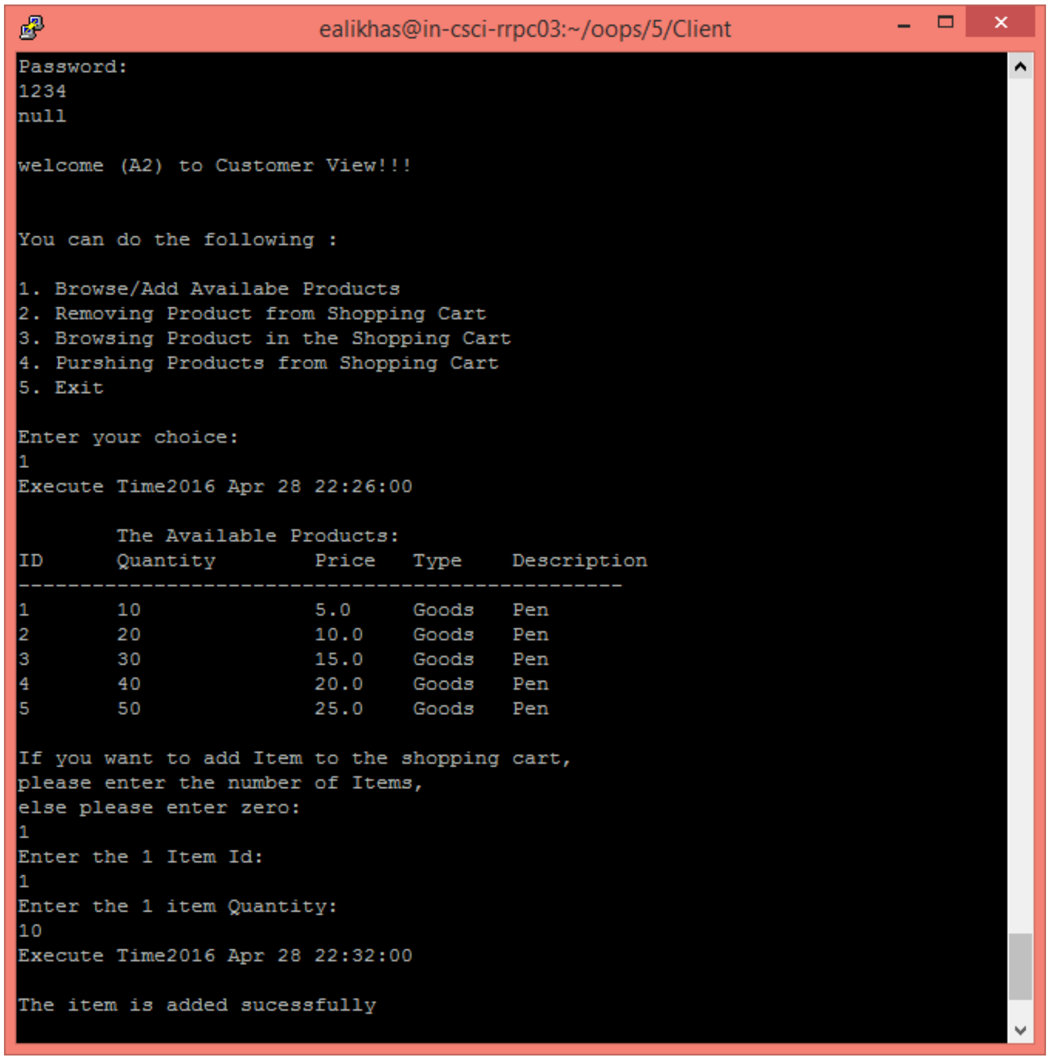


```
You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
4
Execute Time2016 Apr 28 22:19:00

        The Available Products:
ID      Quantity        Price   Type    Description
-------------------------------------------------
2       20              10.0    Goods   Pen
3       30              15.0    Goods   Pen
4       40              20.0    Goods   Pen
5       50              25.0    Goods   Pen




Enter how many Items/Products you want to update:
1
Please enter the updated information in this format:
ID value, description value, type value, price value, quantity value
1,m,m,4,7
```

