Home Service Marketplace System

Module: CS5721 – Software Design

Team: Dijkstra

Team Members:

1) Ashutosh Lembhe (24088714)
2) Achyutam Verma (24046949)
3) Dhruv Upadhyay (24039497)
4) Gorav Sharma (24246182)

Table of Contents

Table of Contents

1. Project Description.

1.1 Project Overview

The Home Service Marketplace System is a web-based utility delivery platform, which helps local Service providers like electrician, plumber and other types of services to help get them more work. Now a days its tough doing all the work by yourself without the proper equipment and knowledge. The Home Service Marketplace system provides customer with wide range of utility services they can choose from and book them. The booked service provider will then reach the customers house to provide the selected service like cleaning, plumbing etc.

There are three main actors in the system, the customer, Service Provider and the admin. When a new customer wants to join the platform to avail the services, they should first register/create their account on the platform, where they should add their name and address. After the account is created, the customer will login and search for the service they want. But they can only choose the service provider from their city. Once they book a service provider, the customer must pay the amount and then the customer will receive an OTP which they will verify with the service provider when he arrives. And then after the service is completed the customer can rate him.

For the service provider, if they would like to expand their customer base, they would have to create an account first. While creating the account the service provider must give his name, address (city which he lives in), the skill in which they are providing services, any proof of skill. After creating the account, the service provider will list his services, the service provider is allowed to de-list, modify and delete his existing services. The Service provider will then wait for someone to book him. Once he gets a request the service provider may decide to choose the booking or not. If they do, a notification will be sent to both customer and service provider themselves.

The admin is responsible for maintaining the accounts of both customer and service providers. The admin is also responsible for dispute handling and refunds of payment to customer if the service was not good or the service provider did not show up.

While payment the customer will get discounts and loyalty points based on how many times, they have booked the service using out platform.

The entire system will be made using Java for coding the business logic, spring boot and spring for the MVC architecture and hosting the application. We can check the app health using the spring boot actuator. The system will provide an OTP system to verify the service provider once they reach their destination.

1.2 Case Tools

Eclipse: Eclipse is an integrated development environment (IDE) tool used in software development for developing computer-based applications. Eclipse is open source available from eclipse.org which has the capability of implementing numerous different programming languages.

Postman: Postman is an API visualisation tool for building and testing APIs. It is used to check the output of APIs. It has all the methods for the API to test. We can get output in JSON and different formats.

GitHub: GitHub is an online repository that provides version control of code, CI/CD operations to update new iteration of code.

Draw.io: This is an online website which is used for creating UML diagrams, Analysis diagrams and Communication diagrams as well as flow chart diagram.

2. Software Development Life Cycle (SDLC)

Software development process is a process of planning and managing software development. It typically involves dividing work into smaller steps and finding out a cost-effective way to improve design and project management. there are multiple methods, including Agile, Waterfall, Iterative, and V-Model, each with its own set of practices and principles

Waterfall model is one of the most approachable methods used in the field of Software engineering, it is great at managing small projects and has a distinct review process, but it does not suit projects which require to be flexible. This method does not allow to move back, after completion of the respective phase. Our system needs to be flexible, so waterfall method is ruled out.

For our Home Service Marketplace System Project, we have decided to adopt Agile Methodology for Software development. Agile is well suited for projects which requires Adaptability and Compliancy, as It emphasizes flexibility, allows redundant development with continuous collaboration with the team enabling to incorporate changes during the development process. Agile Iterative cycle or sprint allows us to divide the project into smaller and multiple tasks ensuring better Teamwork and productivity.

Since every teammate is from a different technical background, every one of them had the ability to figure out how they're going to approach tasks on their own and transfer their skills to making progress in the project.

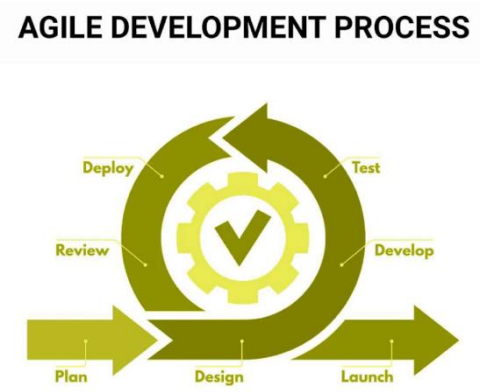


Figure 1: Agile Development Process

We have Structured the project into five main phases:

- Requirement Analysis
- Design
- Development
- Testing
- Deploy

By breaking down the System into smaller components, agile helps us to maintain complication and keep up the continuous Improvement.

We use Teams Planner as an Agile Framework, as it allows us to prioritize task, assign them to team members based on their strength and track progress through sprint planning

We will utilize platform like GitHub for task management and communication, including weekly sprint planning, daily stand-ups and sprint Reviews to ensure smooth co-ordination with the team Member which helps progress through the Project.

3. Project Plan

3.1 Role Assignment

Table 1: Role Assignment table.

	Roles	Description	Team Members
1.	Project Manager	Drives sprint planning and review meetings, gets agreement on the project plan, tracks progress.	Gorav Sharma
2.	Document Manager	Responsible for sourcing relevant supporting documentation from	All

		each team member and composing it in the report.	
3.	Business Analyst/Req. Engineer	Gathers functional and non-functional requirements, use case diagrams & description.	Gorav Sharma
4.	Architect	Defines high level system architecture and gets agreement on technology pipeline	Ashutosh Lembhe
5.	System Analyst	Creates conceptual class model and drives implementation effort.	Achyutam Verma
6.	Designer	Responsible for recovering design time blueprints from implementation.	Dhruv
7.	Technical Lead	Leads the implementation effort, handles daily Scrum stand-up calls.	Ashutosh Lembhe
8.	Programmer	Each team member to develop at least 1 package in the architecture	All
9.	Tester	Drives coding of automated test cases, tracks defect, prioritizes issues.	Achyutam Verma
10.	DevOps	Ensures that each team member is competent with development infrastructure, e.g. GitHub, etc.	Dhruv Upadhyay

3.2 Project Timeline

Table 2: Project Timeline

Week	Task to be Completed
3	Setup team roles, agree on scenario, learn to use GitHub, research on existing projects, etc.
4	Requirements gathering, prepare use case diagrams and description
5	Architecture Pattern and Technology Pipeline planning.
6	Class Diagram, Sequence Diagram, State Chart Diagram, Entity Relationship Diagram Preparation.
7	Phase 1: Learning to use Spring boot and setting up database in MySQL.
8	Phase 2: start development on one use cases and automated test cases.
9	Phase 3: Start developing 2nd use case with design patterns
10	Phase 4: Apply added value code and design patterns wherever it is required.
11	RECOVERED architecture and design blueprints.
12	Documentation and run over.

3.3 Industry Experience

Table 3: Industry Experience

Name	Experience	Domain	Programming Languages	Frameworks
Ashutosh Lembhe (24088714)	2 years	CRM Software development	C#, JavaScript	.Net
Achyutam Verma (24046949)	1 year	Testing	Java	Selenium
Dhruv Upadhyay (24039497)	2 years	CRM Software Development	Java, JavaScript, C#	Junit
Gorav Sharma (24246182)	Fresher	NA	JavaScript, Java	Node.js

4. Projects Requirements.

4.1 Functional requirements

Functional Requirements for Customer:

1. The Customer should be able to Login and Create Account.
2. The Customer should be able to search for services by provided category.
3. The Customer must be able to Update the Booking or should have option to Cancel it.

4. Make sure the payment Gateway should be integrated and Secure.
5. Customer should have option to leave review and rate after the service is done.
6. Customer must receive Email after Confirmed Booking

Functional Requirements for Service Provider:

1. The Service Provider should be able to Register for an account, but it should be Active only when it is Verified & Approved by Admin.
2. The Service Provider should be able to Add, De-list, delete and Modify listed Services
3. Service providers can only create listings in one category.
4. The Service provider should be able to Manage their availability.
5. The service provider should be able to Accept or reject the bookings
6. The service Provider should be able to see the Received Payment.

Functional Requirements for Admin:

1. The Admin Should have a safe Login to Access the Admin Portal
2. Admin Portal should display Option like Total Active Users, Services and Transactions
3. The admin should be able to view, deactivate, or delete both customer and service provider accounts.
4. The admin should have option to reset user passwords.
5. The admin must approve or reject request of service provider Account after reviewing Uploaded document
6. The admin should be able to view and manage all listed services added by service providers.
7. The admin should be able to modify, delete any fraud like Service
8. The admin must be able to see all the transaction between Customer and Provider

4.2 Non-Functional requirements.

Non-Functional Requirements:

1. Customer should have simple interface for browsing and filtering for available services.
2. Customer should have Personal Account management of Create, delete and Update Profile.
3. Customer can Track booking history and can provide service Review.
4. Search Service and filtering should be easy to use.
5. Service Provider should have Management dashboard to List, De-list, delete and modify Offerings.
6. Service provider should have option to Accept/Reject Bookings.
7. Service Provider should be able to receive the feedback and Comments from the Customer
8. Admin should have visibility over all aspects of the system's functioning.
9. Admin can manage the Approval of Service Provider Registration.
10. Admin should have the ability to manage customer and service provider accounts.
11. Admin should handle all the Disputes i.e. - the Refund Issue between Customer and Service Provider.
12. Both Customer and Service Provider receives the Booking confirmation and cancellation Notification.

