Comp62542 Pattern-Based Software Engineering

Team 2

Songtao Wang - 10612858
Shijie Deng - 10645839
Francisco Javier Ballester Arenas - 10766721

Table of contents

Java-based e-commerce application

Our java-based e-commerce application is based around a food ordering system where we have an application where the user sees various restaurants and can order food to their house from these restaurants. There are a few aspects of our application that make it stand out from the rest. To begin with there are two types of users, the normal user and the VIP user. The normal user gets free delivery after they spend 30 pounds but the VIP user gets free delivery after spending 10 pounds on their order. However the VIP user will have to spend a monthly fee of 5 pounds to become a VIP user. In addition the application has a live status bar which informs the user at what state their order is in. Finally to be more environmentally friendly we give the users the option to opt out of receiving plastic cutlery as most customers already have cutlery at their home.

Application Architecture

The application will begin with a login or a registration page to create a new user. Once the user has logged in they will go to the main page where they will be greeted with the list of the most trendy restaurants. The user can then select a restaurant and their menu page will appear. The user can then select what food they wish to order and then move on to the checkout page where the user can pay for their order. Once the user has paid they will be able to see at what stage their food is at.

For our application, we are following the spring MVC pattern. Each service we provide has the same structure. The structure consists of a controller that gets called by the front end to do a certain service. The controller then calls this service. The service then retrieves the necessary information and sends it back to the controller. The controller then sends it to the view object which sends it to the front end via a JSON string. The services the application provides are;

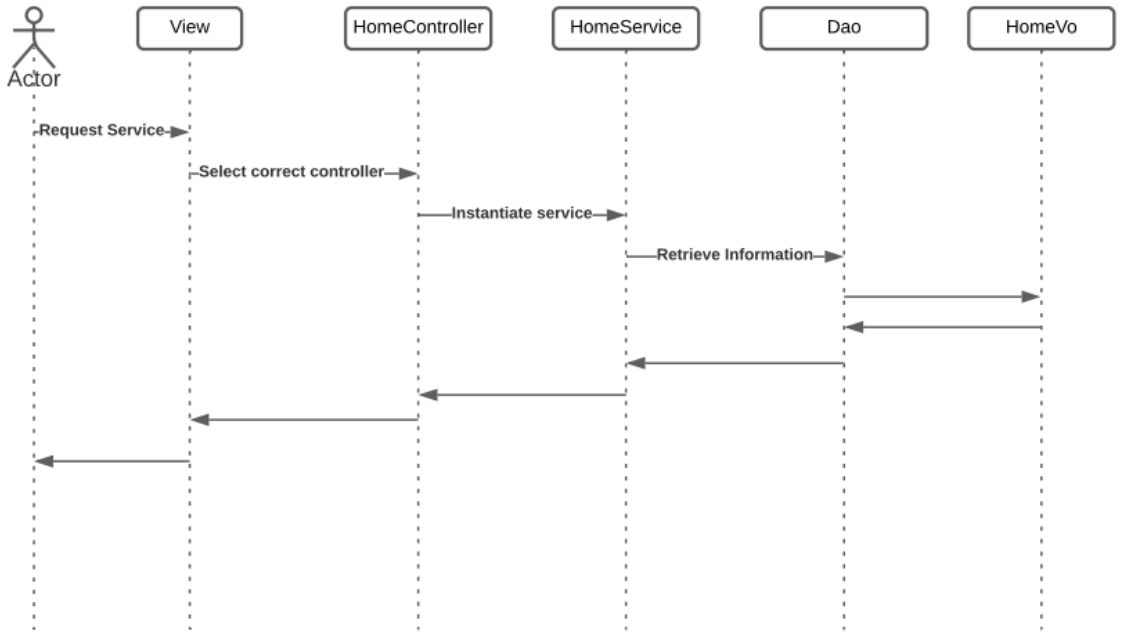
Service	**Explanation**
Address	Handles the information regarding the address of the user
Cart	Handles the information of the food cart of the user
Home	Is the main page that shows the list of the available restaurants
Login	Handles the logging into the application
Menu	Shows all the available food a restaurant has to offer
Order	Manages the past and existing orders of the customer
Register	Handles the creation of a new user
User	Manages the information about the user

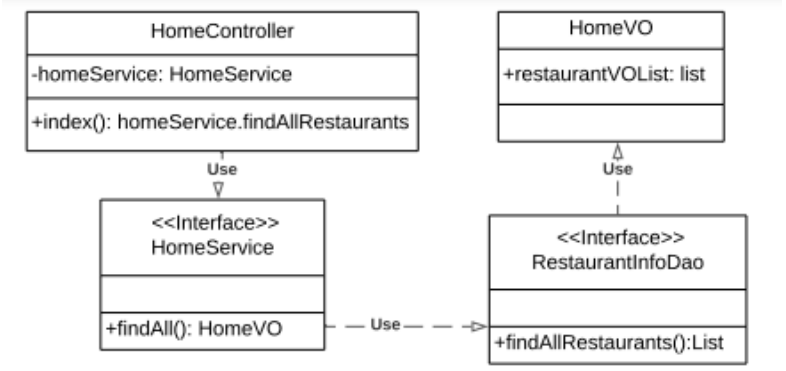
By using the spring MVC model it has been easier to add new services to our application as they are all independent from each other. This led to faster development as one change to a

service did affect the rest of the services. On the other hand the spring MVC model has strict rules which meant the design of our application was more difficult and required more effort. However once we understood the MVC architecture the development of the application went very well.