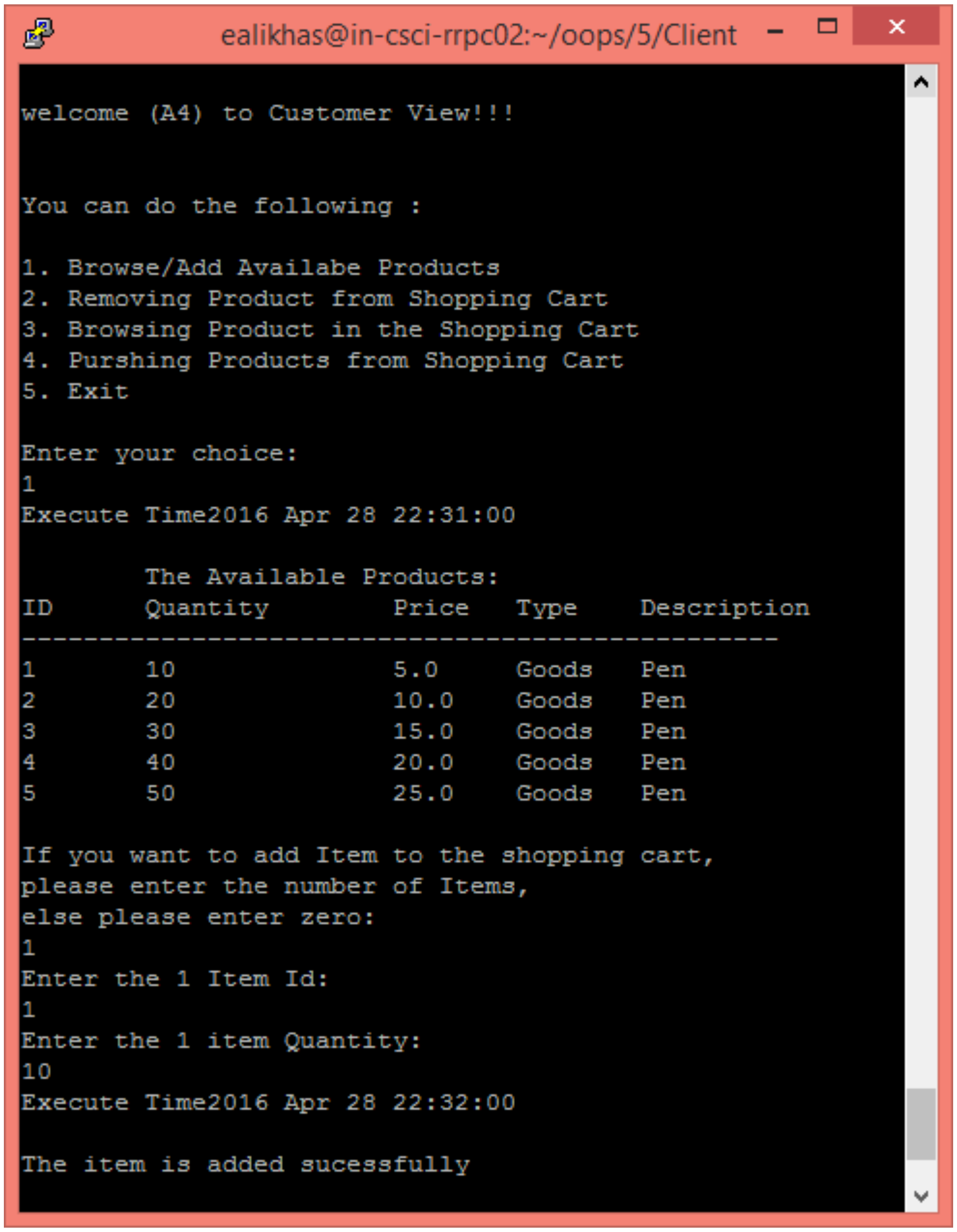


```
        Welcome (A1) to Administrator View!!!!

You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
6
Execute Time2016 Apr 28 21:57:00

The Item/Product( 1 )is Added/Updated successfully; Username:: A1
```

Two Customers browse and add same item with same quantity to the shopping cart concurrently. The item added successfully for both of them.

