ONLINE MARKETPLACE

Assignment 2

Pritesh Ratnappagol

FEBRUARY 13, 2018

INDIANA UNIVERSITY – PURDUE UNIVERSITY INDIANAPOLIS

Table of Contents

Design Patterns:

Front Controller Pattern

Front Controller provides a centralized request handling mechanism so that all requests will be handled by a single hand.
Front Controller is a centralized entry point for handling requests
Front Controller will do the authentication/ authorization/ logging or tracking of request and then pass the requests to the corresponding handlers.
Front controller has the following entities :

Front Controller :	It centralizes the handling of all kinds of request coming to the application
Dispatcher :	Object of the Diaptacher is created at the front controller which helps to delegate the requests to the specific corresponding handler
View :	Views are the objects of which request are made.

Benefits of using Front Controller Pattern:

- Provides centralized controlled logic
- Improves the reusability of the code.
- Improves the separation of concerns and manageability.

Abstract Factory Pattern:

Abstract Factory Pattern comes under creational pattern category.
It is one of the best method for creating an object.
Provides an interface for creating families of dependent or related objects without specifying their concrete class.
As the name suggests abstract factory is a super factory which creates factory of factories.
It always Divides responsibilities between multiple classes.

Benefits of using Abstract Factory Pattern:

- It isolates concrete classes from the client.
- Exchanging product families is easy.
- It promotes consistency among products.

Command Pattern:

Command Pattern Design falls under behavioral design pattern category.
In this pattern an object is used to represent and encapsulate all the information needed to call the method when required.
This information includes the method name, object that owns the method and values needed for method parameter.
In this whenever a client or application part of program wants to execute a specific command or list of code to be executed this pattern calls the calls the execute method on one the

encapsulated objects then an object calls the invoker which invokes to perform a command to set in motion and transfers this command to another object that is called receiver and receiver has the actual code which you want to execute. So whenever invoker gives the command to receiver it executes that corresponding code or method.

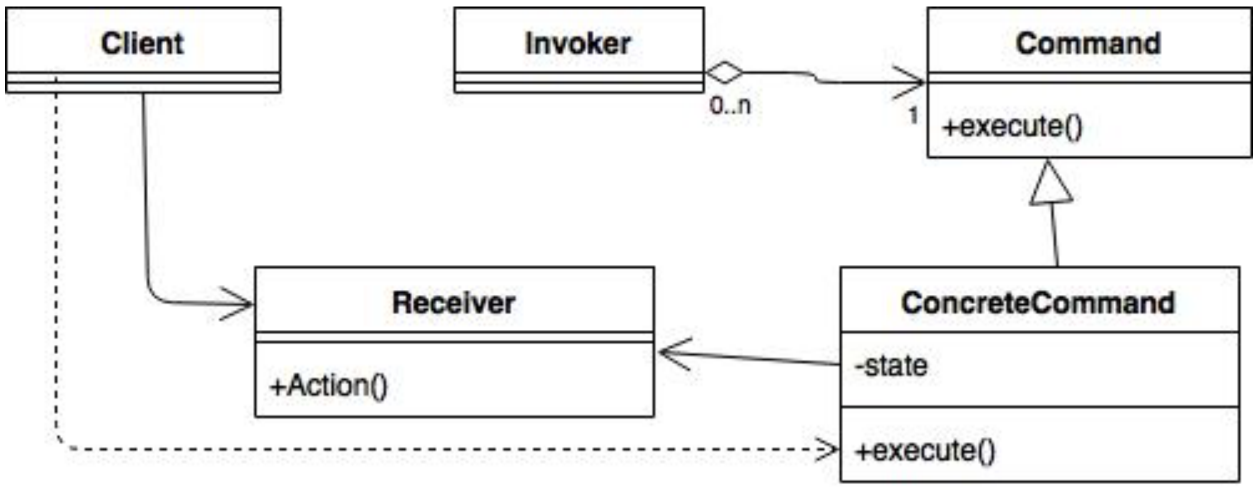


Fig. 1 Command Pattern

In this whenever a client or application part of program wants to execute a specific command or list of code to be executed this pattern calls the calls the execute method on one the encapsulated objects then an object calls the invoker which invokes to perform a command to set in motion and transfers this command to another object that the calls receiver and receiver has the actual code which you want to execute. So whenever invoker gives the command to receiver it executes that corresponding code or method. Figure 1 explains the same.

Assignment Discussion:

In Assignment 1 we have already successfully created an Online Market Place Application using RMI technology for client and server program using MVC design pattern.
In this assignment I have implemented 3 different design patterns discussed above with proper understanding of each pattern.

Front Controller Pattern:

I have used the controller part of the MVC pattern as the front controller for this assignment. In this front controller class I am taking the data from the user to authenticate and delegate the responsibility to the dispatcher to show the admin or client screen according to the user input.

