



ONLINE MARKETPLACE

Assignment 3 Report



MARCH 10, 2017

ZHIHAO CAO
caozh@iupui.edu

Designs

Domain Model

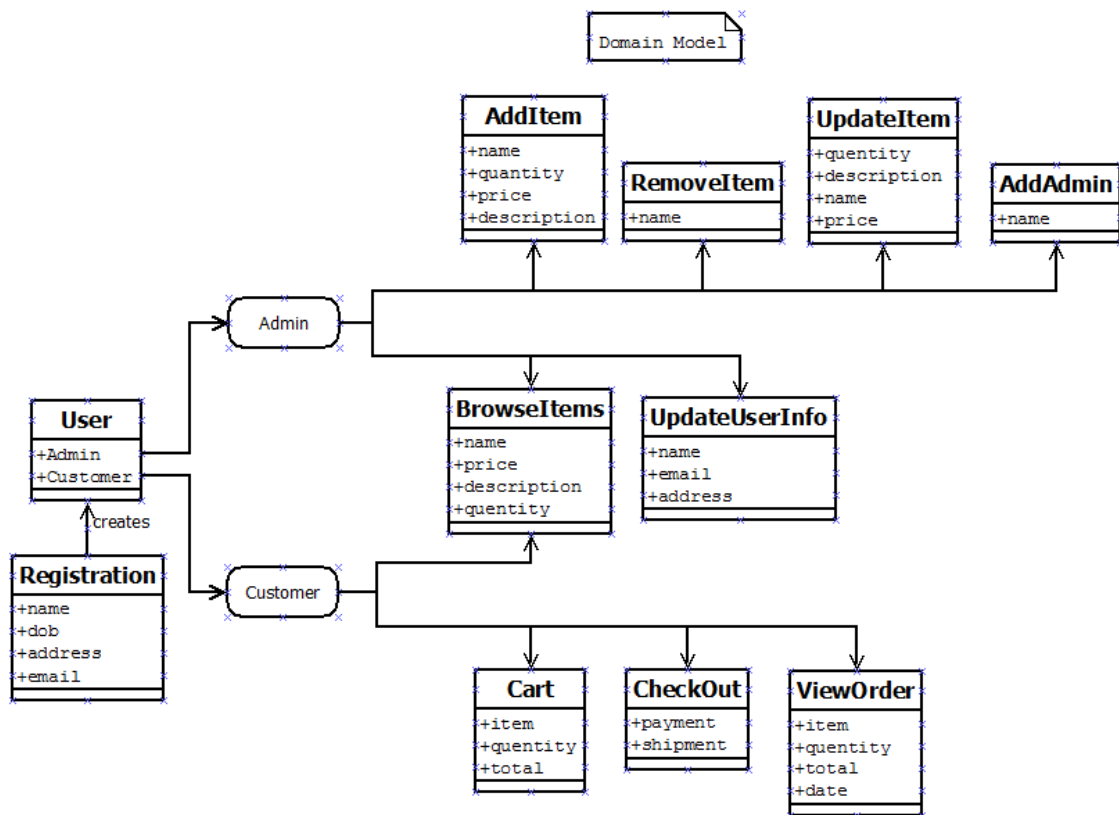
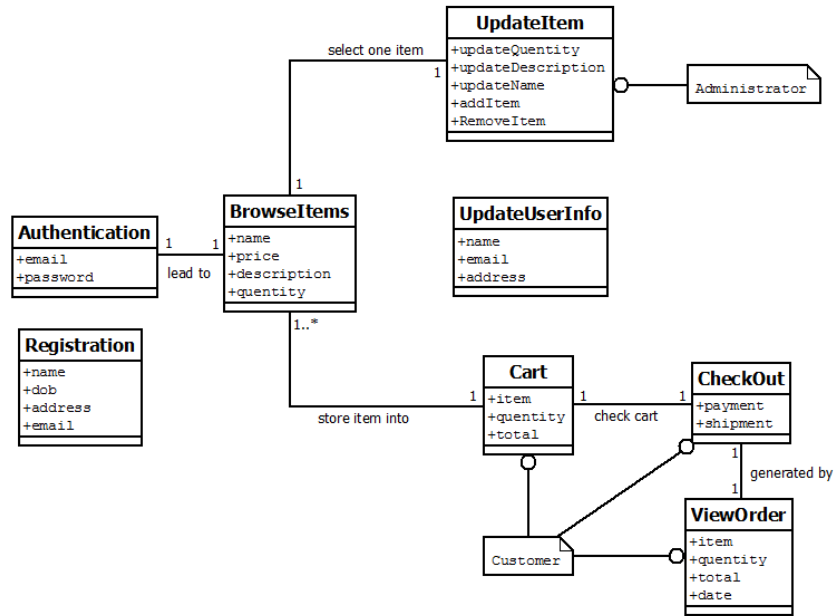


Figure 1: Domain model v.2 (top) and Domain model v.3 (bottom)

#assignment 1

As the assignment 1 instruction indicated, I am going to develop a client-server architecture system that utilizes JAVA RMI technology. Since this system is an online marketplace, there are some famous online shopping websites coming up on my mind. I analyze what components are needed to build up such a system based on designs of Amazon, Newegg, and eBay. I finally generalize my thoughts into a domain model.

Other than thinking about the functionality of the system, to fulfill the expectation of this course, I also need to design the system based on Object Oriented design concept. For this project, MVC pattern is suggested. In next page, the initial draft of the MVC design is shown.

#assignment 2

The bottom diagram in figure 1 shows the updated Domain model. It is revised to be more focused on the abstract functionalities that offered by the marketplace system. The database-objects and logging component are removed. The abstract UpdateItem object includes any update actions regarding to items.

#assignment 3

Comparing to the earlier version, the domain model maintains better clarity of the functionalities separation respecting to two types of role (admin and customer). Some concrete functions are isolated from “UpdateItem” conceptual class, such as “AddItem”, “RemoveItem”, and “AddAdmin”. The point on redesigning the domain model in this way is that, by a glance, clients can obtain a good understanding of the separated role design of the system (admin and customer), and the respective functionalities they concern.

MVC Design

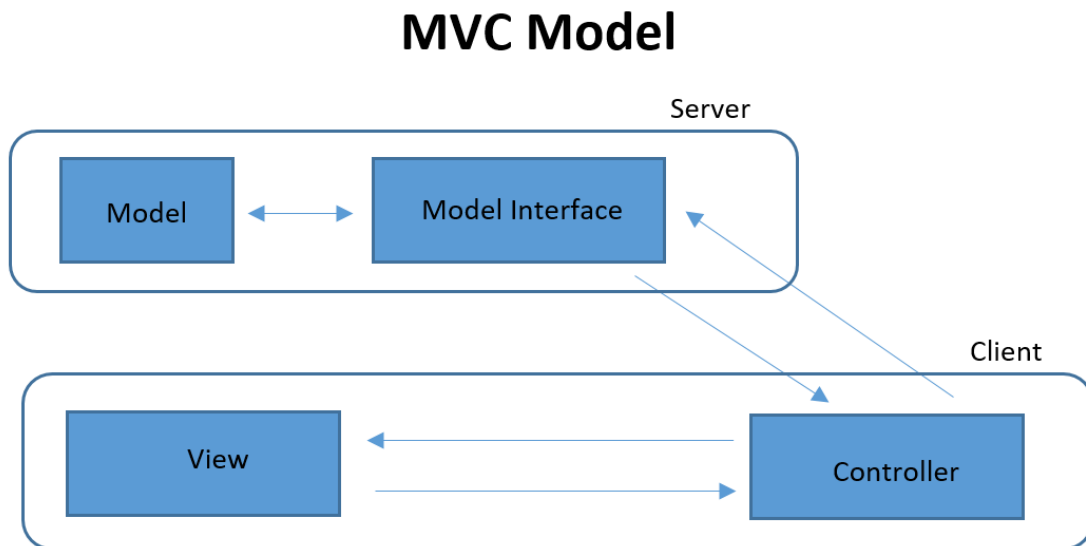


Figure 2: MVC Model

#assignment 1

The model part is located in Server side. The View and Controller parts are located in Clients side. The Server creates remote model object by applying JAVA RMI, and register the object into Server. In the Client side, the connection is initiated by looking up the remote model object, then creates an instance of the object and use this instance to make remote invocation.

