Student:Serban Raluca

**Group:30238**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 3

3. System Architectural Design 3

4. UML Sequence Diagrams 3

5. Class Design 3

6. Data Model 3

7. System Testing 3

8. Bibliography 3

1. Requirements Analysis

# Assignment Specification

The task for this application was to build a deal search engine for furniture products ,using an OOP language. For this application I used Java. Data used in the application is stored in a database , from where it is retrieved later in the application logic. The application lets the users to make an account , login , browse for product and make orders. The administrator maintains the site , manages product operations like adding , deleting or updating products , updates the database when orders are placed .The users can just view the items but cannot modify them .The items selected by an user are kept in a cart , from where the user can make the order .The site provides information about the cost of the products and also about their availability .The users can add as well as remove products from the cart.

# Functional Requirements

The functional requirements of the application are:

* **Register**-users of the application can create an account that they can later use to get access to the application’s provided services
* **Login**-users can access the application based on the username and the password that they provide when they create the account
* **Logout-** this function allows the users to leave their account
* **Search for deals in the product list-**users can search for various provided deals in the application interface
* **Filter deals**-the deals provided can be filtered by name , type and price
* **Add products to cart-**users can add the products they want to buy to a virtual cart
* **Checkout/Order-**after the products had been added to cart , the users can proceed to checkout and specify the order details
* **View order history-**from their account users can see their orders , as well as their state
* **Update order state-**the staff can update the state of an order
* **Feedback-**when an order is delivered the users can provide feedback
* **Add/Remove product-**the admin can add or remove products
* **Add discount-**the admin can apply different discounts to the existing products , changing in this way their price

# Non-functional Requirements

Non-functional requirements describe user visibility aspects of the system that are not directly related to the functional behavior of the system:

* **User interface –**the system should provide a graphical user interface with different forms of presentation and should be easy to use ; a web browser is an interface between the clients and the software system
* **Documentation-**information about how to use the system should be provided to the users
* **Error handling-**the system handles exception in different situations that mightoccur and notifies the user when they happen(ex. If an order cannot be completed)
* **Reliability-**the system is reliable as it efficiently interacts with the user and secures access to user’s confidential information through user authentication.
* **Robustness-**in case of failures the system cannot lose data and it handles exceptions such as invalid data or incomplete fields
* **Flexibility-**the system is flexible to changes
* **Security-**the system provides secure registration and management facilities for users , so that no other unauthorized user can use their account
* **Validation of data-**all data inputs of the application are validated against invalid before being saved in the database

2. Use-Case Model



**Use case description:**

**Login**:

**Use case goal:**

This use case describes how a user logs into the Furniture Deals System.

#### **Main success scenario**

Primary actor for this use case is the user.This use case starts when an user wants to log into the Furniture Deals System.

1. The system requests that the user to enter his/her username and password.
2. The actor enters his/her username and password.
3. The system validates the entered name and password and searches in the database for the data introduced .
4. The user can choose to check the option **Remember me** for the system to remember him for the next login operation.
5. The user can use the button **Reset** to clear all the text boxes.
6. If the username and password are found in the database and are correct then the system lets the user in and redirects him to the home page.

#### **Alternate scenarios**

**Invalid Name / Password**:

If in the *Basic Flow*the user enters an invalid name or password, the system displays an error message(Invalid creditentials).

If the user doesn’t specify all the required fields ,some alert messages will alert him of that.

**Pre-Conditions**

None

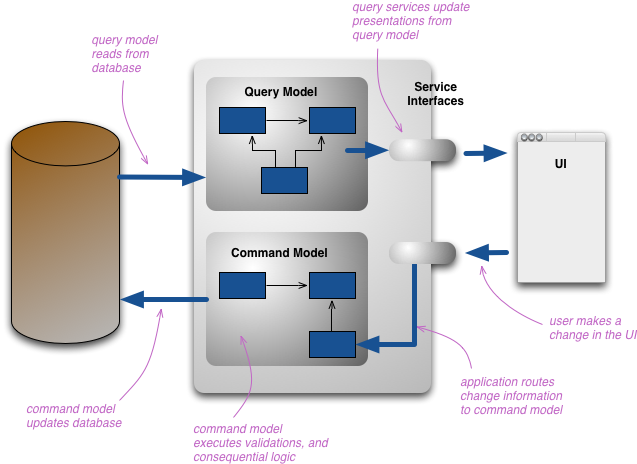
#### **Post-Conditions**

If the use case was successful, the actor is now logged into the system. If not the system state is unchanged.