4.3 Quality Tactics.

Scalability:

When a greater number of customers and service providers will register and start using the platform, the platform should scale and distribute its workload. With spring boot's ability to create microservices and RESTful services when getting OTP and getting notified by the system. Spring boot also can-do load balancing, distributed tracking further aiding in scalability. With scalability we will also need to expand the MySQL database to handle further data. Good thing is MySQL supports both vertical and horizontal scaling. MySQL Cluster and Group Replication allow horizontal scaling with a shared-nothing architecture for distributed databases.

Availability:

Spring Boot applications are perfect for cloud deployment with high availability configurations because they support stateless microservices, which make instance deployment, replication, and restarting easier. This promotes fault tolerance and strong systems. On the other hand, uptime is monitored and maintained by Spring Boot's Actuator module in conjunction with health checks and metrics.

Extensibility:

Spring Boot as framework is quite modular. We can add extensibilities to any existing spring boot project using its autoconfiguration properties. This also helps in connecting third party components. With the use of Annotations, Spring Initializer and plugins we can create lot more than controller and service classes for growing the application.

Serviceability:

Using the Spring boot actuator, we can always check the health of the application. The spring boot actuator has monitoring metric to manage application performance, we can also see the system health in real time. We can add serviceability to our application using it and alert the admins regarding any failure using this actuator.

4.4 Use case Diagrams and Description

4.4.1 Use case Diagrams

1. Customer booking use case diagram.

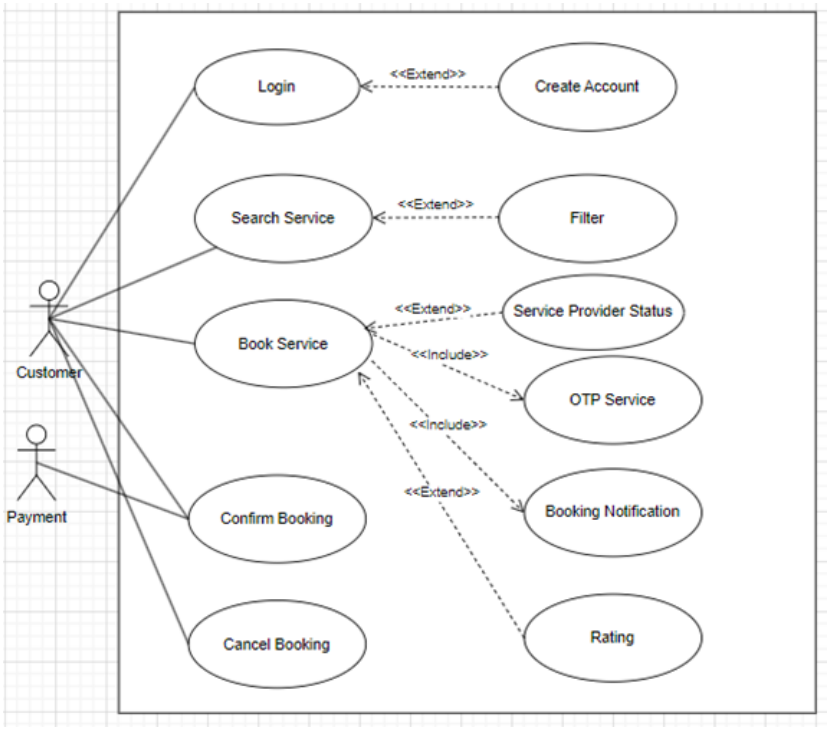


Figure 2: Use case Diagram for customer booking

2. Service Provider use case Diagram:

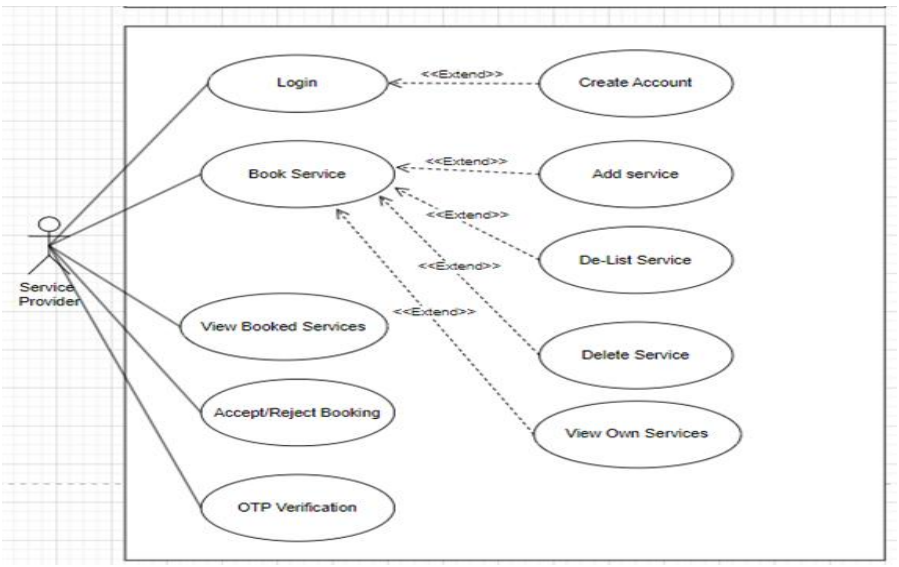


Figure 3: Use case diagram for Service provider

3. Admin use case Diagram:

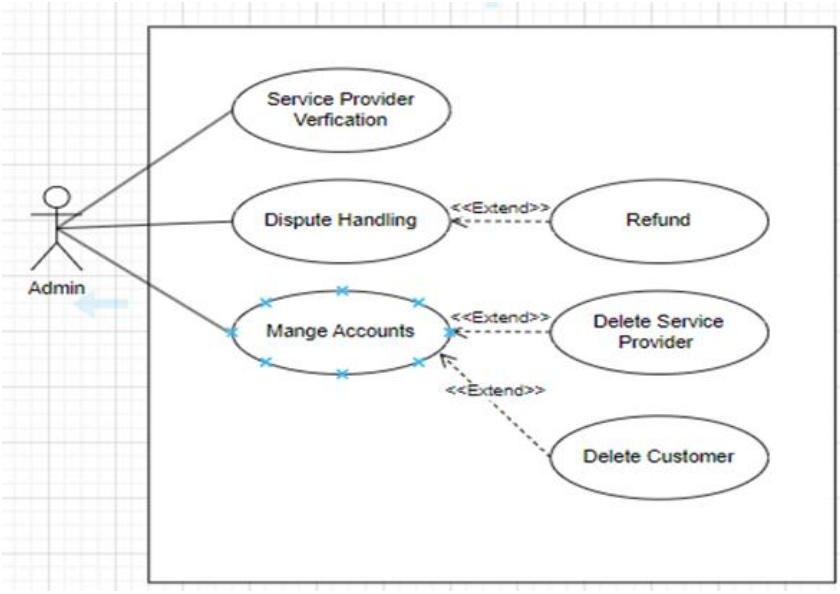


Figure 4: Use case diagram for Admin

4.4.2 Use Case description.

4.4.2.1 Book Service:

Table 4: Use case description for Customer Booking

USE CASE	Customer Booking Service
Goal in Context	Customer searches a service, books a service and confirms it and makes payment.
Scope & Level	System
Preconditions	Customer must be logged in and Service must be available.
Success End Conditions	Customer has booked the service, and payment is processed, and confirmation is sent to customer.
Failed End Condition	Booking not completed or Payment process not completed.
Primary, Secondary, Actors	Customer, Payment system.
Trigger	Customer Initiates booking request.
DESCRIPTION	Step | Action
	1 | Customer searches for required service.
	2 | System will show available services.
	3 | Customer selects required service.
	4 | Customer checks for services availability.
	5 | Customer confirms booking.
	6 | Customer makes payment.
	7 | Customer gets booking notification.
	8 | Customer gets OTP generated.
Extension	Step | Action

	2a	Service not found
	4a	Service not available (service provider busy).
	6a	Payment failure
Priority	Top	
Open Issue	1. What if service provider cancels after booking? 2. How to handle partial refunds? 3. What if OTP delivery fails?	
Release Date	Release 1.0	
Subordinates	1. Payment Processing 2. OTP Service 3. Booking Notification 4. Rating System	

4.4.2.2 Service Provider listing and managing booking.

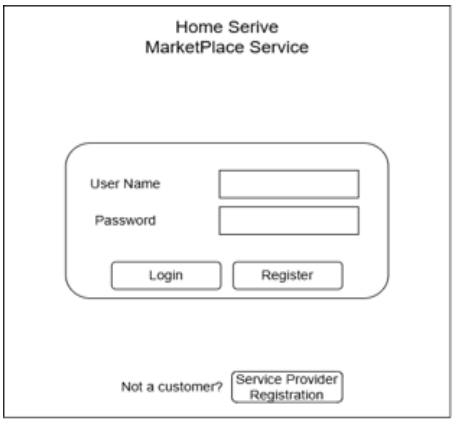
Table 5: Use case Description for Service provider.