After the lecture about controller, I aware that the controller should be located in Server side. However, at the moment, I have not figure out how to perform the control functions in Server side yet. I will fix it in next assignment.

The reason why I choose the controller stay in Client side is that I think the capabilities of remote invocation offered by JAVA RMI benefit the function calling in the Client side. Imaging that the client program receives the user input at the first place and with the ability to perform methods from the remote model, the client program can perform the operations directly. I feels much more convenient than passing the inputs from Client side to Server side and has the controller on the Server side to process the parameters and return back to Client.

#assignment 2

Upon the MVC architecture that built in assignment 1, Front controller, Command, and Abstract Factory patterns are integrated into the system in this assignment.

#assignment 3

After several iterations of the development, the “macro” architecture of the system becomes more and more complex. (More concrete)

The current architecture of the system is interpreted as an UML class diagram, and I will present it in Authorization patter section.

Front Controller

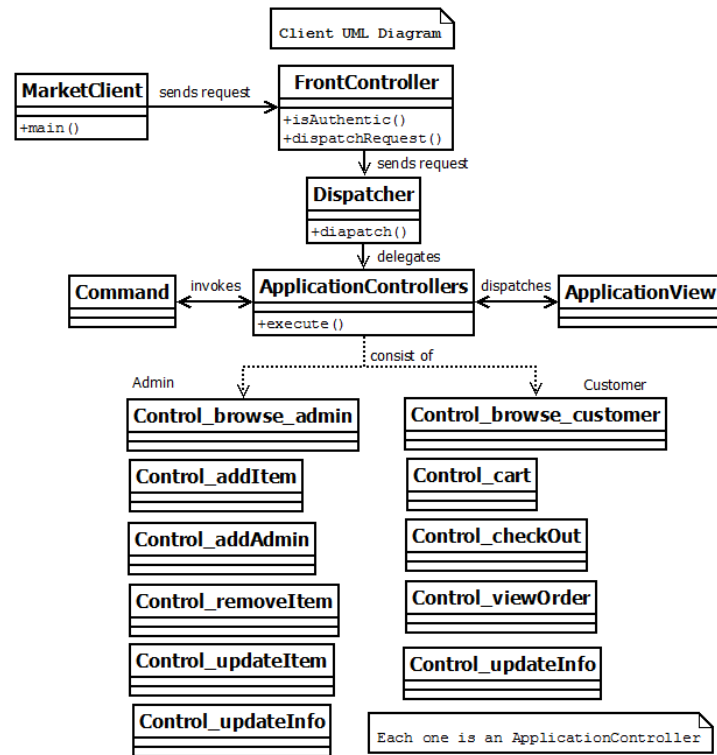


Figure 3: Implementation of Front Controller pattern

#assignment 2

Figure 3 shows how front controller pattern is applied in the Marketplace system.

1. Whenever a request is sent by a user, MarketClient sends the request to FrontController.
2. FrontContrller then performs authentication and sends the request to Dispatcher.
3. Dispatcher receives the request and delegates one of the ApplicationController (there are 11 application controllers) to do the job.
4. ApplicationController will perform the according business logic by interacting with Marketplace server (invokes Command objects) to get the result.
5. Based on the results, ApplicationController dispatches and renders a view for user.

#assignment 3

Significant modifications have been applied on the previous implementation of the front controller pattern.

- A login module is added into front controller. Now, a new connected client will be asked to provide username and password to login.
- To implement the Authorization patter, an additional controller class, Client Controller, is created. The description of this class will be presented in Authorization pattern section. Login process will be delegated to client controller.
- Based on the return information of the login process from client controller, front controller will:

- Update the Login View to display condition for invalid login
- Delegates the responsibility of forwarding the flow to the Dispatcher based upon the resulting User session object.

Abstract Factory Pattern

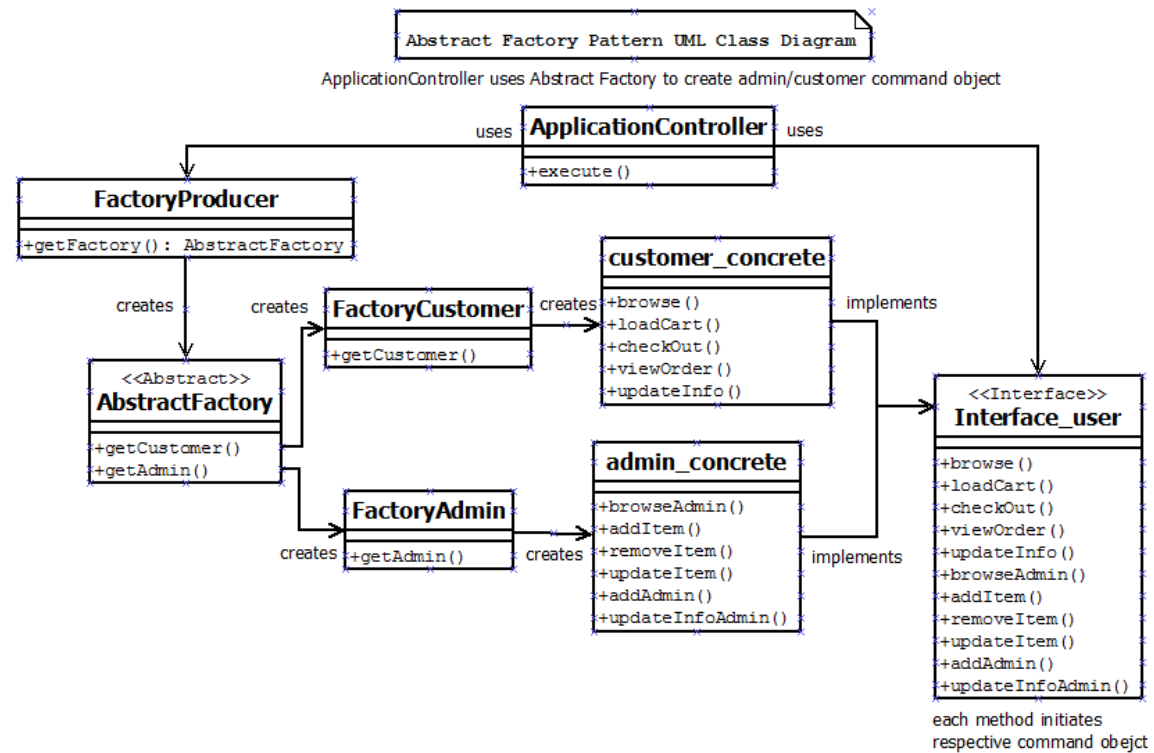
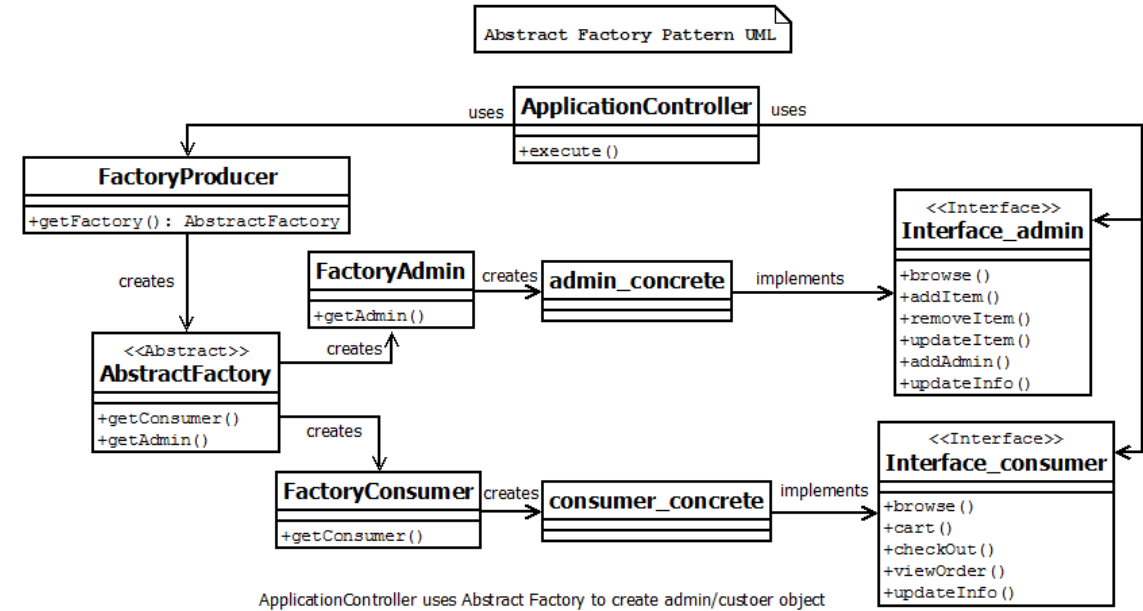