HomeController

Below is a UML flow diagram and a UML class diagram of our home service. All the services follow the same structure. The HomeController is the controller which organizes the flow of the data of the service. HomeService is the class that runs the required service. HomeVO is our data type that is used to collect data from the data storage in the correct format. Dao is the data access point which is the model of our system where the necessary data is stored.





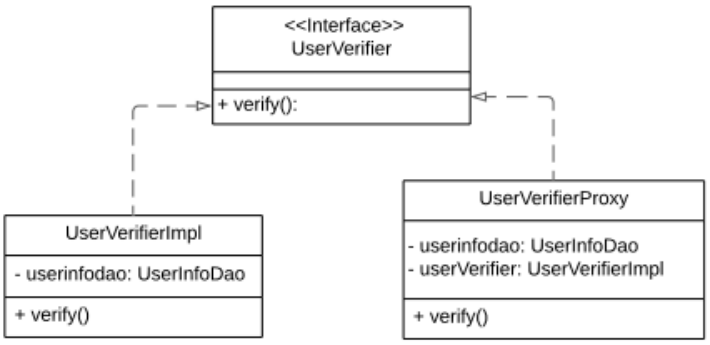
Patterns

All of the patterns used are based on three main design principles.

- Encapsulation, which is grouping the data and how they are accessed into one place.
- Abstraction, encourages the programmer to create a solution for the bigger problem not just this instance of the problem. This allows for reusability of the code.
- Inheritance, allows classes to be built on existing classes

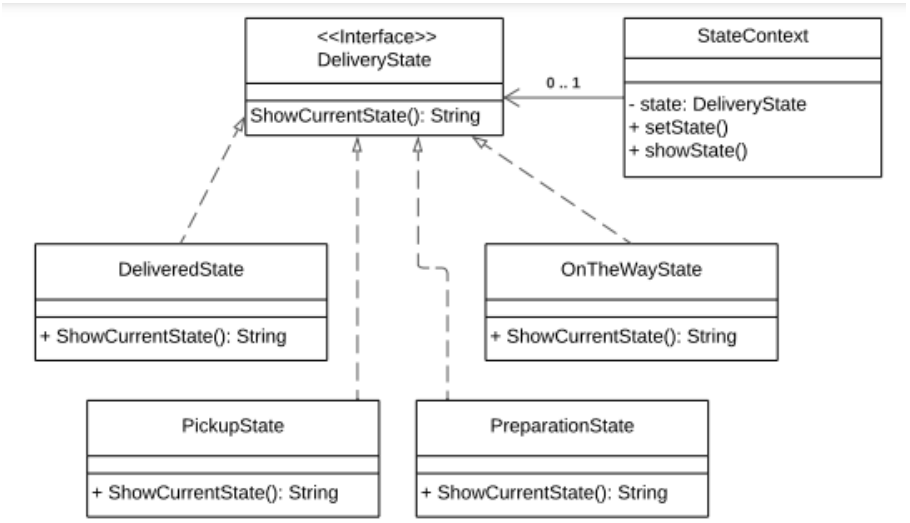
These design principles lead to the increase of cohesion and the decrease of coupling. Thanks to encapsulation, abstraction and inheritance, changing the functionality of one part of the system will not affect another part and this decreases code coupling. Moreover this increases cohesion as the functionality of the service is grouped together. Due to the patterns being based on these principles all the patterns chosen increased our code cohesion and decreased our code coupling.

Proxy Pattern



In our application the proxy pattern which is a structural pattern, is used in the login service to allow access into the application. By implementing the proxy pattern we add a layer of control to the login service. The main improvement the proxy pattern has given to the system is the ability to separate the computation from the control. This improved the readability of our code and added a layer of abstraction.