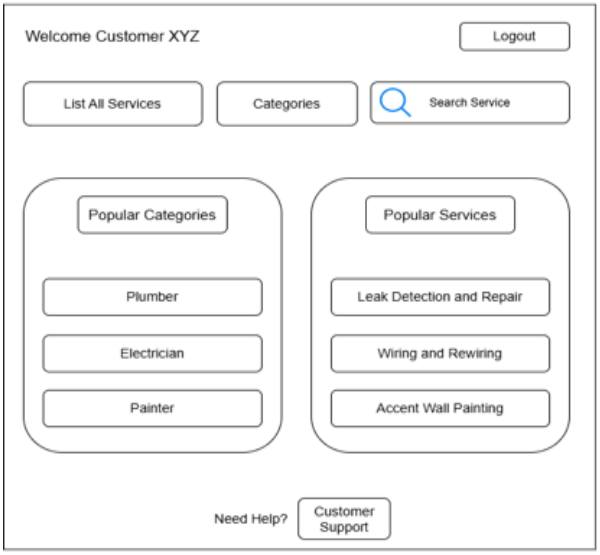
USE CASE	Service Provider Service listing and manging booking	
Goal in Context	Service provider lists services and manages bookings.	
Scope & Level	Service Provider Package, System	
Preconditions	Service Provider must be verified, Service provider must be logged in.	
Success End Conditions	Services are getting listed, bookings are accepted or rejected, Customer gets notified.	
Failed End Condition	Services could not be listed, Booking could not process, OTP verification failed.	
Primary, Secondary, Actors	Service Provider	
Trigger	Service Provider access the listing site.	
DESCRIPTION	Step	Action
	1	Service Provider lists new service.
	2	Service provider checks his own listed service.
	3	Service provider checks booking request.
	4	Service Provider checks booked services.
	5	Service provider accepts/rejects request.
	6	System sends an OTP for verification.
Extension	Step	Action
	2a	Service update, delist or delete
	6a	Booking response (can accept or reject)
Priority	Top	
Open Issue	1. How to handle service provider availability status? 2. What if OTP verification fails? 3. How to manage multiple service locations?	
Release Date	Release 1.0	
Subordinates	1. Service Listing Manager	

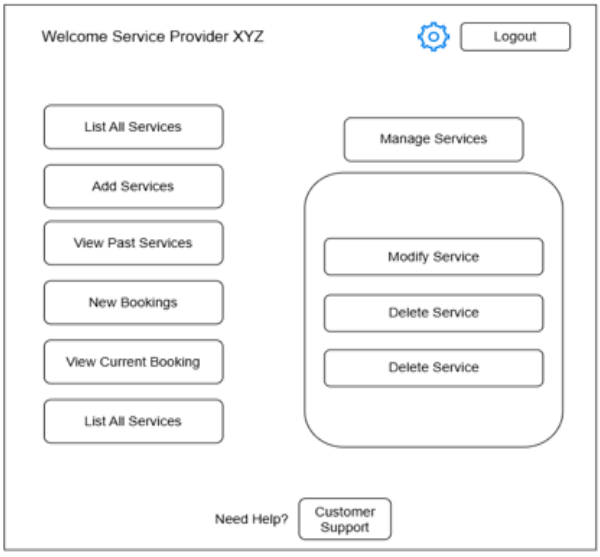
	2. Booking Response Handler 3. OTP Verification System

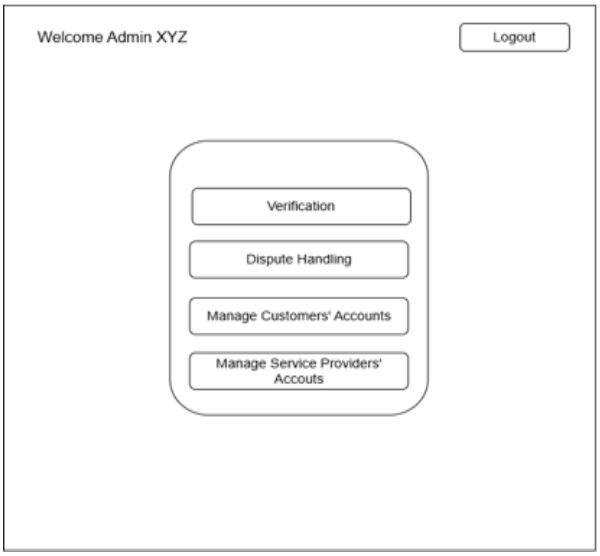
4.5 GUI Prototype:

Figure 5: GUI prototypes for Project









5. Architectural Pattern

5.1 Layered Architecture

The N-tier or layered Architecture is most used architectural pattern. The N-tier architecture consists of individual layers that work as a single software. Each layer is a has an abstraction layer for its components, also every layer is different and has different function to different parts of the overall system. The role of each layer is specified that it must perform. The layered or N-tiered architecture does not specify the number and types that must exists in its layer.

The architecture is like client server architecture which has presentation layer, Business Layer and database layer. Depending upon the complexity and scale of the application layers in this architecture can vary. For our application we have decided to keep 3 – layers.

The presentation layer in our application would be responsible for handling all user interface and browser communication logic for customer, Service Provider and Admin. The business layer is responsible for containing all the business logic for the application to follow when a particular function is executed. The database layer has the database access logic for accessing data when an API or request is invoked.

The layers also provide abstraction for its component that needs to satisfy a particular business request. The presentation layer does not need to know how it gets the data, it just needs to show the data to the end user.

Benefits of N-Tier Architecture:

1. Low coupling:
    a. It should not be necessary for the presentation tier (UI) to understand how the data tier maintains the database or how the business logic tier processes data.
    b. Modifications to one layer, like the data tier (the database's data storage), should have little effect on other tiers, such as the user interface.
2. High Cohesion:
    a. Rendering the user interface and managing user input should be the only priorities of the presentation tier.
    b. Only business rules, application logic, and processing should fall under the purview of the business logic tier; UI and data issues should be kept separate.
    c. Without integrating business logic, the data tier should be devoted to data management tasks including database interfaces, queries, and storage.

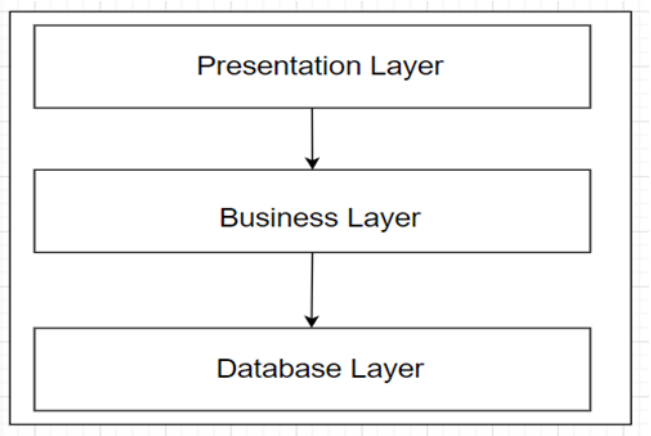


Figure 6: Layered Architecture.

5.2 MVC Architecture.

The Model-View-Controller architectural pattern is a widely used pattern for developing web applications. The MVC architecture is divided into three components Model, View, Controller. Each component has separate functionality of how data is presented or controlled. Like the Layered architecture MVC also provides abstraction for each component.

Model: The data logic and data saved in your application are represented by the model. This could be a straightforward data structure like a dictionary or array, or it could be a SQL database. Information from one or more events, like user input or database queries, is usually stored in the model. The model also contains the database access logic.

View: The view is application user interface from where the user will interact with the application. The view gives user input to the controller which takes data from model and shows it on view.

Controller: A component situated between the view and the model is the controller. It controls how users interact with the view and executes any necessary logic to prepare the data for display. For example, if a user clicks on a button in the user interface of your program, the controller will handle the event in the services. This information may subsequently be used by the controller to get data from the database or perform other tasks.

Benefits:

1. Low Coupling:
    a. Since the View only cares about obtaining data to display and not how it is stored or processed, it is unaffected if you make changes to the database schema in the Model.
    b. The Model and Controller do not need to be altered if the View's user interface is redesigned because the View's sole function is to display data; it has no bearing on how it is handled or processed.
    c. Since the Model and View communicate with the Controller via interfaces or contracts, they stay the same even if you update the Controller to handle a different kind of input or logic.

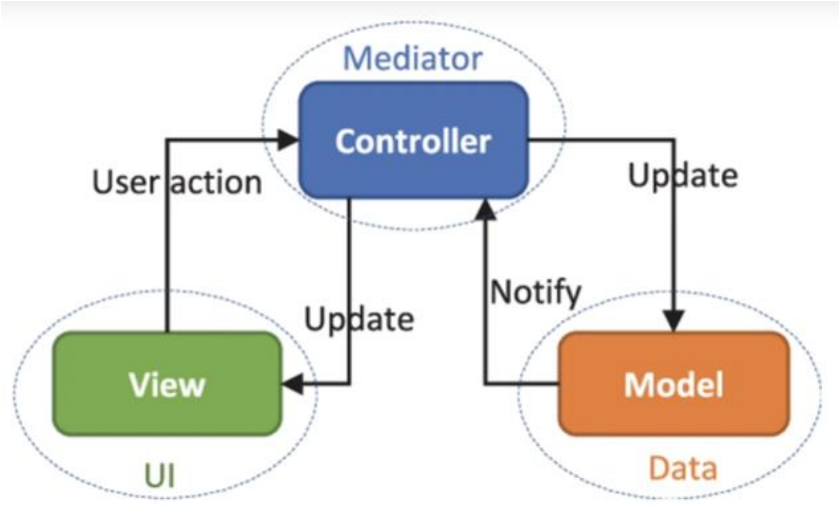


Figure 7: Working of MVC architecture.