Figure 4: Implementation of Abstract Factory [version 1, version 2]

#assignment 2

This Abstract Factory is designed to produce two sub-factories: admin factory and customer factory.

- Admin factory: creates admin object.
- Customer factory: creates customer object.

The Abstract Factory is called in the ApplicationController to create needed object. Then Command Pattern will use this object as its receiver.

Based on the principle of Abstract Factory, each sub-factory should be capable of create more than one type of objects. For example, Customer factory may also create Student type customer, “Prime” customer, or “Diamond VIP” customer. For simplicity of learning how to apply the patterns, only one type of object is coded for each sub-factory.

#assignment 3

Significant modifications have been applied on the previous implementation of the Abstract factory pattern.

1. In previous version, the concrete factories used different interfaces to create concrete customer/admin objects. This way deviates the point that the factory produces families of related objects. In this assignment, two concrete factories use the same interface: “Interface_user”, as the figure shown.
2. In previous version, the methods in the interface was unreasonable to define. It makes the created user object as receiver for command objects. After carefully consideration, I decided to change the way that the abstract factory pattern and command pattern cooperates. Now, command objects are implementation of the methods from user object created by factory. As the figure 4 shown, the methods, in “customer_concrete”/”admin_concrete” class, initiate respective command objects.

```
@Override
public Command_addAdmin addAdmin(MVC_model_interface myModel) throws RemoteException {
    System.out.println("addAdmin command created");
    Command_addAdmin command_addAdmin = new Command_addAdmin(myModel);
    return command_addAdmin;
}
```

Above is a section of source code that defines “addAdmin” method in “admin_concrete” class.

Command Pattern

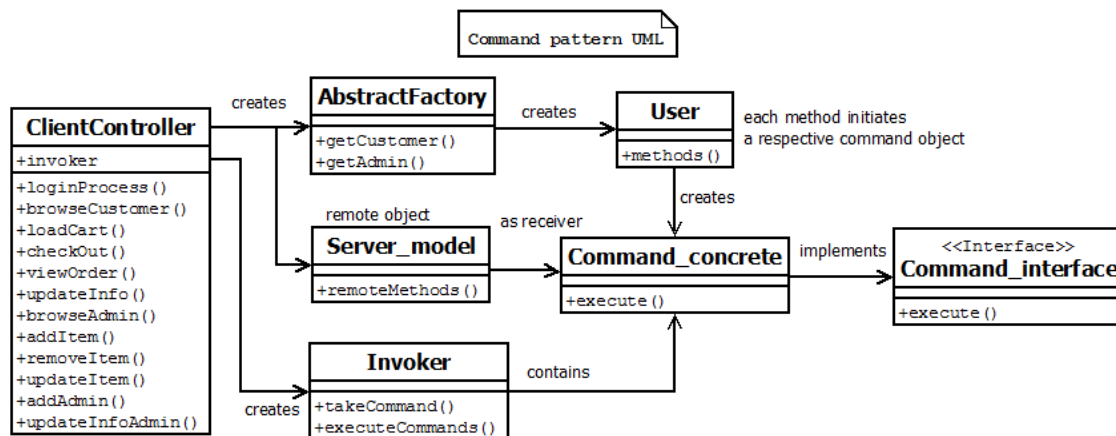
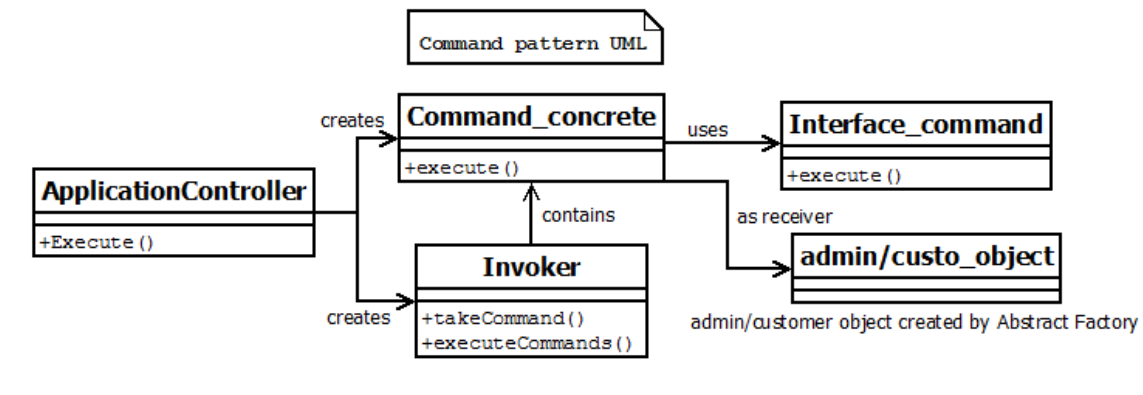


Figure 5: Implementation of Command Pattern [version 1, version 2]

#assignment 2

The goal of Command Pattern is to encapsulate the concrete methods of receivers, by offering developers abstract command objects to perform a pre-defined sequence of method calls to achieve the task.

In this Marketplace system, Command Pattern is applied inside ApplicationController.

When a receiver object is created by the Abstract Factory, ApplicationController creates necessary command objects using this receiver object, and creates invoker to perform the task. The ApplicationController gets the result, and uses it to render a view for user.

#assignment 3

Significant modifications have been applied on the previous implementation of the Command pattern.

1. As explained in Abstract factory pattern, concrete command objects now are initiated by methods in user object created by Abstract factory.

2. As the figure 5 shown, the receiver used to be a user object, which caused very unreasonable logic to implement. Now, the receiver is the remote server model object, which satisfies the need to do intended tasks inside a concrete command object.
3. Now, the Command and Abstract factory patterns are used in client controller instead of application controller.
4. Invoker is a static object in client controller class.