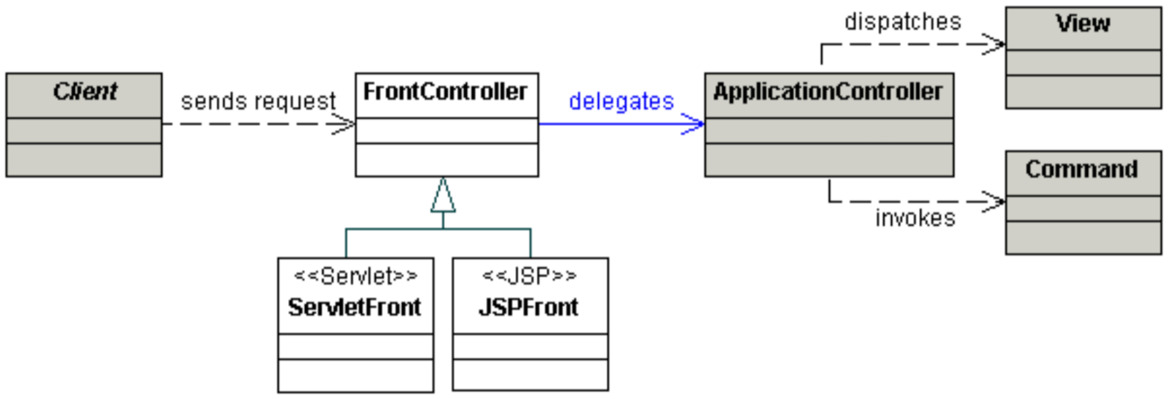


Fig 2. Front Controller

Front controller is the entry point for the application.
After starting the application it asks the user the type of login he wants to make (e.g.: customer or admin) by calling the method in other class.
After that it asks the user for its user-id which provides the user with three options for entering the type of username he wants to login in like:

1. Login by Username
2. Login by Email-id
3. Login by Mobile Number

After this it asks for the password and then encapsulates all the parameters taken form client and sends it to the server side where authentication of the user is done.
If the server is authenticated sends the Boolean value “true” or “false” to the client.
Then client displays the message accordingly i.e. if “true” User is authenticated else not authenticated.

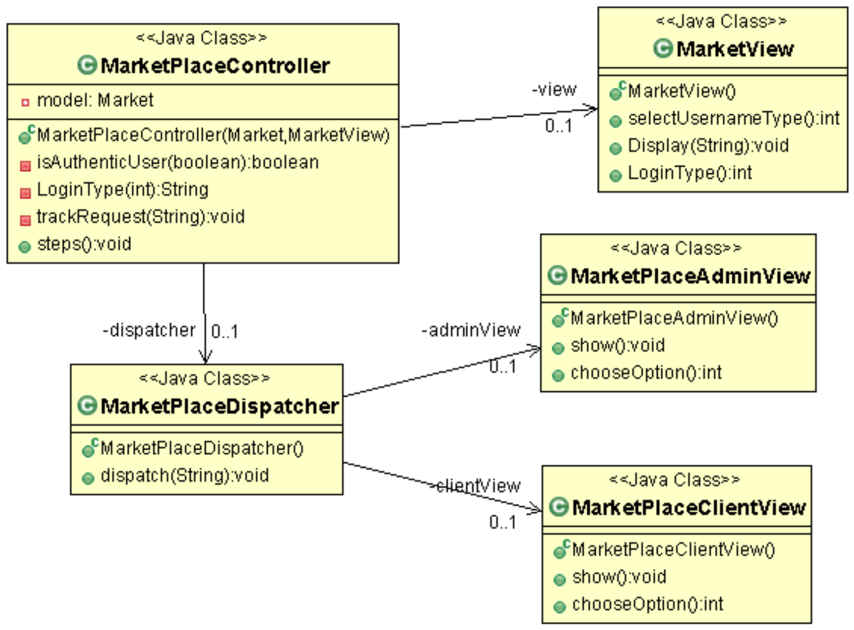


fig 3. Class diagram for front controller

Then front controller delegates the responsibility to the dispatcher where dispatcher checks the type of user whether customer or admin and displays the different views for the corresponding type. (i.e. if user is the customer dispatcher passes it to the customer page and if admin it passes it to the admin page).

Abstract Factory Pattern:

I have implemented this pattern when the user is asked for the type of login, username and password. This all things are called from the Factory design pattern of Username Factory and Password Factory.

