ONLINE MARKETPLACE

Assignment 2

Samira Khorshidi

TABLE OF CONTENTS

ASSIGNMENT DISCUSSION

The purpose of this practice is designing Domain Model of online Marketplace using Java RMI and MVC design pattern and by leveraging useful and proper design patterns.

FRONT CONTROLLER PATTERN

The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. Following are the entities of this type of design pattern.

- *Front Controller - Single handler for all kinds of requests coming to the application (either web based/ desktop based).*
- *Dispatcher - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler.*
- *View - Views are the object for which the requests are made.*

In my experience, all of my requests from my clients come to Front Controller class and using Dispatcher, they go through proper view, either Administrator View or Customer View.

The request for authenticating (Login) user will be created by Front Controller.

COMMAND PATTERN

The Command pattern is a data driven design pattern and falls under behavioral pattern category. In this pattern, a request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

In this assignment, I used this design pattern in the process of user Login.

To implement this pattern, I supposed user (Administrator/Customer) as a request, Login action as a command that needs to be done. The front controller send login request via RMI to the Server controller and Server Controller uses Invoker object to login both type of users.

ABSTRACT FACTORY PATTERN

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

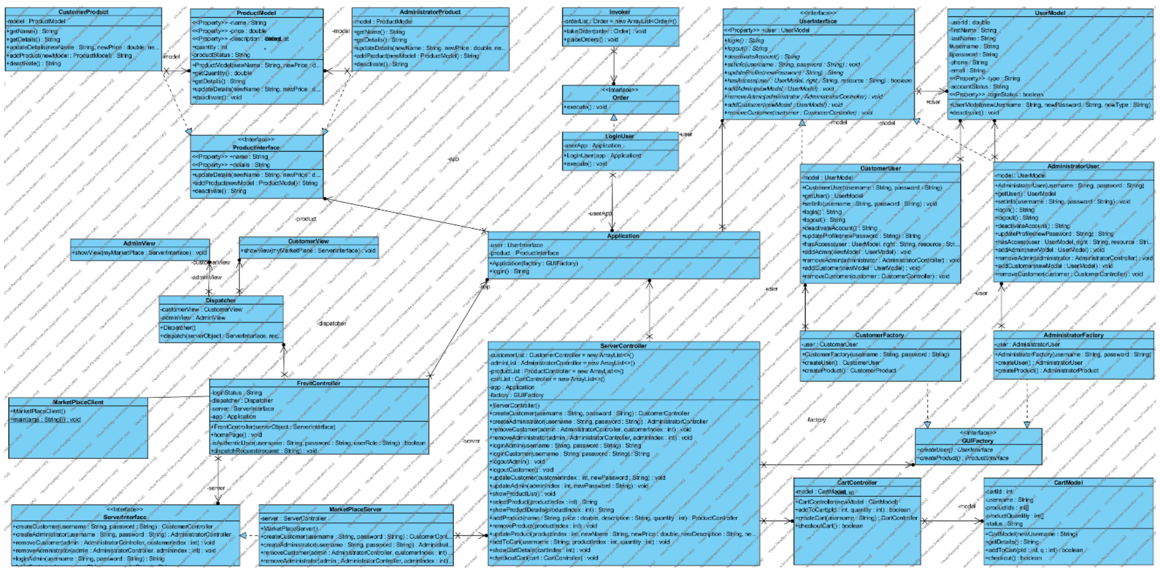
In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

In this assignment, my two families of objects (factory products) are user entity and product entity. I used this pattern to generate proper GUI for each type of users, so I have Administrator GUI and Customer GUI which each of the has their own user, Customer and Administrator, with their valid actions on products in our MarketPlace.

FIGURE, SAMPLE CODES

In following we provide you a Class Diagram of our system in this assignment along with some sample codes.

UML DIAGRAM



The original UML diagram has been provided in Documentation directory. As you see, in our proposed architecture at this step, Controller Components are divided to new classes which together act as old Controllers in server side to obtain high cohesion system design

and keep business logics and data separately related to view and Client component of our system. In this system, business logics and models can change frequently while view components can remain safe in term of affected by these changes.
Also, the rule of RMI interfaces is just providing remote object and functionalities are separate controllers.