3. System Architectural Design

**3.1 Architectural Pattern Description**

For this application the CQRS(Command and Query Responsibility Segregation) architectural pattern is used.The basic idea is that you can divide a system’s operations into two separated categories:**queries-**these just return a result and don’t change the state of the system and they are free of side effects;**commands**-these change the state of the system.each method either returns state or mutates state,but not both.The separation aspect in CQRS is achieved by grouping query operations in one layer and commands in another layer.Each layer has its own data model and is built using its own combination of patterns and technologies.



The web page that the user is looking at is rendered using the query model.If they initiate a change that change is routed to the separate command model for processing;the resulting change is communicated to the query model to render the updated state.

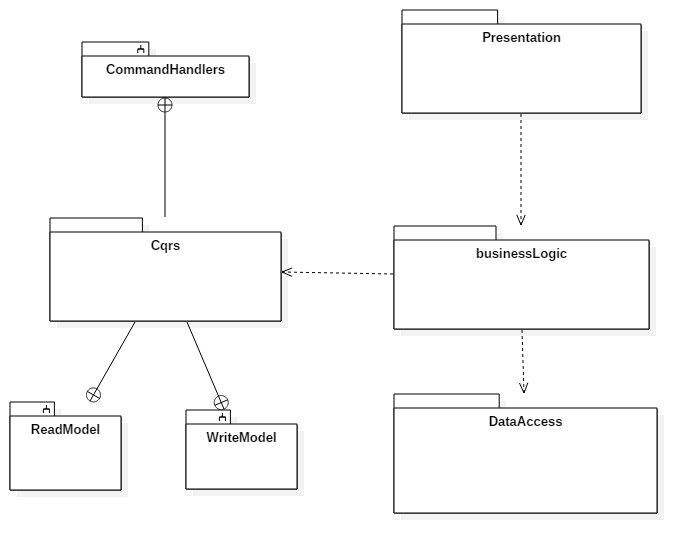
Having separate query and update models simplifies the design and implementation. For greater isolation, you can physically separate the read data from the write data. In that case, the read database can use its own data schema that is optimized for queries.

Benefits of CQRS:

* **Independent scaling**-allows the read and write to scale independently.
* **Security-**it’s easier to ensure that only the right domain entities are performing writes on the data.
* **Separation of concerns**-segregating the read and write sides can result in models that are more maintainable and flexible.Most of the complex business logic goes into the write model ,so the read model can be relatively simple.

**3.2 Diagrams**

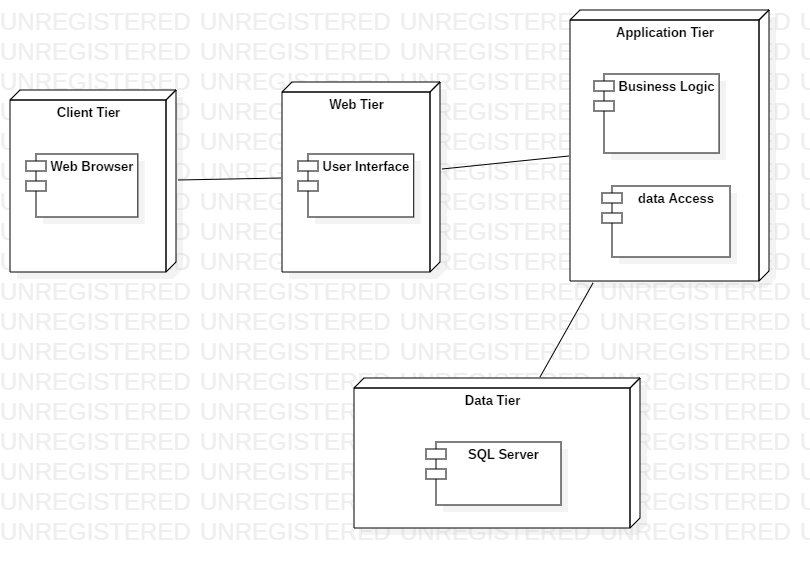
**Package Diagram**

****

**Component diagram**

****

**Deployment Diagram**

****

4. UML Sequence Diagrams

Shopping Cart scenario:



5. Class Design

**5.1 Design Patterns Description**

**Factory Method**

The design pattern used in this application is the factory method pattern .This is a creational pattern that uses factory methods to deal with the problem of creating objects ,but without the need to specify the exact class of the object that will be created .This is done by calling a factory method,usually specified by an interface and implemented by child classes or implemented by a base class and optionally overridden by derived classes,instead of calling a constructor.