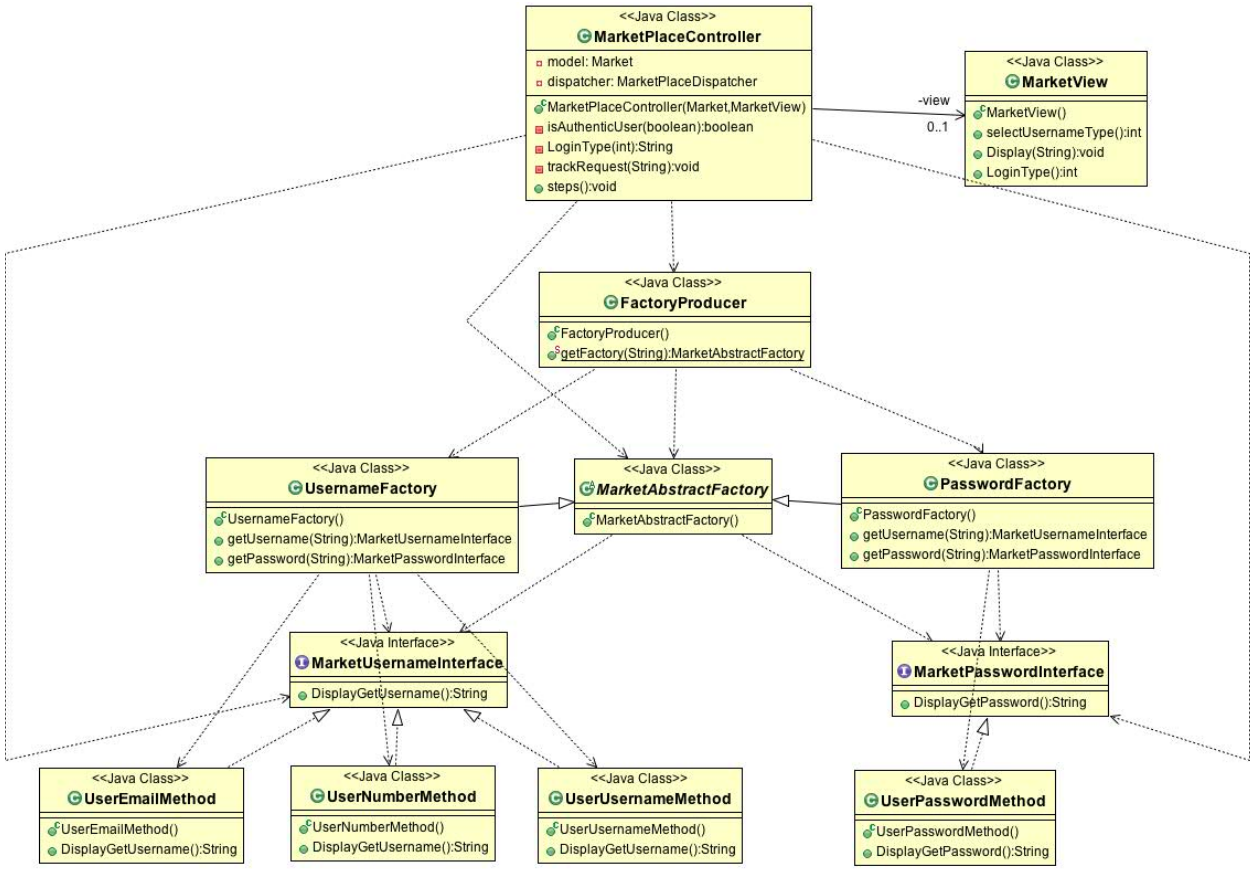


fig. 3 Abstract factory implementation

The above class diagram explains the implementation of the abstract factory pattern. I have implemented the abstract factory on the get input side because now-a-days we can login using username or email or registered mobile number and there would be many different ways in the future through which we can login, and there would be different types in which we can enter password or have biometric authentication as we have on our phones. So using abstract factory pattern here would help us in adding the different methods easily in the future and also provides consistency among the kind of login and password we would use.

I would also like to implement the Factory design pattern in the dispatcher part of the application in the future by creating the customer factory and admin factory where they would invoke the methods in the concrete class patterns.

Command Design Pattern:

I have implemented the command pattern after the dispatcher part as shown in the fig. 2. Here dispatcher gets the command from the admin/customer view about the method to be executed then dispatcher sends it the invoker, then invoker executes the corresponding requested method