5.3 Combined High Level Architecture Diagram:

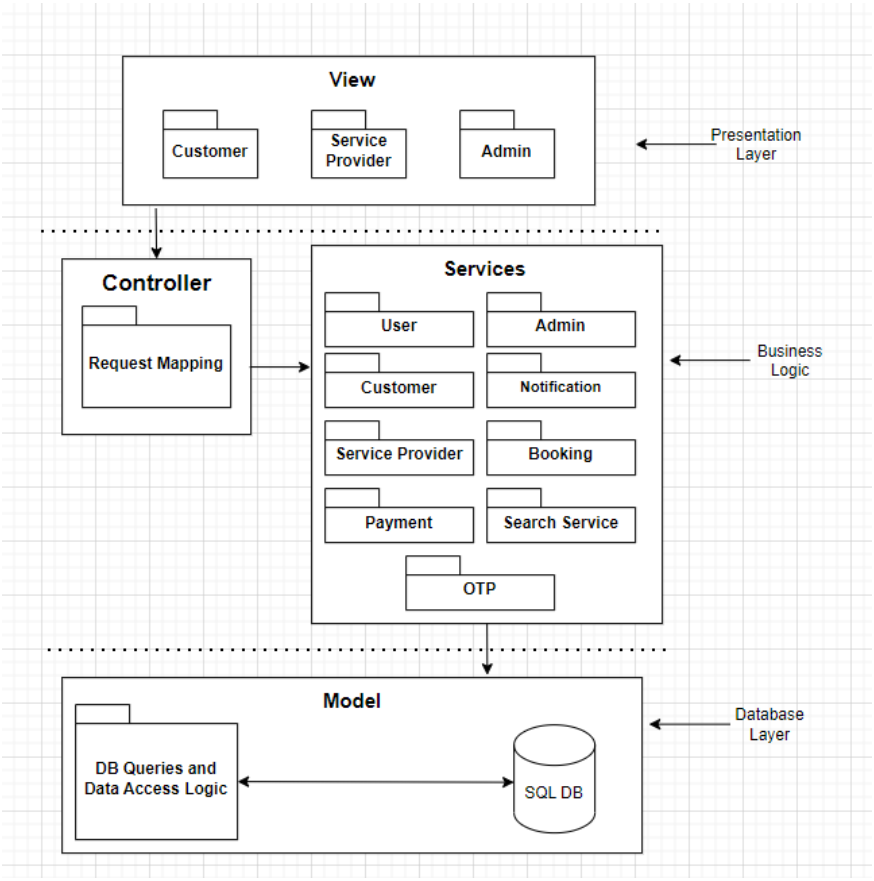


Figure 8: MVC Architecture combined with Layered Architecture.

Technology Pipeline:

- **Development**: Java, Spring boot – Handles real time backend processes like booking, searching, payment, resulting in a highly responsive system due to its architecture.

- **Database**: MySQL - The database management system (DBMS) MySQL is an open-source database. Although MySQL started out as a low-end substitute for more potent proprietary databases, it has progressively developed to accommodate larger requirements as well. By installing MySQL on more potent hardware, like a server with lots of memory, MySQL can be scaled in the medium range.

- **Quality Control:** SonarQube - Sonar Source created SonarQube, an open-source platform for automatic reviews and static code analysis to identify errors and code smells to continuously inspect the quality of code.

- **Version Control:** GitHub – For every project, GitHub offers task management, software feature requests, issue tracking, version control, and CI/CD while writing code.

- **Test Case Framework:** JUnit, Postman API Testing – Creating test cases for API's and backend code for booking, payment etc. Ensuring minimal bugs.

6. Conceptual/Initial design.

6.1 Noun Identification

Customer: Represents users who book services.
Admin: Manages accounts and services.
ServiceProvider: Provides services to customers.
SearchService: Allows customers to search and view services.
Booking: Represents a service booking.
Notification: Manages updates sent to users.
Payment: Handles transaction details for bookings.
OTPService: Manages OTP generation and verification