This enables writing of subclasses to change the way an object is created (to redefine witch class to instantiate).

This pattern relies on inheritance ,as object creation is delegated to subclasses that implement the factory method to create objects.

In the context of this application the pattern is used for creating different types of discounts to apply on products ,in order to change their price .There are 3 types of discounts:10%,15%,20%.For creating the discounts the method **createDiscount** from the class **DiscountCreator** is use and this method returns an instance of the Discount class,instance of one of its subclasses depending on the type of discount wanted:Discount10,Discount15.Discount20.

**Observer Design Pattern**

Observer is a behavioral design pattern.It specifies communication between objects:observable and observers.An observable is an object which notifies observers about changes in its state.To be able to do that ,the observable object needs to keep referances to the observers(it will have a list of Observers).

The Observer should have an update() method which is invoked when the state of the Observable(Subject) is changes.For that we define an interface Observer with the update() method.

Advantages:

Provides a loosely coupled design between objects that interact. Loosely coupled objects are flexible with changing requirements. Here loose coupling means that the interacting objects should have less information about each other.

Observer pattern provides this loose coupling as:

* Subject only knows that observer implement Observer interface.Nothing more.
* There is no need to modify Subject to add or remove observers.
* We can reuse subject and observer classes independently of each other.

In the context of this application the Observer pattern is used to notify users when the state of an order is changed by the administrator.The servlet class responsible for the updateOrder operation is the Subject,and the Observer is the servlet class responsible for the orderHistory option for the user.When the update order method is used the state of the orders in the user history will be updated ,by making the necessary changes in the database.This is done in the update method of the Observer.

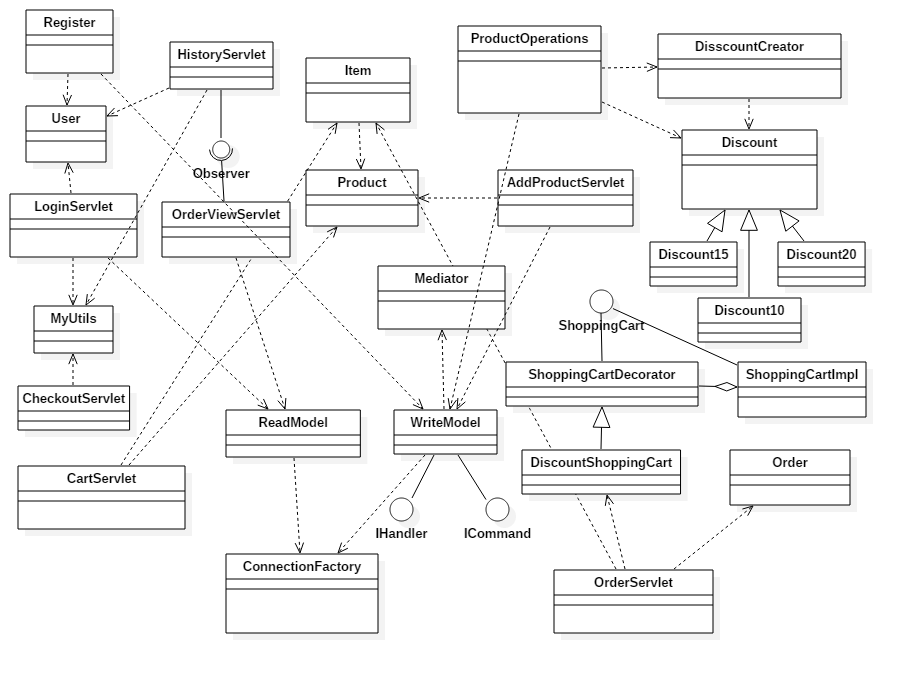
**Decorator Design Pattern**

Another design pattern used in this application is the Decorator Design Pattern.The intent of this pattern is to attach responsabilities to an object dynamically.It provides a flexible alternative to subclassing for extending functionality.

This helps when you want to add behavior or state to individual objects at run-time.In this case inheritance is not feasible because it is static and applies to an entire class.The solution to this problem involves encapsulating the original object inside an abstract wrapper interface.Both the decorator objects and the core object inherit from this abstarct interface.The interface uses recursive composition to allow an unlimited number of decorator “layers” to be added to each core object.This pattern allows responsabilities to be added to an object ,not methods to an object’s interface.The interface must remain constant as succesive layers are specified.In the context of this application the decorator pattern is used for applying deals to the shopping cart.This pattern allows the creation of a specific shopping cart.The pattern is used to add functionality to the getTotal() method of the general ShoppingCart class,so that a discount can be applied on the total price in cases of orders that have the total bigger than 1000.