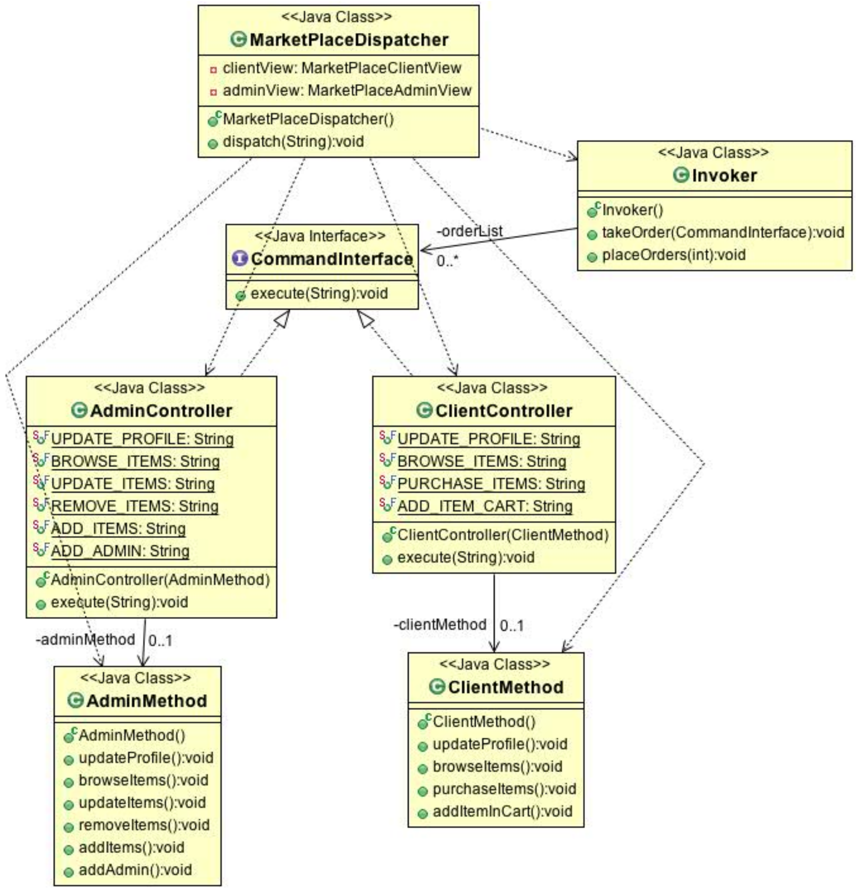
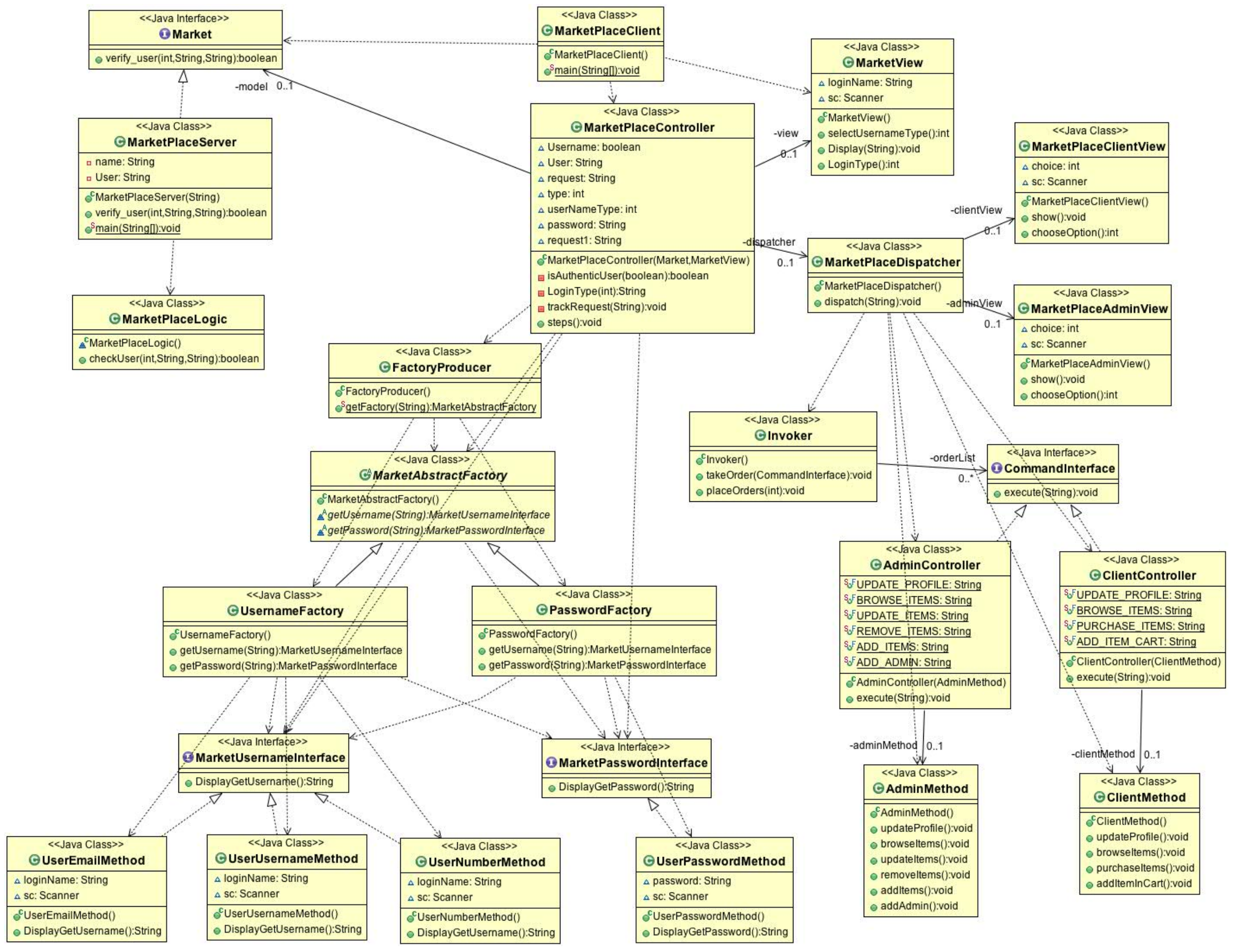


fig. 5 Class Diagram of command pattern design

from the receiver (i.e. AdminMethod and ClientMethod) by sending it to the respected client/admin controller. The above class diagram explains the working of the command pattern implemented.

The methods in the AdminMethod and ClientMethod are same as the methods defined in the domain model from assignment 1. These methods are not yet defined they only display a simple message when called.