Authorization Pattern

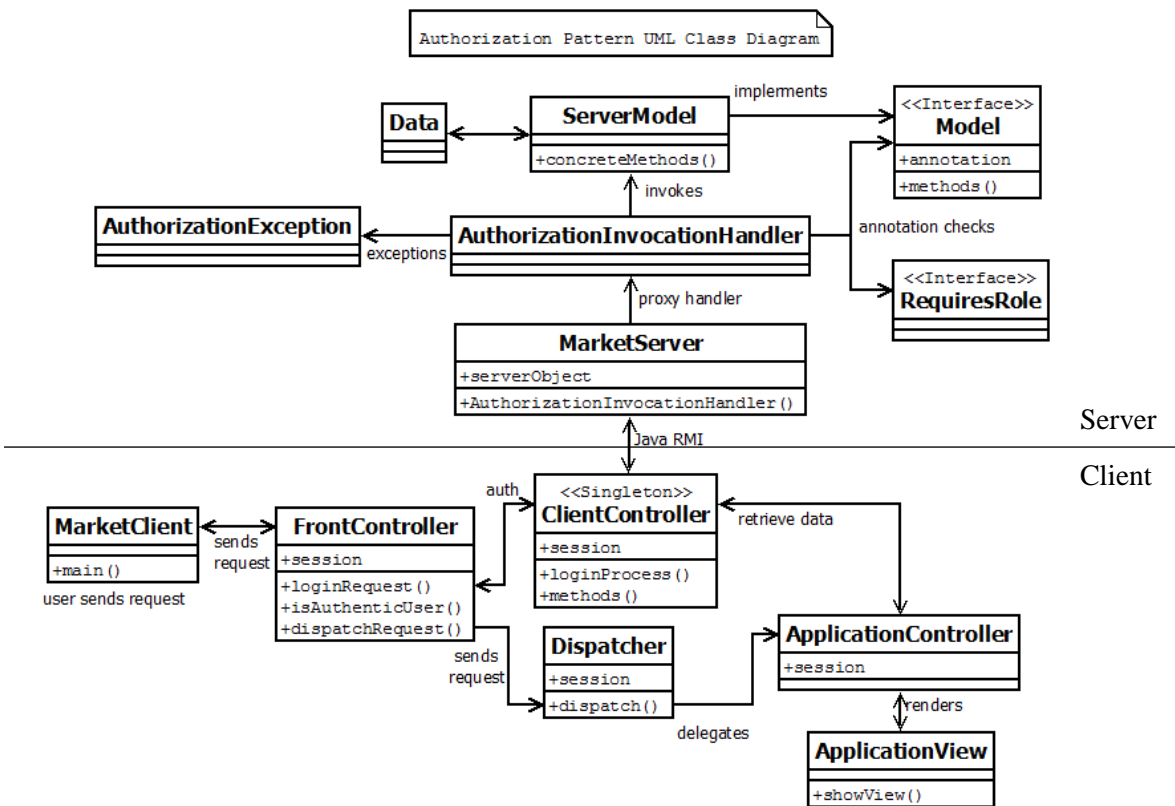


Figure 6: Implementation of Authorization Pattern

#assignment 3

The implementation of Authorization pattern across client and server sides. There are six points to enable this pattern:

1. A new controller is added: Client controller. It is a singleton class that acting as a gateway between client and server. The server RMI model object is initiated in this class, so that command objects are able to utilize it as their receivers.
2. It provides a method to perform login task: loginProcess(). It will be invoked by front controller to perform login authentication. It gets results from server and create a session object based on the returned information. (invalid login/customer login/admin login), and return the session object back to front controller.
3. It offers other functional methods to use Command objects to perform the task.

4. In server side, a proxy object is instantiated to realize the use of annotation. The proxy provides an invocation handler to check if the client of the incoming method call providing a role that matches the annotated method in the server model object. The reflection pattern is applied here to extract needed information from other classes.
5. A “RequiresRole” class is created to offer the interface for annotation.
6. Methods in server model interface (server implementation interface) are added annotations above them if Role-Based Access Control is required. As the section of code shown.

```
@RequiresRole("customer")  
    public String[] loadCart(Session session) throws RemoteException;
```

Coding design & Discussion

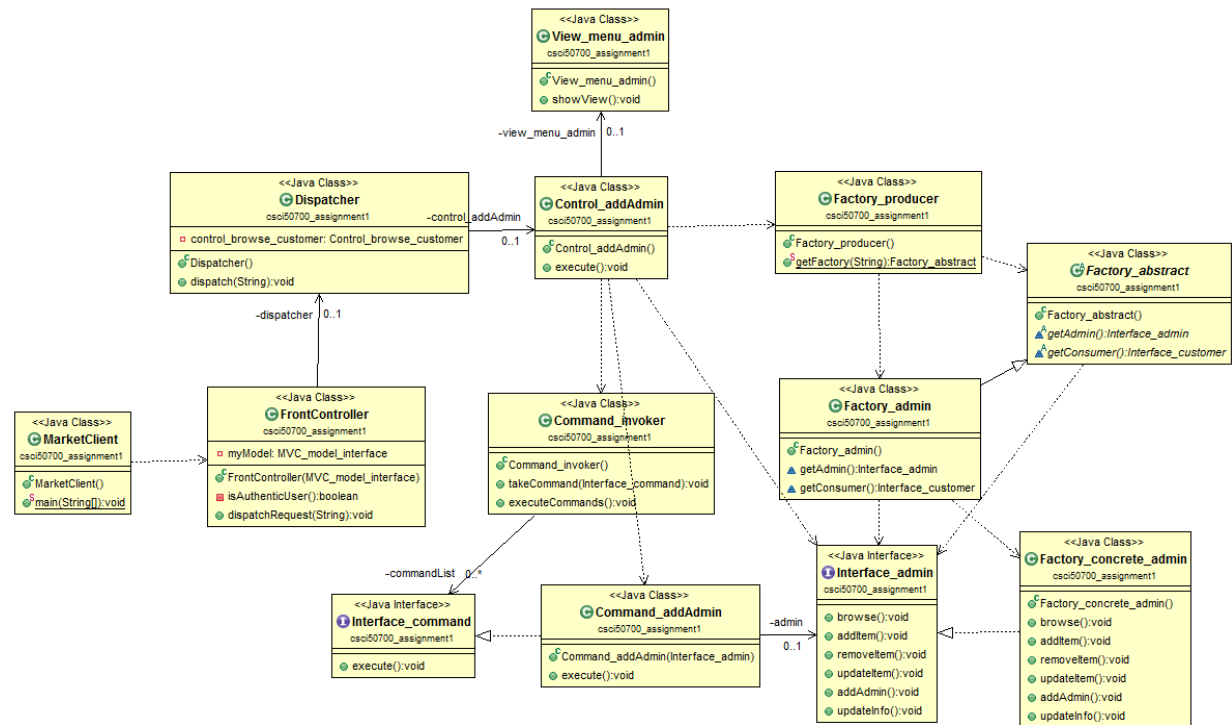


Figure 7: Updated Client Class Diagram (addAdmin request)

#assignment 2

Due to the complexity of showing all classes together, a necessary porting of them is shown here.

This diagram specifically shows the classes in the client side for addAdmin request. The flow of the program is as described in FrontController Pattern above. In addition, this diagram shows the relation between FrontController Pattern, Abstract Factory Pattern, and Command Pattern.

#assignment 3

Since the project is getting more and more complex, there are more than 50 classes. A comprehensive class diagram cannot apply. Authorization pattern UML diagram can demonstrate the overall classes in the system as last section shown.

Classes and Discussion:

MarketServer.java

#assignment 2

```
public static void main(String[] args) {
    // Java RMI Security Manager
    System.setSecurityManager(new SecurityManager());

    try{
        System.out.println("Creating a Online Market Server!");
        String name = "///tesla.cs.iupui.edu:1888/MarketServer";

        // Create a new instance of a Market model.
        MVC_model_interface myModel = new MVC_model();

        System.out.println("Marketplace Server: binding it to name: " + name);

        // Binding of the newly created Bank server instance to the RMI Lookup.
        Naming.rebind(name, myModel);
        System.out.println("Marketplace Server Ready!");
    } catch (Exception e){
        System.out.println("Exception: " + e.getMessage());
        e.printStackTrace();
    }
}
```

It is a server side class that contains main() method to starts the server which offers a remote object.