SAMPLE CODES

In following we provide a sample code of our models, controllers and server Interface to the client. Models are used by Controllers and Server Interface acts as an Interface between Clients which are views and Controllers:

- *Model:*

```
package onlineMarket;

public class UserModel {

	private double userId;
	private String firstName;
	private String lastName;
	protected String username;
	protected String password;
	private String phone;
	private String email;
	private String type; // admin, customer
	private String accountStatus; // active, deactive
	private boolean loginStatus;

	public UserModel(String newUsername, String newPassword, String newType)
{
		username = newPassword;
		password = newUsername;
		type = newType;
		loginStatus = true;
		accountStatus = "active";
	}

	/**
	 * Responsible to login and logout the user
	 *
	 * @param status
	 */
	public void setLoginStatus(boolean status) {
		loginStatus = status;
	}
```

- *AdministratorFactory:*

```
public class AdministratorFactory implements GUIFactory {
      private AdministratorUser user;

      public AdministratorFactory(String username, String password){
            this.user = new AdministratorUser(username, password);

      }


    @Override
    public AdministratorUser createUser() {
        return user;
    }

    @Override
    public AdministratorProduct createProduct() {
        return new AdministratorProduct();
    }
}
```

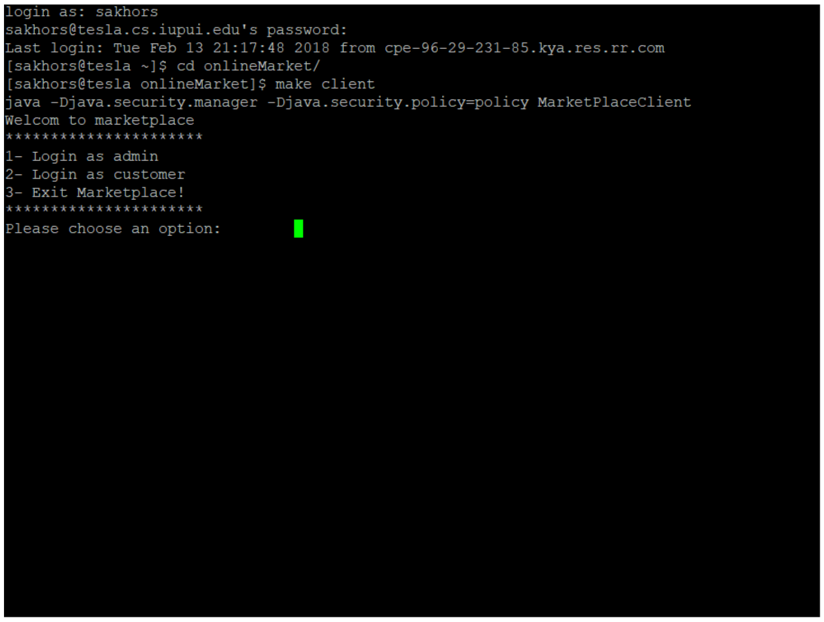
- *Remote Method:*

```
public synchronized String loginAdmin(String username, String password) throws
RemoteException {

            return server.loginAdmin(username, password);

      }
```

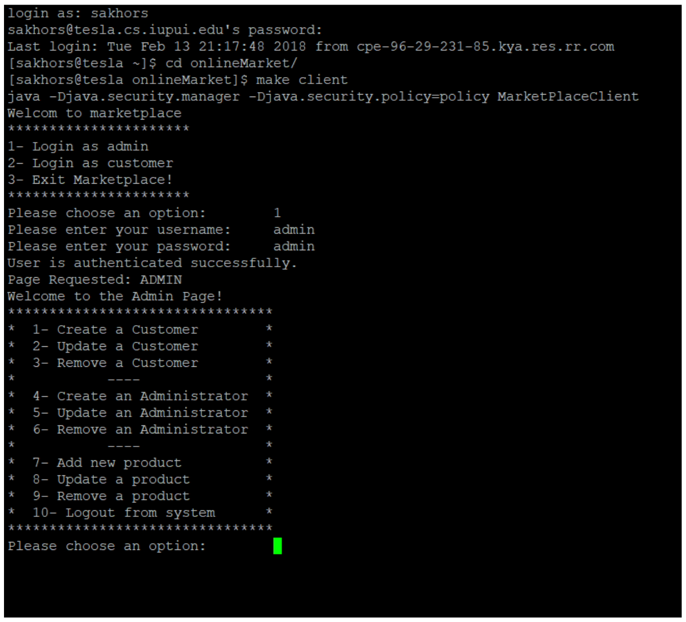
SAMPLE RUNS

In this section you’ll see screenshots of sample runs. The system is minimal, so there is no GUI, instead every interaction happen using command prompt.