Class Diagram :

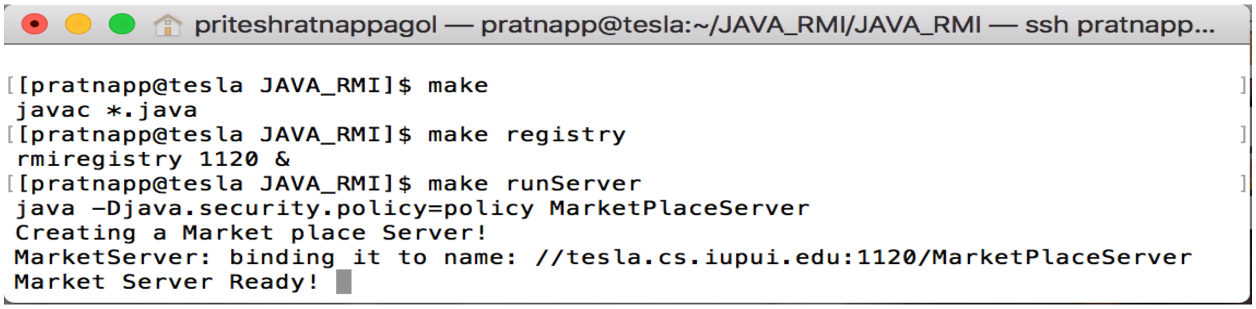


<<Java Interface>> Market
verify_user(int,String,String):boolean
<<Java Class>> MarketPlaceClient
MarketPlaceClient()
main(String[]):void
<<Java Class>> MarketView
loginName: String
sc: Scanner
MarketView()
selectUsernameType():int
Display(String):void
LoginType():int
<<Java Class>> MarketPlaceServer
name: String
User: String
MarketPlaceServer(String)
verify_user(int,String,String):boolean
main(String[]):void
<<Java Class>> MarketPlaceController
Username: boolean
User: String
request: String
type: int
userNameType: int
password: String
request1: String
MarketPlaceController(Market,MarketView)
isAuthenticUser(boolean):boolean
LoginType(int):String
trackRequest(String):void
steps():void
-model 0..1
-view 0..1
-dispatcher 0..1
<<Java Class>> MarketPlaceClientView
choice: int
sc: Scanner
MarketPlaceClientView()
show():void
chooseOption():int
-clientView 0..1
<<Java Class>> MarketPlaceDispatcher
MarketPlaceDispatcher()
dispatch(String):void
-adminView 0..1
<<Java Class>> MarketPlaceAdminView
choice: int
sc: Scanner
MarketPlaceAdminView()
show():void
chooseOption():int
<<Java Class>> MarketPlaceLogic
MarketPlaceLogic()
checkUser(int,String,String):boolean
<<Java Class>> FactoryProducer
FactoryProducer()
getFactory(String):MarketAbstractFactory
<<Java Class>> Invoker
Invoker()
takeOrder(CommandInterface):void
placeOrders(int):void
-orderList 0..*
<<Java Interface>> CommandInterface
execute(String):void
<<Java Class>> MarketAbstractFactory
MarketAbstractFactory()
getUsername(String):MarketUsernameInterface
getPassword(String):MarketPasswordInterface
<<Java Class>> UsernameFactory
UsernameFactory()
getUsername(String):MarketUsernameInterface
getPassword(String):MarketPasswordInterface
<<Java Class>> PasswordFactory
PasswordFactory()
getUsername(String):MarketUsernameInterface
getPassword(String):MarketPasswordInterface
<<Java Class>> AdminController
UPDATE_PROFILE: String
BROWSE_ITEMS: String
UPDATE_ITEMS: String
REMOVE_ITEMS: String
ADD_ITEMS: String
ADD_ADMIN: String
AdminController(AdminMethod)
execute(String):void
<<Java Class>> ClientController
UPDATE_PROFILE: String
BROWSE_ITEMS: String
PURCHASE_ITEMS: String
ADD_ITEM_CART: String
ClientController(ClientMethod)
execute(String):void
<<Java Interface>> MarketUsernameInterface
DisplayGetUsername():String
<<Java Interface>> MarketPasswordInterface
DisplayGetPassword():String
-adminMethod 0..1
-clientMethod 0..1
<<Java Class>> AdminMethod
AdminMethod()
updateProfile():void
browseItems():void
updateItems():void
removeItems():void
addItems():void
addAdmin():void
<<Java Class>> ClientMethod
ClientMethod()
updateProfile():void
browseItems():void
purchaseItems():void
addItemInCart():void
<<Java Class>> UserEmailMethod
loginName: String
sc: Scanner
UserEmailMethod()
DisplayGetUsername():String
<<Java Class>> UserUsernameMethod
loginName: String
sc: Scanner
UserUsernameMethod()
DisplayGetUsername():String
<<Java Class>> UserNumberMethod
loginName: String
sc: Scanner
UserNumberMethod()
DisplayGetUsername():String
<<Java Class>> UserPasswordMethod
password: String
sc: Scanner
UserPasswordMethod()
DisplayGetPassword():String