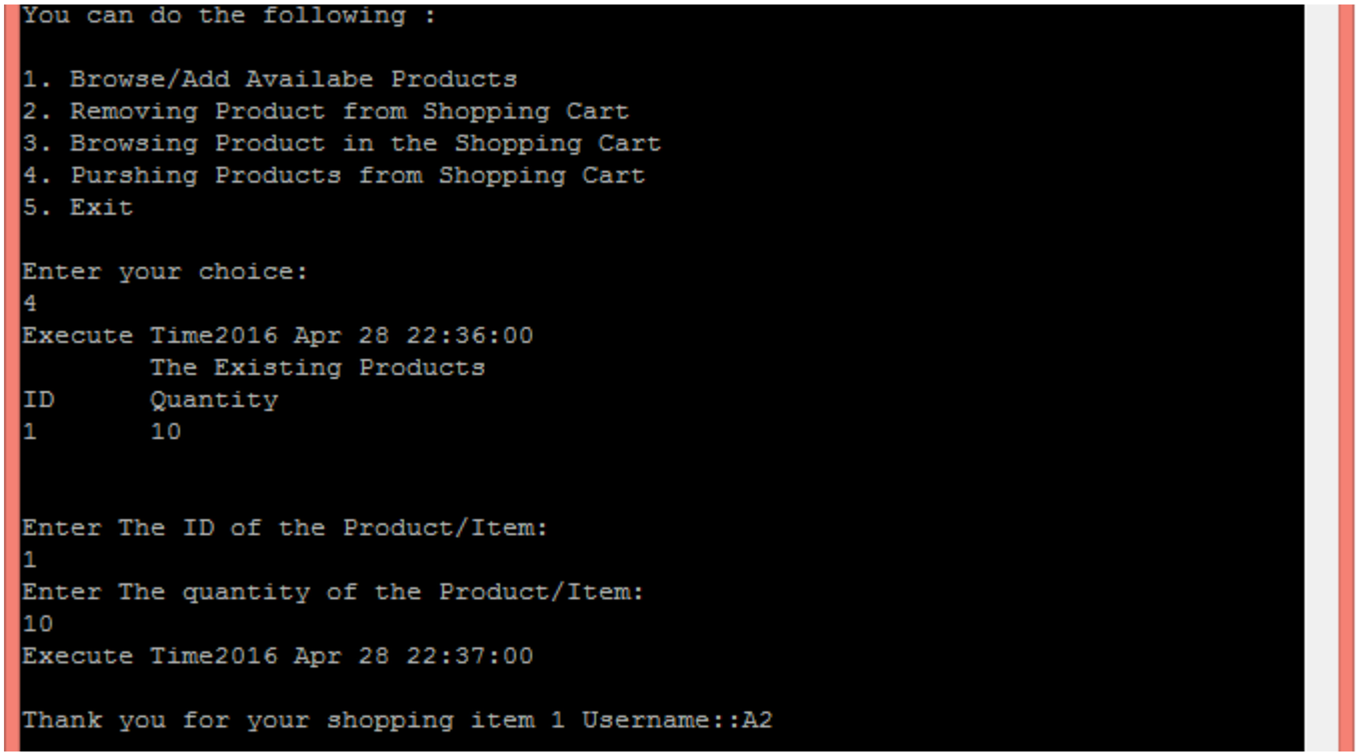


```
Password:
1234
null

welcome (A2) to Customer View!!!


You can do the following :

1. Browse/Add Availabe Products
2. Removing Product from Shopping Cart
3. Browsing Product in the Shopping Cart
4. Purshing Products from Shopping Cart
5. Exit

Enter your choice:
1
Execute Time2016 Apr 28 22:26:00

        The Available Products:
ID      Quantity        Price   Type    Description
-------------------------------------------------
1       10              5.0     Goods   Pen
2       20              10.0    Goods   Pen
3       30              15.0    Goods   Pen
4       40              20.0    Goods   Pen
5       50              25.0    Goods   Pen

If you want to add Item to the shopping cart,
please enter the number of Items,
else please enter zero:
1
Enter the 1 Item Id:
1
Enter the 1 item Quantity:
10
Execute Time2016 Apr 28 22:32:00

The item is added sucessfully
```