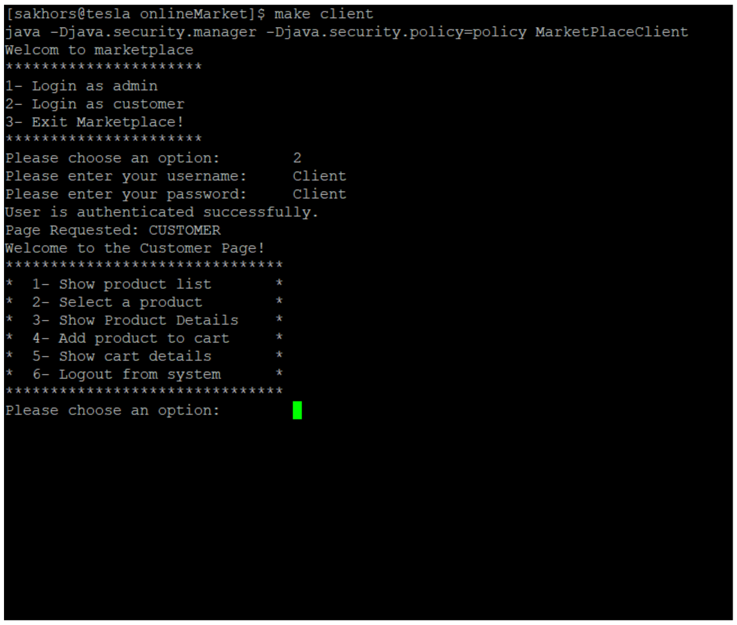
SCREENSHOTS



```
login as: sakhors
sakhors@tesla.cs.iupui.edu's password:
Last login: Tue Feb 13 21:17:48 2018 from cpe-96-29-231-85.kya.res.rr.com
[sakhors@tesla ~]$ cd onlineMarket/
[sakhors@tesla onlineMarket]$ make client
java -Djava.security.manager -Djava.security.policy=policy MarketPlaceClient
Welcom to marketplace
**********************
1- Login as admin
2- Login as customer
3- Exit Marketplace!
**********************
Please choose an option:
```



```
login as: sakhors
sakhors@tesla.cs.iupui.edu's password:
Last login: Tue Feb 13 21:17:48 2018 from cpe-96-29-231-85.kya.res.rr.com
[sakhors@tesla ~]$ cd onlineMarket/
[sakhors@tesla onlineMarket]$ make client
java -Djava.security.manager -Djava.security.policy=policy MarketPlaceClient
Welcom to marketplace
**********************
1- Login as admin
2- Login as customer
3- Exit Marketplace!
**********************
Please choose an option:        1
Please enter your username:     admin
Please enter your password:     admin
User is authenticated successfully.
Page Requested: ADMIN
Welcome to the Admin Page!
********************************
*  1- Create a Customer        *
*  2- Update a Customer        *
*  3- Remove a Customer        *
*          ----                *
*  4- Create an Administrator  *
*  5- Update an Administrator  *
*  6- Remove an Administrator  *
*          ----                *
*  7- Add new product          *
*  8- Update a product         *
*  9- Remove a product         *
*  10- Logout from system      *
********************************
Please choose an option:
```



```
[sakhors@tesla onlineMarket]$ make client
java -Djava.security.manager -Djava.security.policy=policy MarketPlaceClient
Welcom to marketplace
**********************
1- Login as admin
2- Login as customer
3- Exit Marketplace!
**********************
Please choose an option:        2
Please enter your username:     Client
Please enter your password:     Client
User is authenticated successfully.
Page Requested: CUSTOMER
Welcome to the Customer Page!
*******************************
*  1- Show product list       *
*  2- Select a product        *
*  3- Show Product Details    *
*  4- Add product to cart     *
*  5- Show cart details       *
*  6- Logout from system      *
*******************************
Please choose an option:
```

DISCUSSION

The proposed system has not complete functionalities, it means Login is using mockup data and other functionalities needs to be completed in the future.

CONCLUTION
REFERENCES

CONCLUTION

In conclusions, using thre different design pattern in this assignment, I see them useful in some aspects but still I see the disadvantages of command pattern this this particular case(Login), since it is useful when creating a structure, particularly when the creating of a request and executing are not dependent on each other. It means that the **Command** instance can be instantiated by **Client**, but run sometime later by the **Invoker**, and the **Client** and **Invoker** may not know anything about each other and I couldn't address this feature in my code readily.

However, command pattern helps in terms of extensibility as we can add a new command without changing the existing code.

And about Abstract Factory, I see its pro's very clear since it allows me to hide implementation of an application seam (the core interfaces that make up my application) and also lets me easily test the seam of an application (that is to mock/stub) certain parts of my application so I can build and test the other parts. In addition, this pattern lets me change the design of my application more readily, which is known as loose coupling.

REFERENCES

- *https://www.tutorialspoint.com/design_pattern/front_controller_pattern.htm*
- *https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm*
- *https://www.tutorialspoint.com/design_pattern/command_pattern.htm*