fig. 6. Class diagram for the overall application implemented

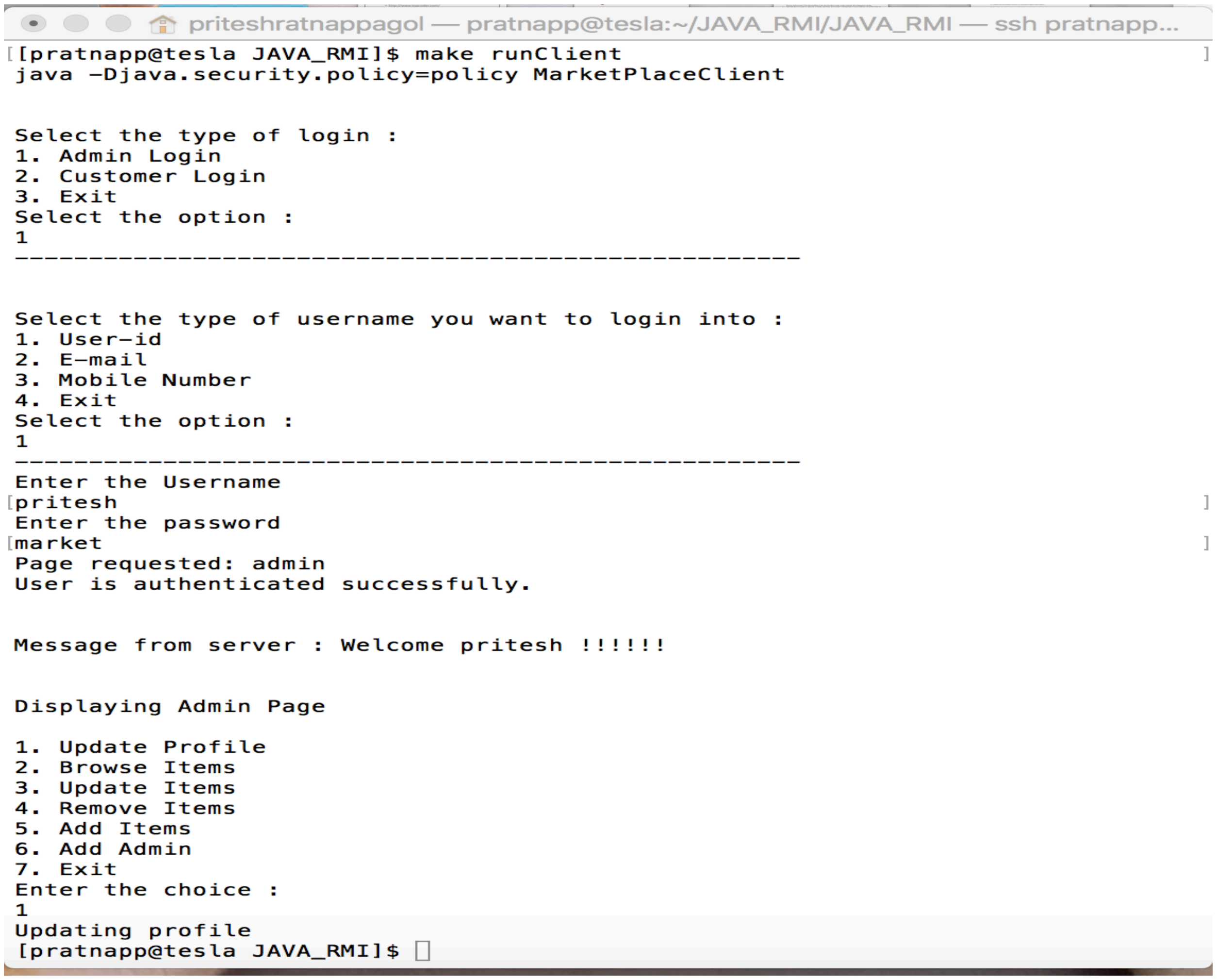
The above class diagram shows all the implemented class and their relations.

Screenhots:



```
priteshratnappagol — pratnapp@tesla:~/JAVA_RMI/JAVA_RMI — ssh pratnapp...

[pratnapp@tesla JAVA_RMI]$ make
javac *.java
[pratnapp@tesla JAVA_RMI]$ make registry
rmiregistry 1120 &
[pratnapp@tesla JAVA_RMI]$ make runServer
java -Djava.security.policy=policy MarketPlaceServer
Creating a Market place Server!
MarketServer: binding it to name: //tesla.cs.iupui.edu:1120/MarketPlaceServer
Market Server Ready!
```