#assignment 3

```
public class MarketServer {

    public static void main(String[] args) {
        // Java RMI Security Manager
        System.setSecurityManager(new SecurityManager());
        try{
            System.out.println("Creating a Online Market Server!");
            String name = "///tesla.cs.iupui.edu:1888/MarketServer";

            // Create a proxy and put model object into it
            MVC_model_interface myProxy = (MVC_model_interface)
                Proxy.newProxyInstance(MVC_model_interface.class.getClassLoader(),
                    new Class<?>[] {MVC_model_interface.class},
                    new AuthorizationInvocationHandler(new MVC_model()));

            // Binding of the newly created Bank server instance to the RMI Lookup.
            System.out.println("Marketplace Server: binding it to name: " + name);
            Naming.rebind(name, myProxy);
            System.out.println("Marketplace Server Ready!");
        } catch (Exception e){
            System.out.println("Server RMI Exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

To realize Authorization pattern, the use of annotation is needed. To enable annotation, reflection and proxy are needed. In the main method of server class, a proxy is created such that an invocation handler is provided to check roles before invoking methods are executed.

AuthorizationInvocationHandler.java

```
public class AuthorizationInvocationHandler implements InvocationHandler, Serializable {
    private static final long serialVersionUID = 6925780928377938176L;
    private Object objectImpl;

    public AuthorizationInvocationHandler(Object impl) {
        this.objectImpl = impl;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //check if the target method require role
        if (method.isAnnotationPresent(RequiresRole.class)) {
            RequiresRole accessControl = method.getAnnotation(RequiresRole.class);
            Session session = (Session) args[0];
            //check session role type with Access control
            System.out.println("annotation: " + accessControl.toString());
            System.out.println("session: " + session.getUser().getRoleType());
            if (session.getUser().getRoleType().equals(accessControl.value())) {
                return method.invoke(objectImpl, args);
            }
        } else {
            throw new AuthorizationException(method.getName());
        }
    } else {
        System.out.println("AuthorizationInvocationHandler: no annotation");
        return method.invoke(objectImpl, args);
    }
}
}
```

The AuthorizationInvocationhandler class implements the handler of the proxy. It checks the role type annotated on a method of the model interface class with the role type provided by the client. If fails the annotation check, an AuthorizationException class will be instantiated and returned.

AuthorizationException.java

#assignment 3

```
public class AuthorizationException extends RuntimeException {
    private static final long serialVersionUID = 5528415690278423524L;

    public AuthorizationException(String methodName) {
        super("Invalid Authorization - Access Denied to "
            + methodName + "() function!");
    }
}
```

The handler inform the user that the access denied.

MVC_model_interface.java

#assignment3

```
public interface MVC_model_interface extends Remote {
    public String processLogin(Credential credential) throws RemoteException;

    //customer
    @RequiresRole("customer")
    public String[] browse(Session session) throws RemoteException;
    @RequiresRole("customer")
    public String[] loadCart(Session session) throws RemoteException;
    @RequiresRole("customer")
    public String[] checkout(Session session) throws RemoteException;
    @RequiresRole("customer")
    public String[] viewOrder(Session session) throws RemoteException;
    @RequiresRole("customer")
    public String[] updateInfo(Session session) throws RemoteException;
}
```

```

//administrator
@RequiresRole("admin")
public String[] browseAdmin(Session session) throws RemoteException;
@RequiresRole("admin")
public String[] addItem(Session session) throws RemoteException;
@RequiresRole("admin")
public String[] removeItem(Session session) throws RemoteException;
@RequiresRole("admin")
public String[] updateItem(Session session) throws RemoteException;
@RequiresRole("admin")
public String[] addAdmin(Session session) throws RemoteException;
@RequiresRole("admin")
public String[] updateInfoAdmin(Session session) throws RemoteException;
}

```

Methods in server model object are annotated with a role based on the logic of the method respecting to.

MVC_model.java

#assignment 3

```

public class MVC_model extends UnicastRemoteObject implements MVC_model_interface{
    private static final long serialVersionUID = 1L;

    //constructor to create remote object
    protected MVC_model() throws RemoteException {
        super();
    }

    public String processLogin(Credential credential) {
        String result = null;
        System.out.println("input username is:" + credential.getUserName());
        System.out.println("input password is:" + String.valueOf(credential.getPwd()));
        if (credential.getUserName().equalsIgnoreCase("josephzelo")){
            System.out.println("username is correct");
            if(String.valueOf(credential.getPwd()).equalsIgnoreCase("asdasd")){
                result = "customer";
            } else {
                System.out.println("password is wrong");
            }
        } else if (credential.getUserName().equalsIgnoreCase("admin")) {
            System.out.println("username is correct");
            if(String.valueOf(credential.getPwd()).equalsIgnoreCase("admin")){
                result = "admin";
            } else {
                System.out.println("password is wrong");
            }
        } else {
            System.out.println("username is wrong");
        }
        return result;
    }

    /**
     * It allows the admin to add an admin account
     */
    public String[] addAdmin(Session session) throws RemoteException {
        System.out.println("Server model invokes addAdmin() method");
        return null;
    }
}

```

Due to the length of code in “MVC_model.java”, only two methods are kept. Other classes are similar to the structure of addAdmin() method.

The processLogin() method takes credential object (contains username and password) as input, validates the combination, and returns a String as result (customer/admin/null).

MarketClient.java

#assignment 2

```
public static void main(String[] args) {  
    // Java RMI Security Manager  
    System.setSecurityManager(new SecurityManager());  
  
    // Try-Catch is necessary for remote exceptions.  
    try {  
        //look up the remote object  
        String name = "///tesla.cs.iupui.edu:1888/MarketServer";  
        MVC_model_interface myModel = (MVC_model_interface) Naming.lookup(name);  
  
        // Create new instance of a Front Controller...  
        FrontController frontController = new FrontController(myModel);  
  
        //client start page  
        View_welcome view_welcome = new View_welcome();  
        view_welcome.showView();  
  
        //receive request  
        Scanner scanner=new Scanner(System.in);  
        String request = null;  
        //get request  
        while (scanner.hasNextLine()) {  
            request = scanner.nextLine();  
            //exit the program  
            if (request.equalsIgnoreCase("exit")) {  
                break;  
            }  
  
            //send the request to dispatcher  
            frontController.dispatchRequest(request);  
        }  
        //close the scanner  
        scanner.close();  
    }  
}
```

It is a client side class that contains main() method to establish the connection between server. It uses a while loop to receive requests from user and pass them to front controller.

#assignment 3

```
public class MarketClient {  
    private static View_login view_login;  
  
    public static void main(String[] args) {  
        // Create new instance of a Front Controller...  
        FrontController frontController = new FrontController();  
  
        //start login routine  
        boolean ifLogin = false;  
        while(!ifLogin){  
            view_login = new View_login();  
            view_login.showView();  
            ifLogin = frontController.loginRequest();  
        }  
  
        //after login success, receive request from user  
        Scanner scanner=new Scanner(System.in);  
        String request = null;  
        //get request  
        while (scanner.hasNextLine()) {  
            request = scanner.nextLine();  
            //exit the program  
            if (request.equalsIgnoreCase("exit"))  
                break;  
            //send the request to front controller  
            frontController.dispatchRequest(request);  
        }  
        //close the scanner  
        scanner.close();  
  
        // Terminate the program.  
        System.exit(0);  
    }  
}
```

Due to the use of client controller, RMI lookup is no longer needed to do in client class. (Client controller will work as a singleton object that holding a RMI remote object.)

When the client program runs, it firstly asks for login to authenticate the user role type.

After the user logged in, a while loop is used to catch requests from user.

FrontController.java

#assignment 2

```
//perform authentication, start session here?
private boolean isAuthenticatedUser() {
    View_login view_login = new View_login();
    View_login_fail view_login_fail = new View_login_fail();
    view_login.showView();