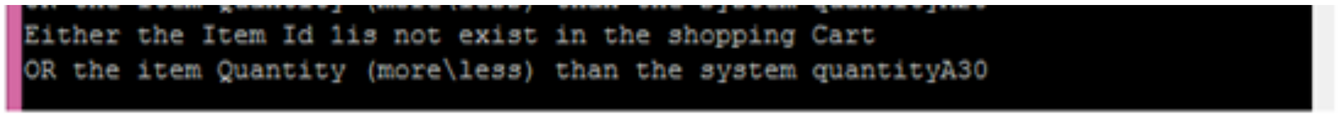


```
ealikhas@in-csci-rrpc02:~/oops/5/Client

welcome (A4) to Customer View!!!


You can do the following :

1. Browse/Add Availabe Products
2. Removing Product from Shopping Cart
3. Browsing Product in the Shopping Cart
4. Purshing Products from Shopping Cart
5. Exit

Enter your choice:
1
Execute Time2016 Apr 28 22:31:00

        The Available Products:
ID      Quantity        Price   Type    Description
-------------------------------------------------
1       10              5.0     Goods   Pen
2       20              10.0    Goods   Pen
3       30              15.0    Goods   Pen
4       40              20.0    Goods   Pen
5       50              25.0    Goods   Pen

If you want to add Item to the shopping cart,
please enter the number of Items,
else please enter zero:
1
Enter the 1 Item Id:
1
Enter the 1 item Quantity:
10
Execute Time2016 Apr 28 22:32:00

The item is added sucessfully
```

Two Customers try to purchase same item with same quantity and it equals to the quantity in the system concurrently. One of them purchase the item successfully, while the another got warning message no enough quantity in the system.

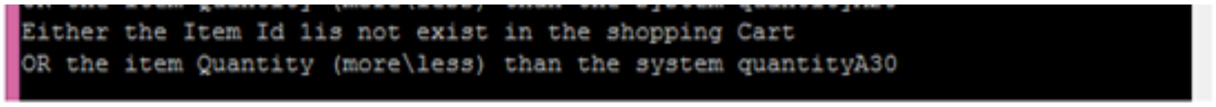


```
You can do the following :

1. Browse/Add Availabe Products
2. Removing Product from Shopping Cart
3. Browsing Product in the Shopping Cart
4. Purshing Products from Shopping Cart
5. Exit

Enter your choice:
4
Execute Time2016 Apr 28 22:36:00
        The Existing Products
ID      Quantity
1       10


Enter The ID of the Product/Item:
1
Enter The quantity of the Product/Item:
10
Execute Time2016 Apr 28 22:37:00

Thank you for your shopping item 1 Username::A2
```