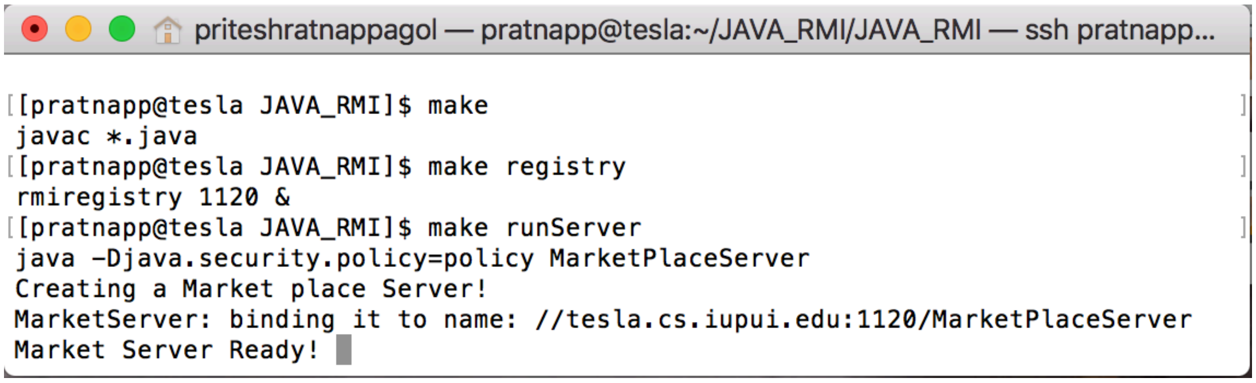
Screenshot 1. Server running



```
priteshratnappagol — pratnapp@tesla:~/JAVA_RMI/JAVA_RMI — ssh pratnapp...
[pratnapp@tesla JAVA_RMI]$ make runClient
java -Djava.security.policy=policy MarketPlaceClient


Select the type of login :
1. Admin Login
2. Customer Login
3. Exit
Select the option :
1
-----------------------------------------------------


Select the type of username you want to login into :
1. User-id
2. E-mail
3. Mobile Number
4. Exit
Select the option :
1
-----------------------------------------------------
Enter the Username
pritesh
Enter the password
market
Page requested: admin
User is authenticated successfully.


Message from server : Welcome pritesh !!!!!!


Displaying Admin Page

1. Update Profile
2. Browse Items
3. Update Items
4. Remove Items
5. Add Items
6. Add Admin
7. Exit
Enter the choice :
1
Updating profile
[pratnapp@tesla JAVA_RMI]$
```

Screenshot 2 : Instance of login

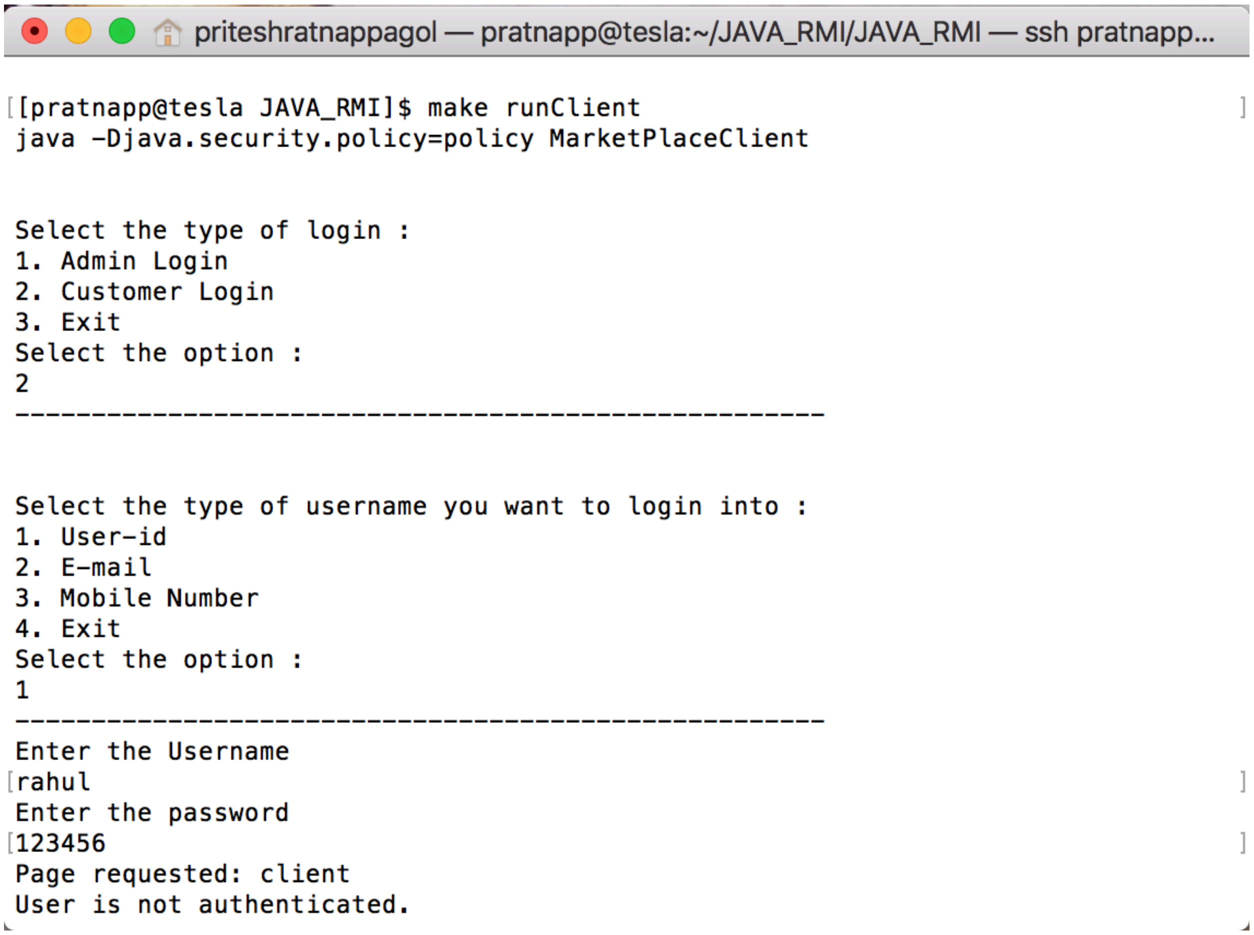
The above instance is running for admin, same message is passed for customer to if he is authenticated.