    String username = System.console().readLine();
    char[] pwd = System.console().readPassword("Enter your password: ");
    try {
        int status = myModel.login(username, pwd);
        if(status == 1) {
            //administrator
            System.out.println("User is authenticated successfully as administrator.");
        }
        else if (status == 2) {
            //customer
            System.out.println("User is authenticated successfully as customer.");
        }
        else { //login fail
            view_login_fail.showView();
            //return false;
        }
    } catch (RemoteException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    //System.out.println("User is authenticated successfully.");
    return true;
}

//Responsible for dispatching the request to the Dispatcher.
public void dispatchRequest(String request) {
    System.out.println("Page Requested: " + request);

    // If the user has been authenticated - dispatch request...
    if(isAuthenticatedUser()) {
        dispatcher.dispatch(request);
    }
}
```

It receives all the requests from the users and collects related parameters.

It performs authentication by making remote invocation (myModel.login()).

And it delivers requests to Dispatcher.

The down side of this authentication method is that it will ask for login each time the front controller receives a request. This will be fix after the session component is perform.

#assignment 3

```
public class FrontController {
```



```

//Declare objects
private Dispatcher dispatcher;
private Session session = null;
public Controller_client controller_client;

//Front Controller Constructor
public FrontController() {
    dispatcher = new Dispatcher();
    controller_client = Controller_client.getInstance();
}

public boolean loginRequest(){
    //input login combination
    Credential myCredential = new Credential();
    myCredential.setUname(System.console().readLine());
    myCredential.setPwd(System.console().readPassword("Enter your password: "));
    session = controller_client.loginProcess(myCredential);
    if (session == null){
        //show login fail view
        View_login_fail view_login_fail = new View_login_fail();
        view_login_fail.showView();
        return false;
    } else {
        String role = session.getUser().getRoleType();
        System.out.println("FrontController: session created, getUser = "
            + role);
        dispatchRequest("menu");
        return true;
    }
}

//check if the client is authenticated
private boolean isAuthenticated() {
    //validate by checking session
    return session.isAuthenticated();
}

//Responsible for dispatching the request to the Dispatcher.
public void dispatchRequest(String request) {
    System.out.println("FrontController: Page Requested: " + request);

    // If the user has been authenticated - dispatch request...
    if(isAuthenticated()) {
        dispatcher.dispatch(request, session);
    } else {
        System.out.println("FrontController: Invalid user");
    }
}
}

```

A loginRequest() method is added to the front controller. It parses the username and password from user input and then passes them as parameters to client controller to call loginProcess() method. A session object based on the user role type will be returned back to the front controller.

Dispatcher.java

#assignment 2

```

//based upon the request = dispatch the view
public void dispatch(String request) {
    if (request.equalsIgnoreCase("BROWSE")) {
        control_browse_customer.execute();
    }
    else if (request.equalsIgnoreCase("ADDADMIN")) {
        control_addAdmin.execute();
    }
    else {
        System.out.println("dispatcher cannot recognize the request: " + request);
    }
}
}

```

It receives requests from FrontController class. Based on the pre-define dispatch rules, it delegates ApplicationController.

So far, only two requests are accepted.












#assignment 3

```
//based upon the request = dispatch the view
public void dispatch(String request, Session session) {
    if (request.equalsIgnoreCase("menu")) {
        control_menu.execute(session);
    }
    else if (request.equalsIgnoreCase("browseCustomer")) {
        control_browse_customer.execute(session);
    }
    else if (request.equalsIgnoreCase("browseAdmin")) {
        control_browse_admin.execute(session);
    }
    else if (request.equalsIgnoreCase("addAdmin")) {
        control_addAdmin.execute(session);
    }
    else {
        System.out.println("dispatcher cannot recognize the request: " + request);
    }
}
}
```

No much changes. More concrete requests will be added to realize more functionalities of the system.

Application Controllers

#assignment 2

-  Control_addAdmin.java
-  Control_addItem.java
-  Control_browse_admin.java
-  Control_browse_customer.java
-  Control_cart.java
-  Control_checkout.java
-  Control_login.java
-  Control_removeItem.java
-  Control_updateInfo_admin.java
-  Control_updateInfo_customer.java
-  Control_viewOrder.java













```
public void execute() {
    Factory_abstract consumerFactory = Factory_producer.getFactory("CONSUMER");
    Interface_customer customer = consumerFactory.getConsumer();
    System.out.println("Customer-browse-controller executed.");
    System.out.println("A customer object is created by consumerFactory.");

    //use command pattern to call method
    // Concrete Commands...
    Command_browse_customer command_browse_customer = new Command_browse_customer(customer);
    System.out.println("A customer-browse-command object is created.");
    // Invoker Object...
    Command_invoker invoker = new Command_invoker();
    invoker.takeCommand(command_browse_customer);
    System.out.println("invoker executed customer-browse-command");
    invoker.executeCommands();

    //dispatch the result to view
    view_browse_consumer.showView("item object array");
    System.out.println("=====");
    //show menu
    view_menu_customer.showView();
}
```

These are ApplicationController. For each request, dispatcher will delegate one of them to perform the task. Inside ApplicationController, it will use Abstract Factory and Command pattern to make it possible to perform the business logic with server model. After it gets results, it dispatches and render a view.

#assignment 3

-  Control_addAdmin.java
-  Control_addItem.java
-  Control_browse_admin.java
-  Control_browse_customer.java
-  Control_cart.java
-  Control_checkout.java
-  Control_login.java
-  Control_menu.java
-  Control_removeItem.java
-  Control_updateInfo_admin.java
-  Control_updateInfo_customer.java
-  Control_viewOrder.java

No much changes. More concrete application controllers will be added to realize more functionalities of the system.

Inside each controller:

```
public class Control_addAdmin {
    private Controller_client controller_client;
    private View_menu view_menu;

    public Control_addAdmin() {
        controller_client = Controller_client.getInstance();
        view_menu = new View_menu();
    }

    public void execute(Session session) {
        //get factory
        Factory_abstract adminFactory =
        Factory_producer.getFactory(session.getUser().getRoleType());
        Interface_user user = adminFactory.getAdmin();

        controller_client.addAdmin(user, session);







        //show menu
        view_menu.showView(session.getUser().getRoleType());
    }
}
```

It instantiates a factory object to create a user object based on the role type of the session object. Then, it invokes the client controller method to perform the task.

The logic in these application controller classes are not concrete yet. I expect them to retrieve inputs from the view and passes them as parameters to call methods from client controller, and get return data from it and render the views.

View objects

#assignment 2






-  View_browse_customer.java
-  View_login_fail.java
-  View_login.java
-  View_menu_admin.java
-  View_menu_customer.java
-  View_welcome.java

```
public class View_browse_customer {
    public void showView(String itemArr) {
        System.out.println("Welcome to the consumer browse view");
        System.out.println("Products are under construction");
    }
}
```

These are part of view classes that be dispatched by Application controllers, except for the “View_welcome” which is rendered before any requests.

There are more view classes need to be created, however, due to the number of files is already very annoying, the rest view classes will be created when they are needed.

#Assignment 3

 View_browse_customer.java
 View_login_fail.java
 View_login.java
 View_menu.java
 View_welcome.java

No much changes. More concrete view objects will be added to realize more functionalities of the system.