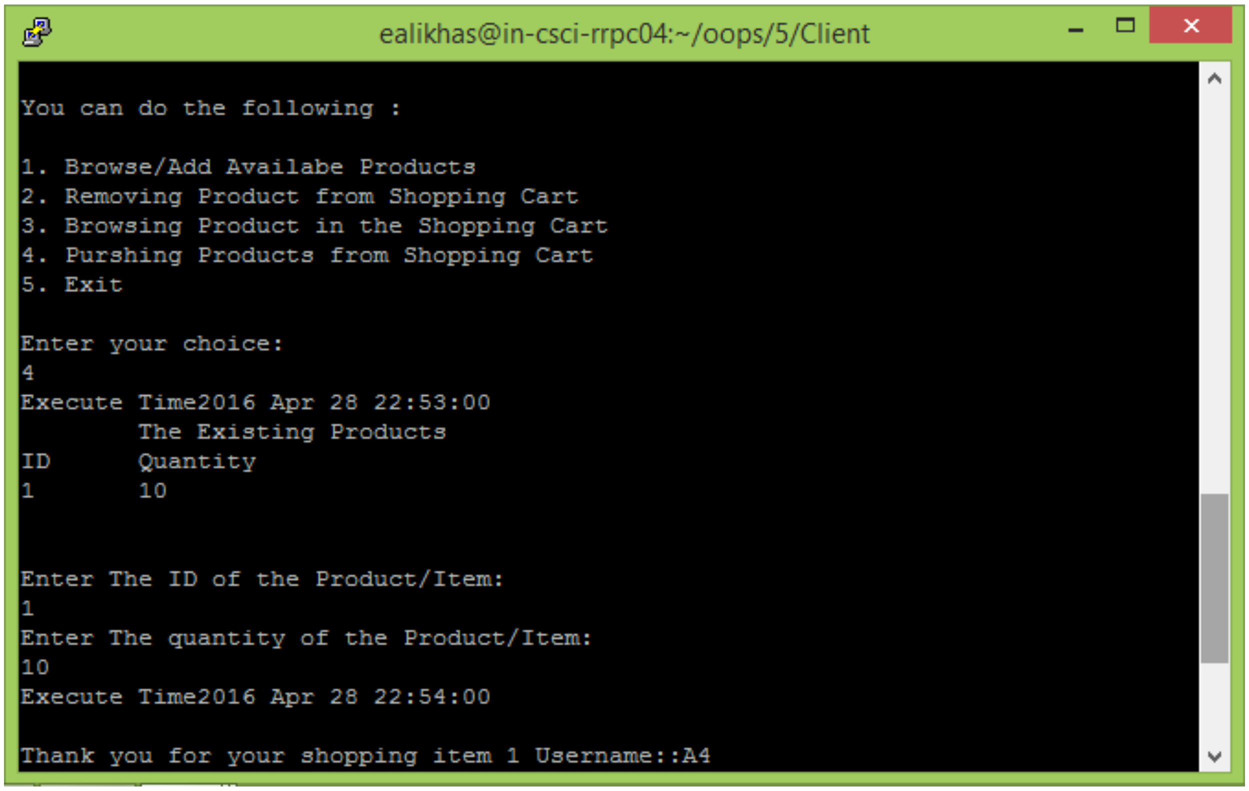


```
Either the Item Id 1is not exist in the shopping Cart
OR the item Quantity (more\less) than the system quantityA30
```

Admin add/update item1 with quantity 10, and the system has 10 as quantity, another 2 Customers attempt in the same time to purchase same item with same quantity. As we discussed on the monitor object pattern the synchronization schedule the access to the method but don't guarantee the order of request execution thus if the Admin perform first both of the Customers will purchase the item successfully because the quantity in the system will be enough else one customer will done while the another will not.