priteshratnappagol — pratnapp@tesla:~/JAVA_RMI/JAVA_RMI — ssh pratnapp...

```
[pratnapp@tesla JAVA_RMI]$ make
javac *.java
[pratnapp@tesla JAVA_RMI]$ make registry
rmiregistry 1120 &
[pratnapp@tesla JAVA_RMI]$ make runServer
java -Djava.security.policy=policy MarketPlaceServer
Creating a Market place Server!
MarketServer: binding it to name: //tesla.cs.iupui.edu:1120/MarketPlaceServer
Market Server Ready!
```

Screenshot 3. Instance of server running



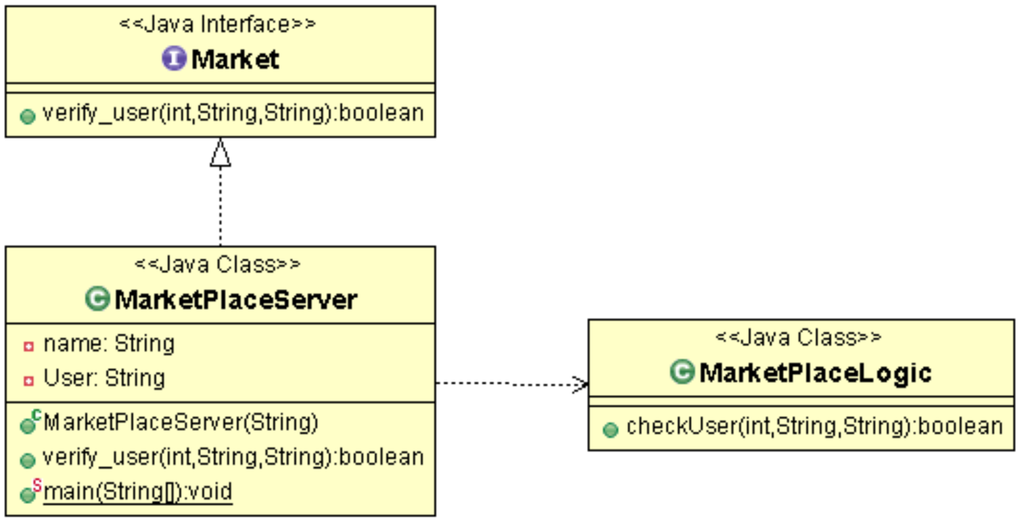
priteshratnappagol — pratnapp@tesla:~/JAVA_RMI/JAVA_RMI — ssh pratnapp...

```
[pratnapp@tesla JAVA_RMI]$ make runClient
java -Djava.security.policy=policy MarketPlaceClient


Select the type of login :
1. Admin Login
2. Customer Login
3. Exit
Select the option :
2
-----------------------------------------------------


Select the type of username you want to login into :
1. User-id
2. E-mail
3. Mobile Number
4. Exit
Select the option :
1
-----------------------------------------------------
Enter the Username
rahul
Enter the password
123456
Page requested: client
User is not authenticated.
```

Screenshot 4. Instance of user not authenticated

The above instance is running for customer, same message is passed for admin to if he is not authenticated.
Corrections in Assignment 1:

1) In assignment 1 I have mixed application logic with the framework functionality at the server side which did not ensure highly cohesive application.
   So to make it highly cohesive application I have created a different class called logic which is invoked by the server main class by creating the object of it.



<<Java Interface>>
Market
verify_user(int,String,String):boolean
<<Java Class>>
MarketPlaceServer
name: String
User: String
MarketPlaceServer(String)
verify_user(int,String,String):boolean
main(String[]):void
<<Java Class>>
MarketPlaceLogic
checkUser(int,String,String):boolean

fig 7. Class diagram for server

   This MarketPlaceLogic class has the method in it which authenticates the user when called by the server main method with the parameters passed from the client side. Fig 7. Shows the corrected implementation using class diagram.

2) In domain model I have not added any new classes right now as I didn't feel any important class missing as per the 1st assignment but would make changes in the future if any change in requirements.

Conclusion:

By implementing these different design patterns it made me realize that it makes the programming more well structured and can be managed easily. Even it is helpful when we want to add new features or delete any existing feature easily without affecting the while system.

Reference:

1] Front Controller Design :
http://www.oracle.com/technetwork/java/frontcontroller-135648.html

2] Abstract Factory Pattern :
https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm

3] Command Pattern :
https://dzone.com/articles/design-patterns-command

4] Class notes – Prof. Ryan Rybarczyk