```
public class View_menu {
    public void showView(String role) {




        System.out.println("Welcome to Online Market place, " );// + username);
        System.out.println("Identity: " + role );
        System.out.println("Options:" );

        //show menu based on role type
        if (role.equalsIgnoreCase("admin")){
            System.out.println("1. Browse Items (Update/Remove) (browseAdmin)" );
            System.out.println("2. Add Item (empty)" );
            System.out.println("3. Remove Item (empty)" );
            System.out.println("4. Update Item (empty)" );
            System.out.println("5. Add administrator (addAdmin)" );
            System.out.println("6. Update Personal Information (empty)" );
        }
        else if (role.equalsIgnoreCase("customer")){
            System.out.println("1. Browse Items (browseCustomer)" );
            System.out.println("2. Shopping Cart (empty)" );
            System.out.println("3. View Order (empty)" );
            System.out.println("4. Update Personal Information (empty)" );
        }
        else {
            System.out.println("menu error: unknown user role" );
        }
    }
}
```

The change of menu view is worth to mention. After the user is authenticated successfully, it will display the menu based on the role type of the session object.

Interface classes

#assignment 2

 Interface_admin.java
 Interface_command.java
 Interface_customer.java


```
/**
 * RMI Interface for admin
 */
public interface Interface_admin extends java.rmi.Remote{
    //command pattern
    void browse() throws java.rmi.RemoteException;
    void addItem() throws java.rmi.RemoteException;
    void removeItem() throws java.rmi.RemoteException;
    void updateItem() throws java.rmi.RemoteException;
    void addAdmin() throws java.rmi.RemoteException;
    void updateInfo() throws java.rmi.RemoteException;
}
```

These are interface classes. Abstract Factory will need to use Interface_admin and Interface_customer to create according objects. Command pattern will need Interface_command to create concrete classes.

#assignment 3

```
public interface Interface_user extends Remote{
    //command pattern
    //customer
    Command_browse_customer browse(MVC_model_interface myModel) throws RemoteException;
    Command_loadCart loadCart(MVC_model_interface myModel) throws RemoteException;
    Command_checkOut checkOut(MVC_model_interface myModel) throws RemoteException;
    Command_viewOrder viewOrder(MVC_model_interface myModel) throws RemoteException;
    Command_updateInfo_customer updateInfo(MVC_model_interface myModel) throws RemoteException;







    //admin
    Command_browse_admin browseAdmin(MVC_model_interface myModel) throws RemoteException;
    Command_AddItem addItem(MVC_model_interface myModel) throws RemoteException;
    Command_RemoveItem removeItem(MVC_model_interface myModel) throws RemoteException;
    Command_updateItem updateItem(MVC_model_interface myModel) throws java.rmi.RemoteException;
    Command_addAdmin addAdmin(MVC_model_interface myModel) throws RemoteException;
    Command_updateInfo_admin updateInfoAdmin(MVC_model_interface myModel) throws RemoteException;
}
```

The interfaces for Abstract Factory are combined into one:  Interface_user.java

Since the admin and customer belong to a family of related objects, there is no need to separate the interfaces.

Abstract Factory

#assignment 2

-  Factory_abstract.java
-  Factory_admin.java
-  Factory_concrete_admin.java
-  Factory_concrete_customer.java
-  Factory_customer.java
-  Factory_producer.java

There are necessary classes to implement the Abstract Factory patter. A super factory produces two sub-factories: Factory_customer and Factory_admin.

#assignment 3

```
public class Factory_concrete_customer implements Interface_user {

    @Override
    public Command_browse_customer browse(MVC_model_interface myModel) throws RemoteException{
        System.out.println("customer call browse()");
        Command_browse_customer command_browse_customer = new
        Command_browse_customer(myModel);
        return command_browse_customer;
    }

    @Override
    public Command_loadCart loadCart(MVC_model_interface myModel) throws RemoteException {
        System.out.println("customer call cart()");
        Command_loadCart command_loadCart = new Command_loadCart(myModel);
        return command_loadCart;
    }

    @Override
    public Command_viewOrder viewOrder(MVC_model_interface myModel) throws RemoteException {
        System.out.println("customer call viewOrder()");
        Command_viewOrder command_viewOrder = new Command_viewOrder(myModel);
        return command_viewOrder;
    }
}
```

```

        @Override
        public Command_updateInfo_customer updateInfo(MVC_model_interface myModel) throws
RemoteException {
            System.out.println("customer call updateInfo()");
            Command_updateInfo_customer command_updateInfo_customer = new
Command_updateInfo_customer(myModel);
            return command_updateInfo_customer;
        }

        @Override
        public Command_checkOut checkOut(MVC_model_interface myModel) throws RemoteException{
            System.out.println("customer call checkOut()");
            Command_checkOut command_checkOut = new Command_checkOut(myModel);
            return command_checkOut;
        }

        /**
         * Admin
         * unimplemented methods
         */
        @Override
        public Command_browse_admin browseAdmin(MVC_model_interface myModel) throws RemoteException {
            return null;
        }

        @Override
        public Command_AddItem addItem(MVC_model_interface myModel) throws RemoteException {
            return null;
        }

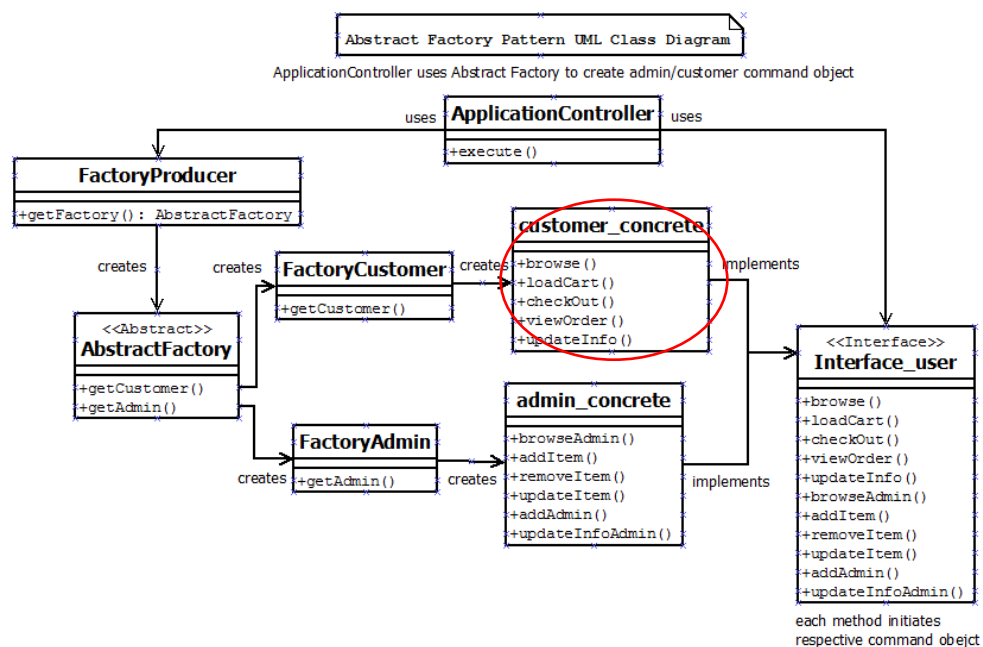
        @Override
        public Command_RemoveItem removeItem(MVC_model_interface myModel) throws RemoteException {
            return null;
        }

        @Override
        public Command_updateItem updateItem(MVC_model_interface myModel) throws RemoteException {
            return null;
        }

        @Override
        public Command_addAdmin addAdmin(MVC_model_interface myModel) throws RemoteException {
            return null;
        }

        @Override
        public Command_updateInfo_admin updateInfoAdmin(MVC_model_interface myModel) throws
RemoteException {
            return null;
        }
    }
}













```



The source code is from “customer_concrete” class. In each method, it instantiates a respective command object. It returns null for all non-customer type methods.

Command Pattern

#assignment 2

-  Command_addAdmin.java
-  Command_AddItem.java
-  Command_browse_admin.java
-  Command_browse_customer.java
-  Command_cart.java
-  Command_checkOut.java
-  Command_invoker.java
-  Command_RemoveItem.java
-  Command_updateInfo_admin.java
-  Command_updateInfo_customer.java
-  Command_updateItem.java
-  Command_viewOrder.java













```
public class Command_browse_customer implements Interface_command{
    private Interface_customer customer;

    /**
     * Constructor
     *
     * @param customer1
     */
    public Command_browse_customer(Interface_customer customer1){
        this.customer = customer1;
    }