6.2 Class Diagram with Design Patterns.

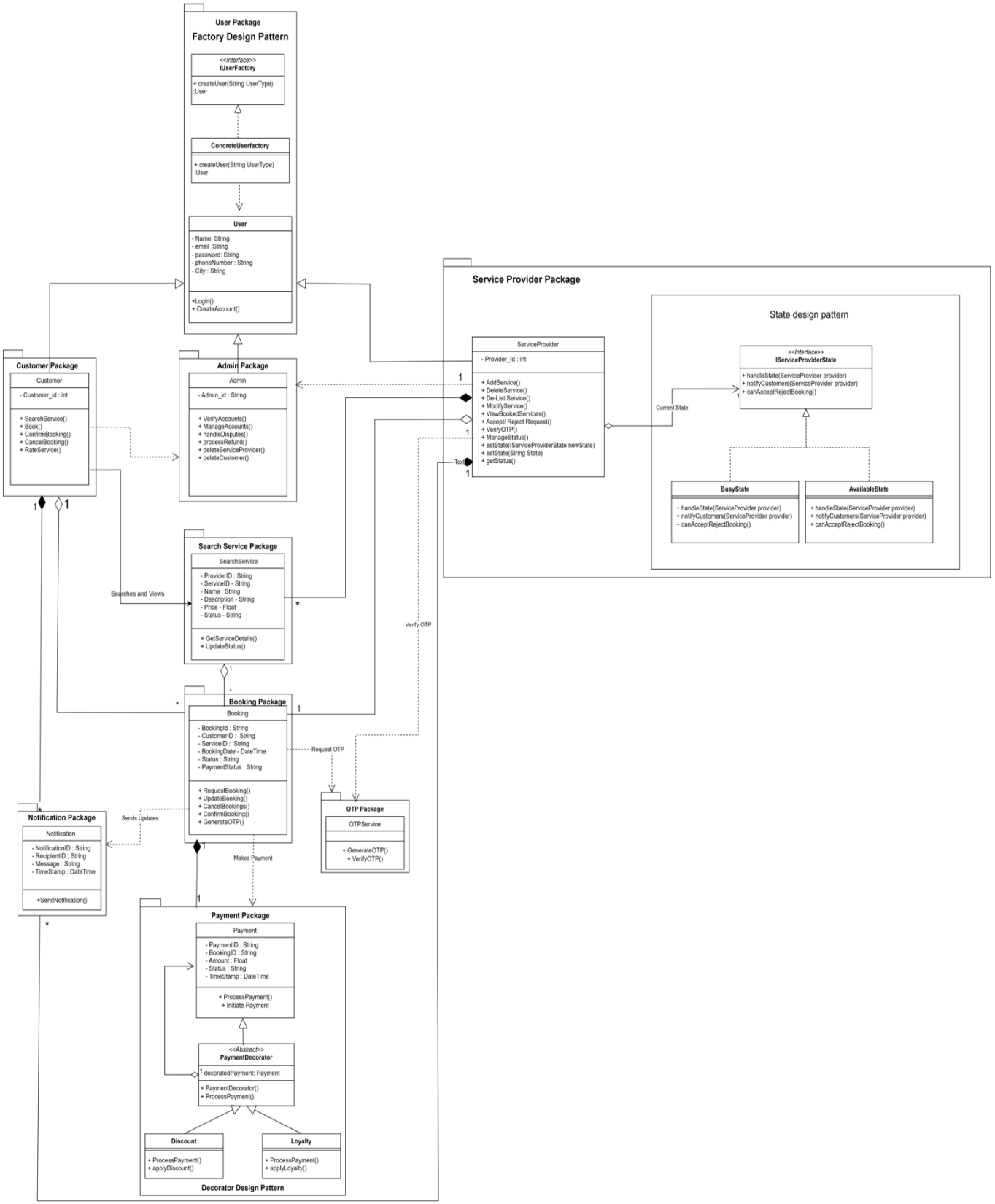


Figure 9: Analysis Time Class Diagram

6.3 Sequence Diagram

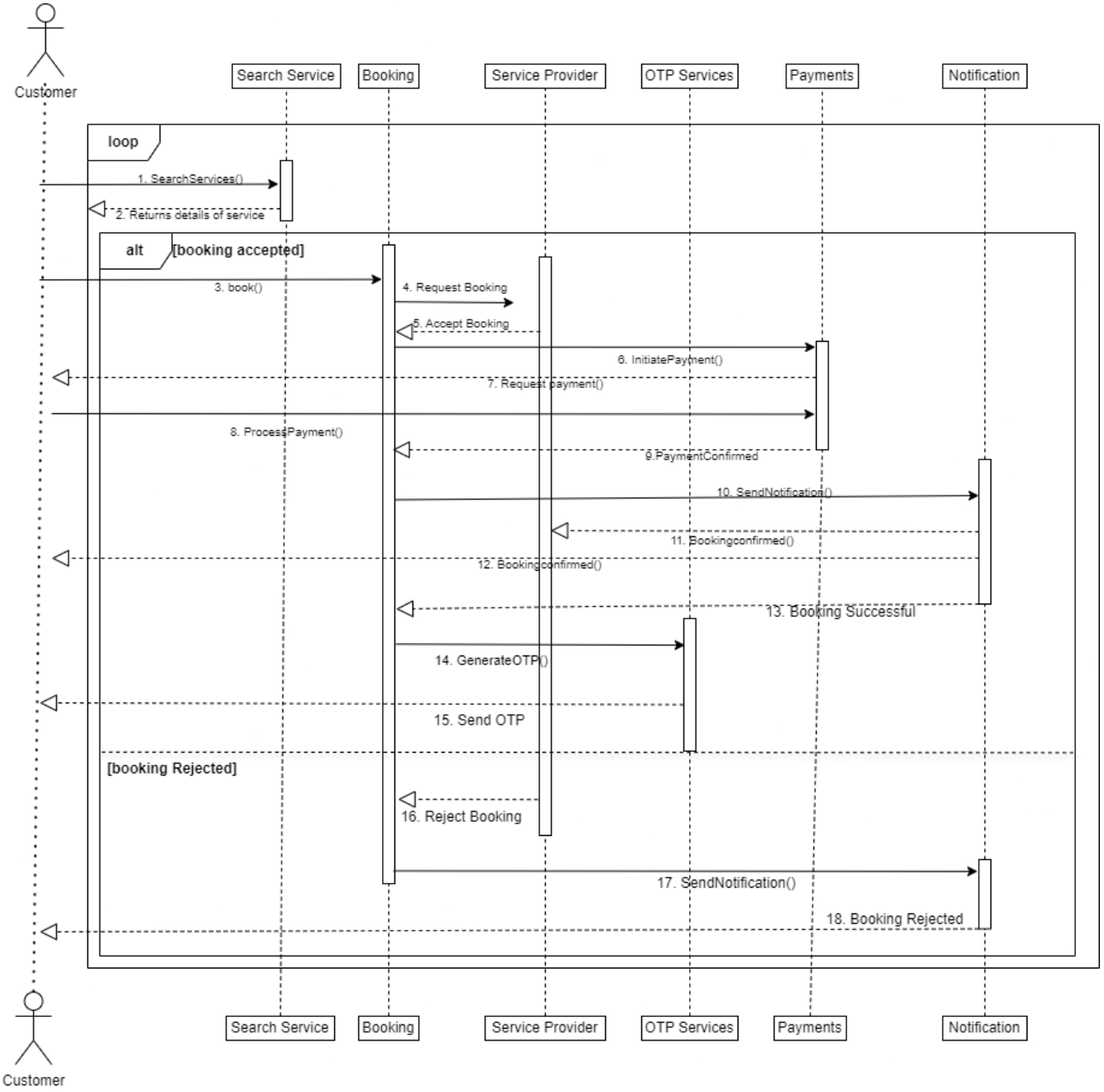


Figure 10: Analysis Time Sequence Diagram.

6.4 State Chart Diagram

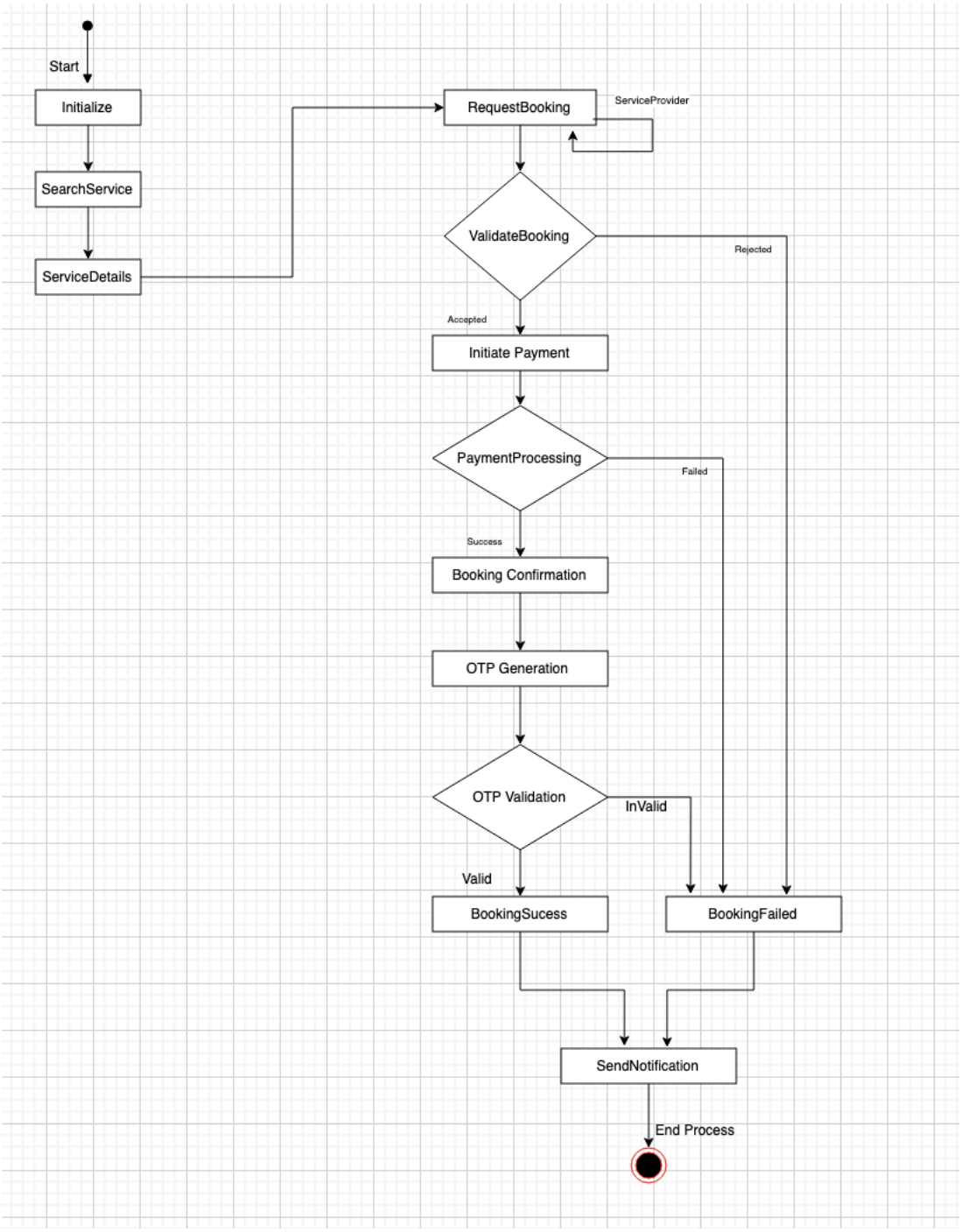


Figure 11: Analysis Time State Diagram

6.5 Entity Relationship Diagram

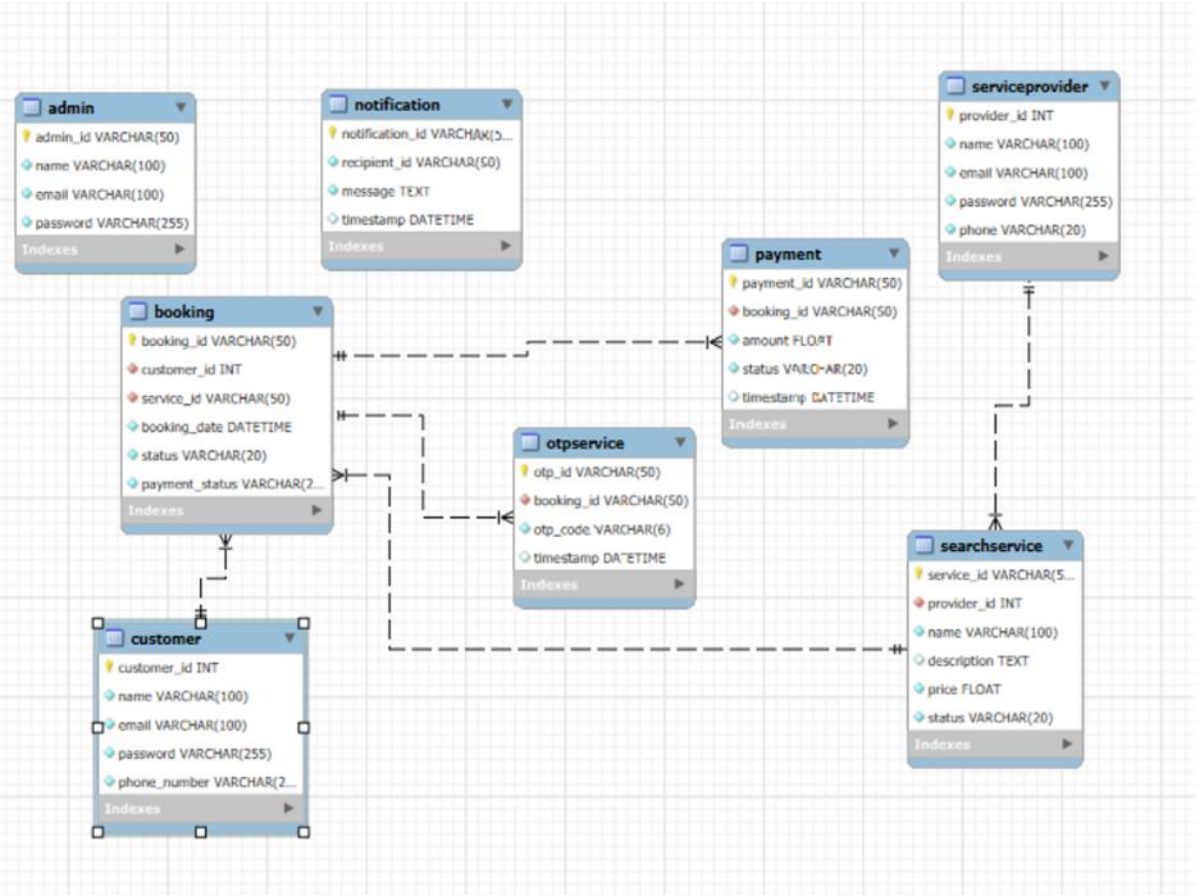


Figure 12: Analysis Time Entity Diagram.