In order to do this the DiscountShoppingCart extends the Decorator class,and gets to the constructor an instance to the general ShoppingCart class; in the implementation of the getTotal() method there is used the getTotal() method from the superclass, that gets the normal total price ,that will then be modified.

**5.2 UML Class Diagram**



6. Data Model

In this application the data model is represented as Java classes and as database tables .The business logic of the system is done by Java objects ,while the database provides storage for those objects. When needed those objects are then retrieved form the database.

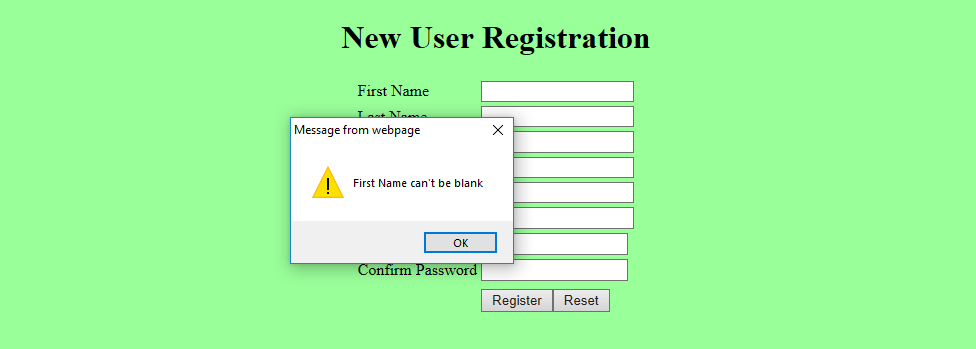
The classes that from the data model for this application are:User,Order,Product.In the datbase there are coresponding tables for these type of objects.

1. System Testing

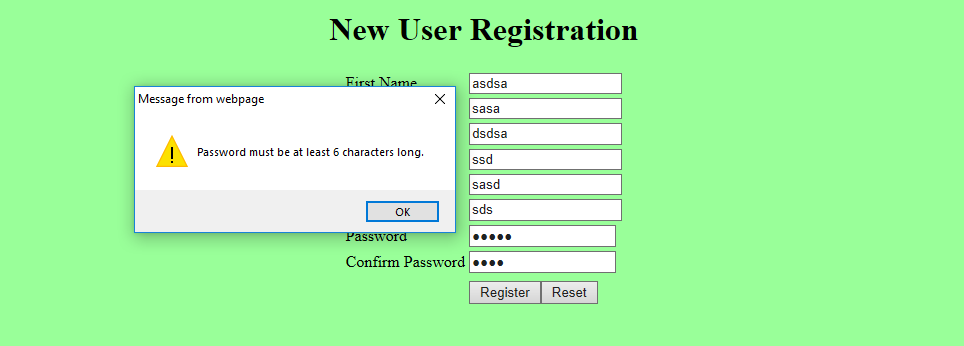
Integration testing-more components are combined together and tested as a group.

Validation testing is used to see if the system satisfies the business requirements.

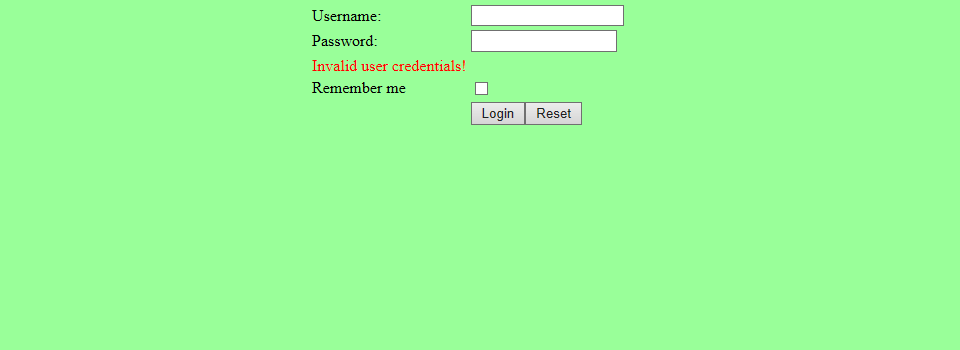
Testcase1-Preventing empty fields in registration form of users