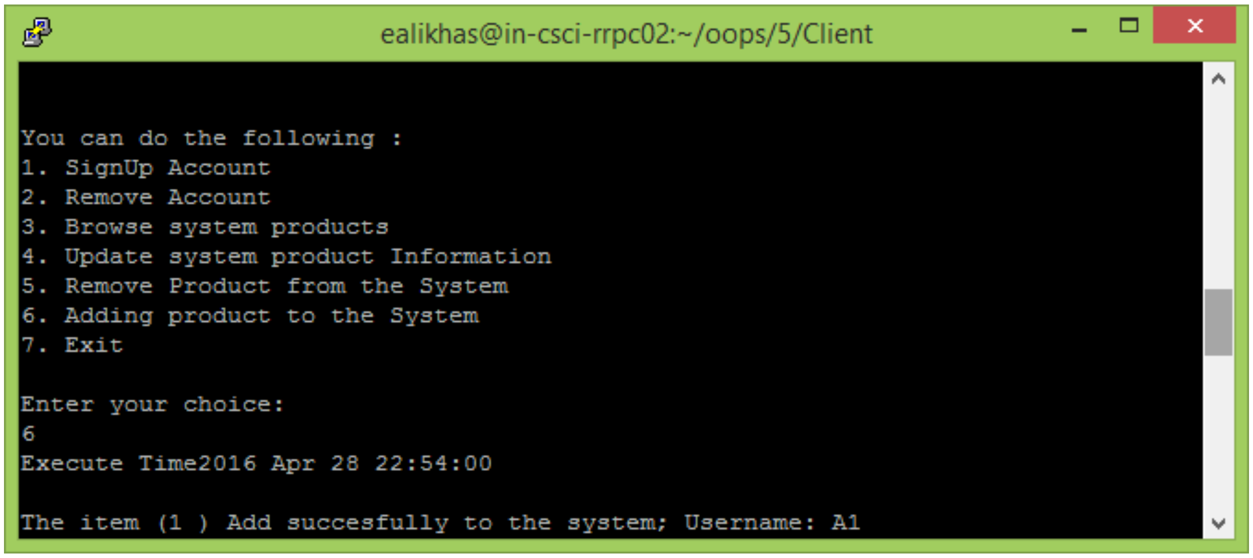


```
Either the Item Id 1is not exist in the shopping Cart
OR the item Quantity (more\less) than the system quantityA30
```



ealikhas@in-csci-rrpc04:~/oops/5/Client

```
You can do the following :

1. Browse/Add Availabe Products
2. Removing Product from Shopping Cart
3. Browsing Product in the Shopping Cart
4. Purshing Products from Shopping Cart
5. Exit

Enter your choice:
4
Execute Time2016 Apr 28 22:53:00
        The Existing Products
ID      Quantity
1       10


Enter The ID of the Product/Item:
1
Enter The quantity of the Product/Item:
10
Execute Time2016 Apr 28 22:54:00

Thank you for your shopping item 1 Username::A4
```



ealikhas@in-csci-rrpc02:~/oops/5/Client

```
You can do the following :
1. SignUp Account
2. Remove Account
3. Browse system products
4. Update system product Information
5. Remove Product from the System
6. Adding product to the System
7. Exit

Enter your choice:
6
Execute Time2016 Apr 28 22:54:00

The item (1 ) Add succesfully to the system; Username: A1
```

The result of all the above running samples happens in this way because none of the clients while be execute any method concurrently and this because we use the monitor object pattern (synchronized methods) in the application controller layer and the model layer. Thus, in this situation when the first thread invokes the synchronized remote method using the **applicationcontroller** class instance and the model class instance it will acquire the lock on these instances (the class instance). And because all the clients use same object (based on my design) they will by suspended until the thread release the lock and they can't invoke any method. If the lock is released the waiting threads will notify using either the **notify** method or **notifyAll** method because the monitor condition is hold by releasing the lock. And one of them will acquire this lock and execute the method.

5 Conclusion

The most important stage in developing and design a system or application is to fully understand the requirements and the domain of that system for this reason a domain model was created to allow the developer focus on the design. The domain model identified the basic classes for this application and how they related and cooperate. There are three basic components: Client, Server, and Database. In order to build and connect the client and Server the Java RMI was used. Apply the design patterns to a distributed system is a difficult task, because the designers have to concern the trade-off between the patterns and they have to increase the cohesion and coupling. A set of design patterns were applied for the Online Marketplace application such as: Front controller, Application Controller, monitor object patterns. Each of them play important role.

6 References

https://dzone.com/articles/javautilconcurrentfuture

http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html

http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

https://www.artima.com/insidejvm/ed2/threadsynch.html

https://en.wikipedia.org/wiki/Guarded_suspension

http://sudo.ch/unizh/concurrencypatterns/ConcurrencyPatterns.pdf