7. Transparency and Traceability

Table 6: Code Diary 1

Name	**Achyutam Verma**
Package	• com.example.demo.Model.OTPService • com.example.demo.Service.ServiceProvider • com.example.demo.model.booking • Com.example.demo.Service.OTP • Com.example.demo.Model.OTPService • com.example.demo.Controller (test)
Class	**Model** • OTPServiceModel.java • OTPServiceMapper.java • Booking.java • BookingRowMapper.java • ServiceProviderBookingDTO.java • IPaymentSubject.java • IServiceProviderObserver.java **DAORepo** • ServiceProvider_Repository.java • OTPRepository.java **Controller** • ServiceProvider_Controller.java **Services** • ServiceProviderService.java • OTPService.java **Junit Test** • ServiceProvider_ControllerTest.java • SearchServiceControllerTest.java
View	• VerifyOTP.html • ServiceProviderRegistration.html • ServiceProvider_login.html • PastBooking.html
Line of Code	**1095**
Design Pattern	• Observer Design Pattern

Table 7: Code Diary 2

Author	**Dhruv Upadhyay**
Package	• com.example.demo.Service.ServiceProvider • com.example.demo.Model.ServiceProvider • com.example.demo.Controller (test) **Controller and Repository are common packages**
Class	**Model** • AvailableStatusState.java • BusyStatusState.java • IServiceProviderStatusState.java • IServiceProviderSubject.java • ServiceProvider.java • ServiceProviderRowMapper.java • ServiceProviderStateManager.java • ICustomerObserver.java **DAORepo** • ServiceProvider_Repository.java **Controller** • ServiceProvider_Controller.java **Services** • GlobalContext.java • ServiceProviderService.java **Junit Test** • ServiceProvider_ControllerTest.java
View	• AddServiceForm.html • all-services.html • ServiceProviderWelcomeScreen.html • ServiceProviderBookedServices.html • ModifyService.html • ListServices.html • CustomerCurrentBookedServices.html
Line of Code	**1097**
Design Pattern	• State Design Pattern • Observer Design Pattern

Table 8: Code Diary 3

Author	**Ashutosh Lembhe**
Package	**Model**: • com.example.demo.Model.Customer • com.example.demo.Model.SearchServices • com.example.demo.Model.User **Service**: • com.example.demo.Service.Customer • Com.example.demo.Service.SearchService **Controller and Repository are common packages**
Class	**Model**: • User.java • IUserFactory.java • ConcreteUserFactory.java • Customer.java • CustomerRowMapper.java • RatingRowMapper.java • SearchService.java • SearchServiceFilterContext.java • SearchServiceRowMapper.java • IFilterStrategy.java • CityStrategy.java • SkillStrategy.java • StatusStrategy.java **Controller**: • CustomerController.java • SearchServiceController.java **Service:** • CustomerService.java • SearchServiceService.java **DAORepo:** • CustomerRepository.java • SearchServiceRepository.java **JUNIT Test:** • CustomerControllerTest.Java
View	• all-service.html • Bookingscreen.html • CustomerLogin.html • CustomerPastService.hmtl • CustomerRegistration.html • CustomerWelcomeScreen.html • Filter-services.html
Line of Code	1838
Design Pattern	• Factory Design Pattern • Strategy Design Pattern

Table 9: Code Dairy 4

Author	**Gorav Sharma**
Package	**Model**: • com.example.demo.Model.Admin • com.example.demo.Model.Payment • com.example.demo.Model.Wallet **Service**: • com.example.demo.Service.Admin • com.example.demo.Service.Payment • com.example.demo.Service.WalletService
Class	**Model**: • Admin.java • ServiceProviderRowMappperForAdmin.java • BasePayment.java • DiscountDecorator.java • LoyaltyPointsDecorator.java • Payment.java • PaymentDecorator.java • Wallet.java **Controller**: • AdminController.java • PaymentController.java • WalletController.java **Service:** • AdminService.java • PaymentService.java • WalletService.java **DAORepo:** • AdminRepository.java • PaymentRepository.java • WalletRepository.java
View	• ManageAccountScreen.html • paymentForm.html • paymentResult.html • Wallet.html
Line of Code	**938**
Design Pattern	• Decorator Design Pattern

8. Code and Implementation

8.1 Model-View-Controller

The Model-View-Controller architectural pattern is a widely used pattern for developing web applications. The MVC architecture is divided into three components Model, View, Controller. Each component has separate functionality of how data is presented or controlled. Like the Layered architecture MVC also provides abstraction for each component.

Model: The data logic and data saved in your application are represented by the model. This could be a straightforward data structure like a dictionary or array, or it could be a SQL database. Information from one or more events, like user input or database queries, is usually stored in the model. The model also contains the database access logic.

View: The view is application user interface from where the user will interact with the application. The view gives user input to the controller which takes data from model and shows it on view.

Controller: A component situated between the view and the model is the controller. It controls how users interact with the view and executes any necessary logic to prepare the data for display. For example, if a user clicks on a button in the user interface of your program, the controller will handle the event in the services. This information may subsequently be used by the controller to get data from the database or perform other tasks.

We have implemented different classes for all model, view, Controller and Repository**.**

Model: The model class will contain the attributes and other fields which will be used on the views and column names same as attributes. In the below image is the user class model. This model class will be extended by customer, admin, ServiceProvider in our class.

```
2
3  import org.springframework.stereotype.Component;
4
5  @Component
6  public abstract class User {
7      protected String name;
8      protected String email;
9      protected String password;
10     protected int phone_number;
11     protected String city;
12     protected int id;
13
14     //This method will get override since it will be used by the Admin, Customer and ServiceProvider
15     public abstract String getUserType();
16
17     //We need getter and setters to get the value of users
18     //Since both customer and service provider need the same basic info
19     public String getName() {
20         return name;
21     }
22     public void setName(String name) {
23         this.name = name;
24     }
25     public String getEmail() {
26         return email;
27     }
28     public void setEmail(String email) {
29         this.email = email;
30     }
31     public String getPassword() {
32         return password;
33     }
34     public void setPassword(String password) {
```

Figure 13: Model/Entity Code for User

View: Here we have created thyme leaf HTML pages for our views and for the user to interact with. Like search services, create account, login to account. These views are connected to the controller. And call the controller using the Post and Get method. Below is the HTML code for one of views which shows All the services, and its corresponding view in localhost.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Available Services</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
</head>
<body>
<h2>Available Services</h2>
<h3>Customer Information</h3>
<p>Name: <span th:text="${customer.name}"></span></p>
<p>Customer City: <span th:text="${customer.city}"></span></p>
<form th:action="@{/Customer/filterservice}" method="get">
    <label>Search by Skill:</label>
    <input type="text" name="skill" placeholder="e.g., plumber" />

    <label>Search by City:</label>
    <input type="text" name="city" placeholder="e.g., Limerick" />

    <label>Search by Status:</label>
    <input type="text" name="status" placeholder="e.g., free or busy" />

    <button type="submit">Search</button>
</form>

<table>
    <thead>
        <tr>
            <th>Service ID</th>
            <th>Provider ID</th>
            <th>Skill</th>
            <th>Category</th>
            <th>Rating</th>
            <th>Price</th>
```

Figure 14: HTML code View for Available services in thyme leaf

```
            <th>Price</th>
            <th>Status</th>
            <th>Provider Name</th>
            <th>City</th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="service : ${services}">
            <td>
                <form th:action="@{/Customer/book}" method="get">
                  <input type="hidden" name="serviceId" th:value="${service.serviceId}" />
                   <input type="hidden" name="customerCity" th:value="${customer.city}" />
                   <input type="hidden" name="serviceCity" th:value="${service.city}" />
                   <input type="hidden" name="serviceStatus" th:value="${service.status}" />
                   <input type="hidden" name="serviceSkill" th:value="${service.skill}" />
                   <input type="hidden" name="serviceCategory" th:value="${service.category}" />
                   <input type="hidden" name="serviceRating" th:value="${service.rating}" />
                   <input type="hidden" name="servicePrice" th:value="${service.price}" />
                   <input type="hidden" name="serviceProviderName" th:value="${service.providerName}" />
                   <button type="submit" class="btn btn-primary">Book</button>
                </form>
            </td>
            <td th:text="${service.serviceId}">Service ID</td>
            <td th:text="${service.providerId}">Provider ID</td>
            <td th:text="${service.skill}">Skill</td>
            <td th:text="${service.category}">Category</td>
            <td th:text="${service.rating}">Rating</td>
            <td th:text="${service.price}">Price</td>
            <td th:text="${service.status}">Status</td>
            <td th:text="${service.providerName}">Provider Name</td>
            <td th:text="${service.city}">City</td>
        </tr>
    </tbody>
```

Figure 15: HTML code View for Available services in thyme leaf

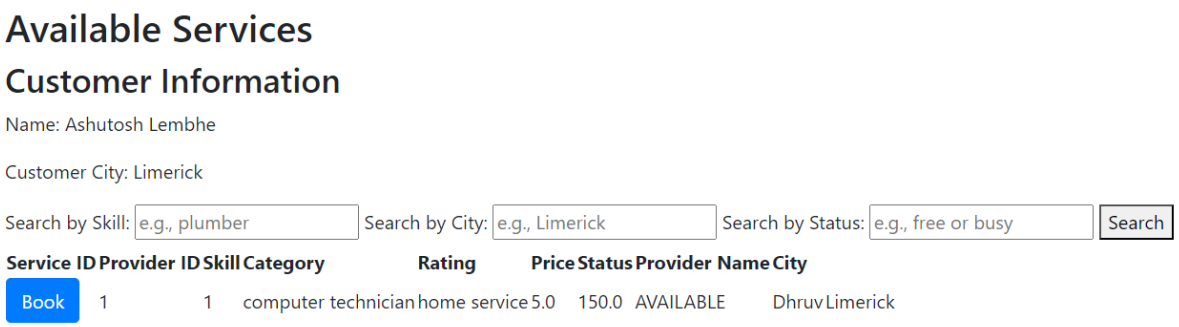


Figure 16: Web browser View.

Controller: The controller is invoked whenever a user action is done on the view. Usually, the controller Api is places on the button in our view. In the above code for our view, we can see that Api is getting called for the form action submit and the method used is get.