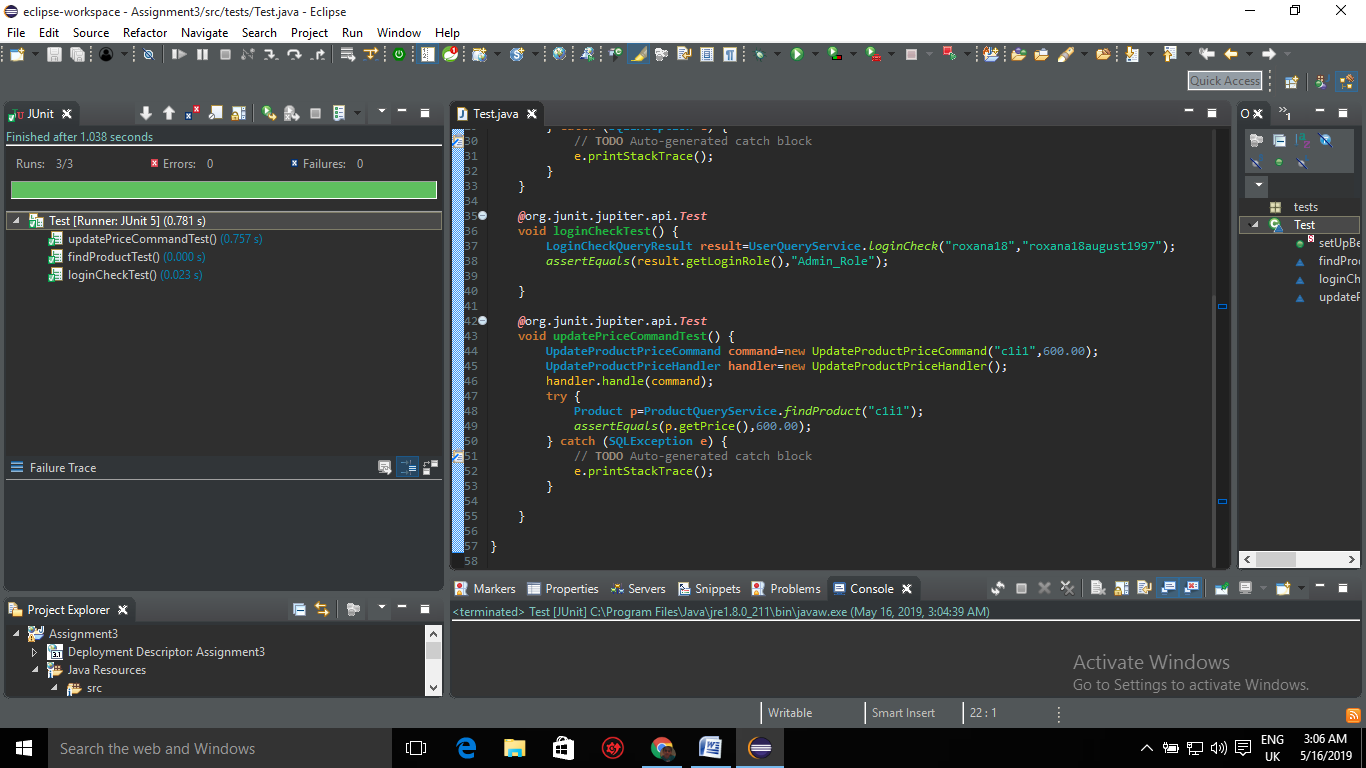
Testcase2-Validating registration of user



Testcase3-Preventing empty fields in login form



To test the application there were also used some Junit tests:



1. Bibliography

* <https://www.javagists.com/abstract-factory-pattern-in-java>
* <https://www.ntu.edu.sg/home/ehchua/programming/java/JavaServlets.html>
* <https://o7planning.org/en/11071/create-a-simple-login-application-and-secure-pages-with-java-servlet-filter>
* <http://met.guc.edu.eg/OnlineTutorials/JSP%20-%20Servlets/Full%20Login%20Example.aspx>
* <https://way2java.com/servlets/servlet-cookies-simple-shopping-cart-example/>
* <https://www.baeldung.com/java-observer-pattern>
* <https://sourcemaking.com/design_patterns/decorator>
* <https://dzone.com/articles/mediator-pattern-1>
* <https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/apply-simplified-microservice-cqrs-ddd-patterns>