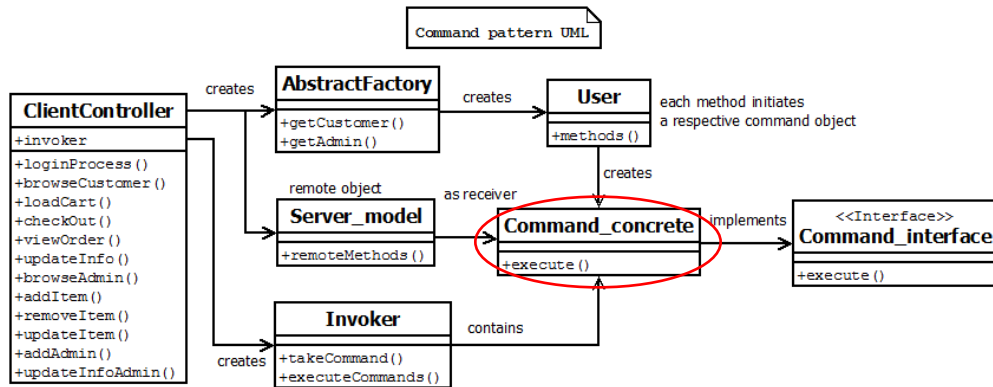
    /**
     * Execute Command Method
     */
    public void execute() {
        customer.browse();
    }
}
```

These are pre-defined concrete command classes. Each will contain a sequence of method calls to perform the task.

#assignment 3

-  Command_addAdmin.java
-  Command_AddItem.java
-  Command_browse_admin.java
-  Command_browse_customer.java
-  Command_checkOut.java
-  Command_invoker.java
-  Command_loadCart.java
-  Command_RemoveItem.java
-  Command_updateInfo_admin.java
-  Command_updateInfo_customer.java
-  Command_updateItem.java
-  Command_viewOrder.java

The way that concrete command object works is changed. As the below source code from “Command_addAdmin” command object shown.



```

public class Command_addAdmin implements Interface_command{
    private MVC_model_interface myModel;

    //Constructor
    public Command_addAdmin(MVC_model_interface myModel){
        this.myModel = myModel;
    }

    //Execute Command Method
    public void execute(Session session) {
        try {
            myModel.addAdmin(session);
        } catch (Exception e) {
            System.out.println("Command_addAdmin: Model calls Exception: " +
e.getMessage());
        }
    }
}
  
```

Inside execute() method, it makes remote invocation with a session object as its parameter.

Presentation/Sample Runs

1. Go to the project root directory in Tesla and type “make” to compile the files.

```
[caozh@tesla test]$ make
```

2. Type “rmiregistry 1888&” to register port 1888 for the program.

```
[caozh@tesla test]$ rmiregistry 1888&
[1] 89732
```

3. Type “java -Djava.security.policy=policy MarketServer” to run the server.

```
[caozh@tesla test]$ java -Djava.security.policy=policy MarketServer
Creating a Online Market Server!
Marketplace Server: binding it to name: //tesla.cs.iupui.edu:1888/MarketServer
Marketplace Server Ready!
```

4. Open another Putty window and login to your Tesla server, navigate to the same project directory, type “java -Djava.security.policy=policy MarketClient” to run the client.


```
[caozh@tesla test]$ java -Djava.security.policy=policy MarketClient
Please login
Enter your user name: █
```

5. First test is to login as a customer and try to do a task that requires admin role to do.

```
Enter your user name: josephzelo
Enter your password:
AuthorizationInvocationHandler: no annotation
Controller_client: processLogin() returns customer
FrontController: session created, getUser = customer
FrontController: Page Requested: menu
Welcome to Online Market place,
Identity: customer
Options:
1. Browse Items (browseCustomer)
2. Shopping Cart (empty)
3. View Order (empty)
4. Update Personal Information (empty)
```

Use username: josephzelo, password: asdasd to login as a customer.

Client controller received a return value “customer” from server, then it created a session object with role type “customer”. Returned the session back to front controller.

The Front controller received the session object from client controller. It delegated the request “menu” to dispatcher. The Dispatcher delegated menu controller to render the menu view and display it.

6. “addAdmin” is a request to perform a task that only admin role can do which is to add a new admin. Type “addAdmin”.

```
addAdmin
FrontController: Page Requested: addAdmin
addAdmin command created
annotation: @RequiresRole(value=admin)
session: customer
Command_addAdmin: Model calls Exception: Invalid Authorization - Access Denied to addAdmin() function!
Welcome to Online Market place,
Identity: customer
Options:
1. Browse Items (browseCustomer)
2. Shopping Cart (empty)
3. View Order (empty)
4. Update Personal Information (empty)
```

Front controller received the request “addAdmin”, sent to dispatcher, then dispatcher delegated addAdmin application controller to do the task. It created an admin object and sent to client controller. Client controller make the remote invocation to perform the task. In the server side, AuthorizationInvocationHandler checked the role requirement. Since the role type in session object is “customer”, however, the required role for calling addAdmin() method in server model object is “admin”. The call fell into AuthorizationException. It returned back to client with the error message “Invalid Authorization”. Then the addAdmin application controller in the client side transferred the view back to main menu.

7. Next, use admin account to do this request (addAdmin) which a customer cannot do.

```

Please login
Enter your user name: admin
Enter your password:
AuthorizationInvocationHandler: no annotation
Controller_client: processLogin() returns admin
FrontController: session created, getUser = admin
FrontController: Page Requested: menu
Welcome to Online Market place,
Identity: admin
Options:
1. Browse Items (Update/Remove) (browseAdmin)
2. Add Item (empty)
3. Remove Item (empty)
4. Update Item (empty)
5. Add administrator (addAdmin)
6. Update Personal Information (empty)

```

Type “exit” request to close the program. And type “java -Djava.security.policy=policy MarketClient” to rerun the program.

Type username: admin, password: admin to login as an admin.

```

addAdmin
FrontController: Page Requested: addAdmin
addAdmin command created
annotation: @RequiresRole(value=admin)
session: admin
Welcome to Online Market place,
Identity: admin
Options:
1. Browse Items (Update/Remove) (browseAdmin)
2. Add Item (empty)
3. Remove Item (empty)
4. Update Item (empty)
5. Add administrator (addAdmin)
6. Update Personal Information (empty)

```

Type “addAdmin” to send it as a request to the program.

In the server side, AuthorizationInvocationHandler checked the required role of addAdmin() method with the role type of the session object. They both are “admin”. Role Based Access Control passed. Method addAdmin() invoked. As below screenshot shown, the server side echo the message.

```

input username is:admin
input password is:admin
username is correct
Server model invokes addAdmin() method

```

These steps demonstrated the implementation of the Role Based Access Control, and it worked great.

Discussion

#assignment 1

At the moment, I have implemented the system using JAVA RMI, and design the system using MVC pattern. However, to optimize the Object-Oriented design, I need to move the controller from client side to server side. This will be done in next assignment.

On the other hand, the designs inside the view object is not object-oriented. I need to improve it in the future.

#assignment 2

At the moment, I implemented two additional design patterns: command pattern and abstract factory pattern. There is a severe problem in my mind. Inside command pattern, I did not invoke remote methods, instead I try to invoke methods offering from the interface implemented by customer/admin concrete object produced by respective factory. Inside the customer/admin concrete object, methods are hard coded so far, but I am aware that it is unreasonable to define making remote invocation inside customer/admin concrete object.

#assignment 3

Many things are changed in this assignment: Command pattern, Abstract Factory pattern, front controller pattern, Singleton pattern, Authorization pattern, reflection, proxy, annotation. They worked well. The next thing to do is to write the concrete functionalities of the system to do the real task.

Problem: I am still not confident of the way that apply the Abstract Factory and Command pattern. Also, I am not sure where should I instantiate them. In application controller or client controller?