```
20
21 @Controller
22 @RequestMapping("/Customer")
23 public class CustomerController {
```

Figure 17: Controller and request mapping code snippet

```
    @GetMapping("/book")
    //@ResponseBody
    public String bookservice(@RequestParam String serviceId, @RequestParam String customerCity, @RequestParam String serviceCity,
            @RequestParam String serviceStatus,@RequestParam String servicePrice, @RequestParam String serviceSkill,
            @RequestParam String serviceRating, @RequestParam String serviceProviderName, @RequestParam String serviceCategory,Model model) {
        String response=customerService.checkServiceParameters(customerCity,serviceCity,serviceStatus);
        if(response=="success")
        {
            model.addAttribute("serviceId",serviceId);
            model.addAttribute("customerCity", customerCity);
            model.addAttribute("serviceCity", serviceCity);
            model.addAttribute("serviceStatus", serviceStatus);
            model.addAttribute("servicePrice", servicePrice);
            model.addAttribute("serviceSkill", serviceSkill);
            model.addAttribute("serviceRating", serviceRating);
            model.addAttribute("serviceProviderName", serviceProviderName);
            model.addAttribute("serviceCategory", serviceCategory);
            return "bookingscreen";
        }
        else
        {
            return response;
        }
    }
```

Figure 18: GetMapping inside of Controller

Inside our controller we have different RequestMapping for different functionalities. These different request mapped functions are called from our views.

Service: The service class usually holds the business logic for our project. And is in the middle of the repository and controller, whenever our repository returns some response, or our controller sends some input. The service class checks the response or input and uses if and else condition to then call the and send the necessary feedback. Inside our service class we have different services which get called.

```java
@Service
public class CustomerService {


//Login logic check
public int checkCustomerLogin(String name, String password) {
    List<Map<String, Object>> response= customerRepository.loginCustomer(name,password);
    if(!response.isEmpty())
    {
        Map<String,Object> firstResult=response.get(0);
        String getname=(String)firstResult.get("name");
        System.out.println(getname);
        String storedHashedPassword=(String)firstResult.get("password");
        System.out.println(storedHashedPassword);
        BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
        boolean passwordMatches = passwordEncoder.matches(password, storedHashedPassword);
        if(!getname.equals(name) || passwordMatches!=true) {
            return 0;
        }
        else
        {
            return 1;
        }
    }
    else
    {
        return 0;
    }
}
```

Figure 19: Service Class Code with Business Logic

Here we have the service to check the logic of our customer login. If we have customers credentials, we will allow them to login.

Repository: The repository class holds the query, that will be used to perform CRUD Operations on our database and will eventually be used for updating our views. Like every other view, here also we have different functions which have queries within them for different operations.

```java
@Repository
public class CustomerRepository {


//Query to check for login with and password
public List<Map<String, Object>> loginCustomer(String name, String password){
    String query = "SELECT name, password FROM customer WHERE LOWER(name) = LOWER(?)";
    return queryTemplate.queryForList(query,new Object[]{name});
}

//Query to display the name on the html page
public Customer getCustomerByNamequery(String name) {
    String query = "SELECT * FROM customer WHERE name = ?";
    List<Customer> customers= queryTemplate.query(query, new CustomerRowMapper(),name);
    return customers.isEmpty() ? null : customers.get(0);
}

public List<Customer> findCustomerByCity(String city) {
    String query = "SELECT * FROM customer WHERE city = ?";
    List<Customer> customers= queryTemplate.query(query, new CustomerRowMapper(),city);
    return customers;
}
```

Figure 20: Repository Class with SQL queries to perform CRUD operations in the database.

8.2 Design Pattern Implemented

8.2.1 Factory Design Pattern

We are implementing Factory design pattern to create records for admin, customer and Service Provider. We are using abstract user class to inherit the properties so that the same properties do not repeat while an object is created for each admin, customer and service provider. We are using factory method since we are not sure which class object to initialize during runtime. We also don't want other people to know how the initializing of object is happening.

Main Product Class: abstract User class

From this class our admin, customer and service provider will be inherited.

```
2
3  import org.springframework.stereotype.Component;
4
5  @Component
6  public abstract class User {
7      protected String name;
8      protected String email;
9      protected String password;
10     protected int phone_number;
11     protected String city;
12     protected int id;
13
14     //This method will get override since it will be used by the Admin, Customer and ServiceProvider
15     public abstract String getUserType();
16
17     //We need getter and setters to get the value of users
18     //Since both customer and service provider need the same basic info
19     public String getName() {
20         return name;
21     }
22     public void setName(String name) {
23         this.name = name;
24     }
25     public String getEmail() {
26         return email;
27     }
28     public void setEmail(String email) {
29         this.email = email;
30     }
31     public String getPassword() {
32         return password;
33     }
34     public void setPassword(String password) {
```

Figure 21: Abstract class User as Super class (product)

Concrete Product Class: Customer, Admin, ServiceProvider

```
1  package com.example.demo.Model.Customer;
2  import org.springframework.stereotype.Component;
5
6  @Component
7  public class Customer extends User implements ICustomerObserver{
8
9      @Override
10     public String getUserType() {
11         return "Customer";
12     }
13
14     @Override
15     public void update(String message, String email) {
16         // TODO Auto-generated method stub
17         System.out.println("Sending notification to " + email + ": " + message);
18     }
19
20 }
21
```

Figure 22: child class customer (concrete product)

```java
package com.example.demo.Model.Admin;

import org.springframework.stereotype.Component;

@Component
public class Admin extends User{

    @Override
    public String getUserType() {
        return "Admin";
    }
}
```

Figure 23: child class Admin (concrete Product)

```java
package com.example.demo.Model.ServiceProvider;
import java.util.ArrayList;

@Component
public class ServiceProvider extends User implements IServiceProviderSubject, IServiceProviderObserver{

        private String providerId;
        private String currentState;
        private IServiceProviderStatusState state;
        private String status;

        private final List<Customer> observers = new ArrayList<>();


        public void setState(IServiceProviderStatusState state) {
             if (state == null) {
                    throw new IllegalArgumentException("State cannot be null");
                }

            this.state = state;
            this.currentState = state.getStateName(); // Update state name
            notifyObservers("Service Provider state changed to: " + state.getStateName());
        }


        public void handleState(ServiceProvider_Repository serviceProviderRepository) {
            state.handleStatusState(this, serviceProviderRepository);
        }

        public boolean canAcceptBooking() {
            return state.canAcceptRejectBooking();
        }
```

Figure 24: child class service provider (Concrete Product)

Creator/Factory Class: IUserFactory.

This interface is the creator/factory class which intializes the product class that needs to be used while instantiating the concrete product.

```java
package com.example.demo.Model.User;
public interface IUserFactory {
    User createUser(String UserType);
}
```

Figure 25: Factory Interface.

Concrete Creator/Factory Class: ConcreteUserFactory

This class will create the admin, customer and service provider objects when they are called from the controller.

```java
package com.example.demo.Model.User;
import org.springframework.stereotype.Component;

@Component
public class ConcreteUserFactory implements IUserFactory {

 @Override
 public User createUser(String userType) {
     switch (userType) {
         case "Customer":
             System.out.println("Inside customer factory");
             return new Customer();
         case "Admin":
             System.out.println("Inside Admin factory");
             return new Admin();
         case "ServiceProvider":
             System.out.println("Inside Servide Provider factory");
             return new ServiceProvider();
     }
    return null;
 }
}

```

Figure 26: Concrete Factory which creates concrete product.

8.2.2 Strategy Design Pattern

The strategy design pattern is used to for filter search functionality in our project. This design pattern helps filter only those items for what we are searching. Eg. If we are searching for limerick city services, only those will show up. Same way the filter is applied for status and skills. This design pattern allows me to change the behaviour of my objects by encapsulating them, allows user to choose through different strategies at runtime.

Strategy Class: IFilterStrategy

```java
package com.example.demo.Model.SearchServices;

import java.util.List;

public interface IFilterStrategy {
    String applyFilter(StringBuilder sql, List<Object> params);
}

```

Figure 27: Strategy Interface for filter strategy.

Context class: SearchServiceFilterContext

This class will help delegate the function from the main strategy interface.