State Pattern

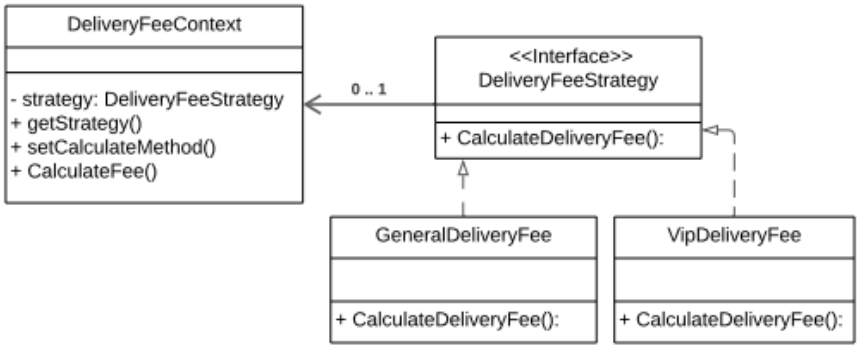


In our system the state pattern which is a behavioural pattern, is used to control what state the user's order is in. It can be either,

1. Preparing
2. Picking up
3. On the way
4. Delivered

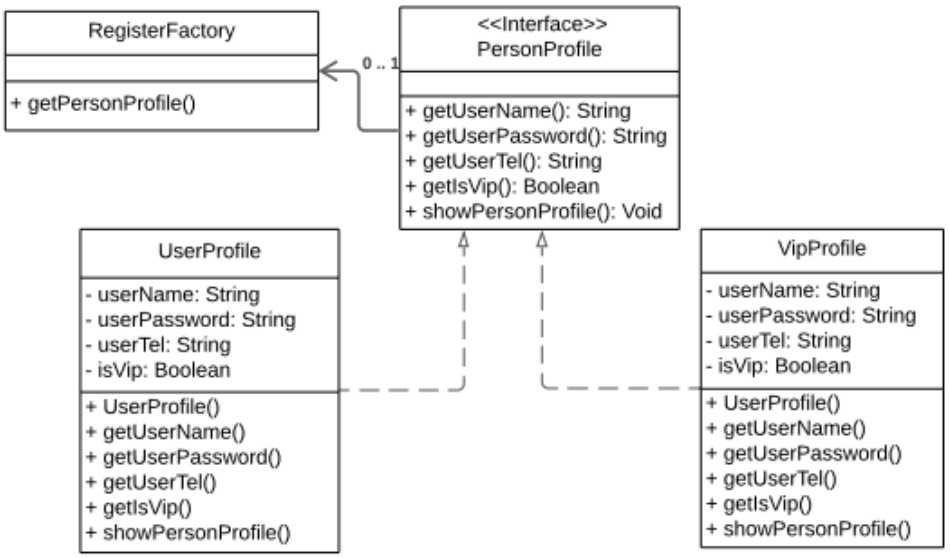
By using the state pattern we have simplified the process of changing the state of the order. The state pattern simplified the process of creating or editing states as adding new behaviour means just adding a new class.

Strategy Pattern



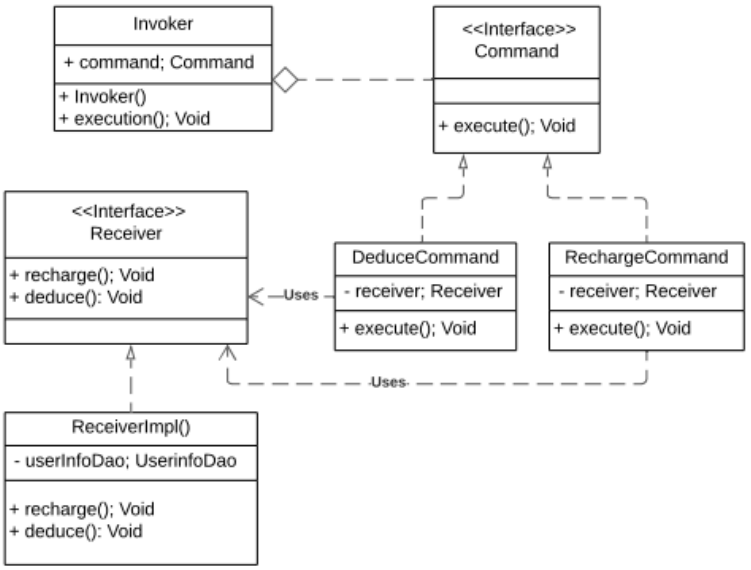
The Strategy pattern was used in the system to switch between different delivery fees during runtime. If a normal user spent more than 30 pounds they would get free delivery. However if they were VIP users they would get free delivery after spending 10 pounds. This behavioural pattern simplified our process of adding this price difference to our checkout service.

Factory Pattern



In our application the factory pattern is used when creating new users in the registration page. This creational pattern can create new users without having to know the instantiation logic. Therefore the creation logic of the users can change without affecting the client code. The Factory pattern allowed us to easily add the VIP user to the system which added a unique selling point to our system.

Command Pattern



The command pattern has been used to increase and decrease the user's balance depending if they have bought some food if they have inserted funds into their account. By using this behavioural design pattern we avoided hard-wiring the request of depositing and taking away money from the users account. The command pattern allowed us to easily add actions to the receiver without affecting the client code.