```java
package com.example.demo.Model.SearchServices;

import java.util.ArrayList;

public class SearchServiceFilterContext {
     private List<IFilterStrategy> strategies = new ArrayList<>();

        public void addStrategy(IFilterStrategy strategy) {
            strategies.add(strategy);
        }

        public String applyFilters(StringBuilder sql, List<Object> params) {
            for (IFilterStrategy strategy : strategies) {
                strategy.applyFilter(sql, params);
            }
            return sql.toString();
        }
}
```

Figure 28: Context class to delegate functionality of the main class.

Concrete Strategy Classes: CityStrategy, SkillStrategy, StatusStrategy.

```java
package com.example.demo.Model.SearchServices;

import java.util.List;

public class CityStrategy implements IFilterStrategy{
    private String city;
    public CityStrategy(String city) {
        this.city = city;
    }

    @Override
    public String applyFilter(StringBuilder sql, List<Object> params) {
        if (city != null && !city.isEmpty()) {
            sql.append(" AND LOWER(city) = LOWER(?)");
            params.add(city);
        }
        System.out.println("City based filter strategy");
        return sql.toString();
    }

}
```

Figure 29: Concrete Strategy class CityStrategy

```java
package com.example.demo.Model.SearchServices;

import java.util.List;

public class SkillStrategy implements IFilterStrategy{

    private String skill;

    public SkillStrategy(String skill) {
        this.skill = skill;
    }

    @Override
    public String applyFilter(StringBuilder sql, List<Object> params) {
        if (skill != null && !skill.isEmpty()) {
            sql.append(" AND LOWER(skill) = LOWER(?)");
            params.add(skill);
        }
        System.out.println("Skill based filter strategy");
        return sql.toString();
    }
}

```

Figure 29: Concrete Strategy class SkillStrategy

```java
package com.example.demo.Model.SearchServices;

import java.util.List;

public class StatusStrategy implements IFilterStrategy {
    private String status;

    public StatusStrategy(String status) {
        this.status = status;
    }

    @Override
    public String applyFilter(StringBuilder sql, List<Object> params) {
        if (status != null && !status.isEmpty()) {
            sql.append(" AND LOWER(status) = LOWER(?)");
            params.add(status);
        }
        System.out.println("Status based filter strategy");
        return sql.toString();
    }
}
```

Figure 30: Concrete Strategy class StatusStrategy

8.2.3 Observer Design Pattern

Observer design pattern is implemented for notifying all the customers of a city when Status of Service provider is changed.

Observer: ICustomerObserver

ICustomerObserver.java

```java
package com.example.demo.Model.Customer;

public interface ICustomerObserver {
     void update(String message, String email);
}

```

Figure 31: Interface for Observer

Concreate Observer: Customer

Customer.java

```java
package com.example.demo.Model.Customer;
import org.springframework.stereotype.Component;

@Component
public class Customer extends User implements ICustomerObserver{

    @Override
    public String getUserType() {
        return "Customer";
    }

    @Override
    public void update(String message, String email) {
        System.out.println("Sending notification to " + email + ": " + message);
    }
}

```

Figure 32: Concrete observer class customer.

Subject: IServiceProvider

IServiceProviderSubject.java

```java
package com.example.demo.Model.ServiceProvider;

import com.example.demo.Model.Customer.Customer;

public interface IServiceProviderSubject {
    void addObserver(Customer observer);
    void notifyObservers(String message);
}

```

Figure 33: Subject Interface IServiceProviderSubject

Concreate Subject: Service Provider

```java
@Component
public class ServiceProvider extends User implements IServiceProviderSubject{

        private String providerId;
        private String currentState;
        private IServiceProviderStatusState state;
        private String status;

        private final List<Customer> observers = new ArrayList<>();


        public void setState(IServiceProviderStatusState state) {
             if (state == null) {
                    throw new IllegalArgumentException("State cannot be null");
                }

            this.state = state;
            this.currentState = state.getStateName(); // Update state name
            notifyObservers("Service Provider state changed to: " + state.getStateName());
        }
```

Figure 34: Concrete subject class ServiceProvider

Usage Class: ServiceProviderStateManager

provider.setState() method of ServiceProvider is triggering notify observer method. (Screenshot attached above)

ServiceProviderStateManager.java

```java
        this.availableState = availableState;
    }

    public void changeState(ServiceProvider provider, String stateName, List<Customer> customers) {

        for (Customer customer : customers) {
            provider.addObserver(customer);
        }

        switch (stateName.toUpperCase()) {
        case "COMPLETED":
            // availableState.updateServicesStatus(provider, serviceProviderRepository);
            provider.setState(availableState);
            break;
        case "ACCEPTED":
            // busyState.updateServicesStatus(provider, serviceProviderRepository);
            provider.setState(busyState);
            break;
        default:
            throw new IllegalArgumentException("Unknown state: " + stateName);
        }
        provider.handleState(serviceProviderRepository);
    }

}
```

Figure 35: ServiceProviderStateManager, Context class used for adding observer and notifying

---

8.2.4 State Design Pattern

State design pattern is used to manage status of service provider from AVAILABLE to BUSY, when booking is completed or accepted. For example, when booking is completed status changes to available, and when booking is accepted status changes to Busy.

State Interface: IServiceProviderStatusState

IServiceProviderStatusState.java

```java
package com.example.demo.Model.ServiceProvider;

import com.example.demo.DAORepo.ServiceProvider_Repository;

public interface IServiceProviderStatusState {
    void handleStatusState(ServiceProvider provider,ServiceProvider_Repository repository);
    String getStateName();
}
```

Figure 36: State Interface

Concrete State: AvailableStatusState, BusyStatusState

AvailableStatusState.java

```java
package com.example.demo.Model.ServiceProvider;

import org.springframework.beans.factory.annotation.Autowired;

@Component
public class AvailableStatusState implements IServiceProviderStatusState{
    @Autowired
    ServiceProvider_Repository serviceprovider_repository;


    @Override
    public void handleStatusState(ServiceProvider provider,ServiceProvider_Repository repository) {
         if (repository != null) {
             repository.updateServicsStatus(provider); // Use the repository only if available
         }
    }

    @Override
    public String getStateName() {
        return "AVAILABLE";
    }

}
```

Figure 37: Concrete state, AvailableStatusState.

BusyStatusState.java

```java
package com.example.demo.Model.ServiceProvider;

import java.util.List;

@Component
public class BusyStatusState implements IServiceProviderStatusState {

    @Autowired
    ServiceProvider_Repository serviceprovider_repository;

    @Override
    public void handleStatusState(ServiceProvider provider,ServiceProvider_Repository repository) {
        if (repository != null) {
            repository.updateServicsStatus(provider);
        }
    }

    @Override
    public String getStateName() {
        return "BUSY";
    }

}
```

Figure 38: Concrete state, BusyStatusState.

Context: ServiceProviderStateManager.java

ServiceProviderStateManager.java

```java
@Component
public class ServiceProviderStateManager {

    private final AvailableStatusState availableState;

    private final BusyStatusState busyState;

    private final ServiceProvider_Repository serviceProviderRepository;

    public ServiceProviderStateManager(AvailableStatusState availableState, BusyStatusState busyState,
            ServiceProvider_Repository serviceProviderRepository) {
        this.serviceProviderRepository = serviceProviderRepository;
        this.busyState = busyState;
        this.availableState = availableState;
    }

    public void changeState(ServiceProvider provider, String stateName, List<Customer> customers) {

        for (Customer customer : customers) {
            provider.addObserver(customer);
        }

        switch (stateName.toUpperCase()) {
        case "COMPLETED":
            // availableState.updateServicesStatus(provider, serviceProviderRepository);
            provider.setState(availableState);
            break;
        case "ACCEPTED":
            // busyState.updateServicesStatus(provider, serviceProviderRepository);
            provider.setState(busyState);
            break;
        default:
            throw new IllegalArgumentException("Unknown state: " + stateName);
        }
        provider.handleState(serviceProviderRepository);
    }
}
```

Figure 39: State Manager, context class for state design pattern

ServiceProvider.java

```java
@Component
public class ServiceProvider extends User implements IServiceProviderSubject{

        private String providerId;
        private String currentState;
        private IServiceProviderStatusState state;
        private String status;

        private final List<Customer> observers = new ArrayList<>();


        public void setState(IServiceProviderStatusState state) {
             if (state == null) {
                    throw new IllegalArgumentException("State cannot be null");
                }

            this.state = state;
            this.currentState = state.getStateName(); // Update state name
            notifyObservers("Service Provider state changed to: " + state.getStateName());
        }


        public void handleState(ServiceProvider_Repository serviceProviderRepository) {
            state.handleStatusState(this, serviceProviderRepository);
        }

        public String getStateName() {
            return state.getStateName();
        }

```

Figure 40: ServiceProvider Class, used for setting state object

8.2.5 Observer Design Pattern

Here Observer design pattern is implemented for notifying the Customer and ServiceProvider about Booking Confirmation.

Observer: ICustomerObserver

ICustomerObserver.java

```java
package com.example.demo.Model.Customer;

public interface ICustomerObserver {  3 usages  1 implementation
     void update(String message, String email);  1 implementatio
}

```

Figure 41: Interface class of Customer Observer

Concreate Observer: Customer

Customer.java

```java
package com.example.demo.Model.Customer;
import org.springframework.stereotype.Component;

import com.example.demo.Model.User.User;

@Component  dhruvupa +1
public class Customer extends User implements ICustomerObserver{

    @Override  no usages  AshutoshLembhe2000
    public String getUserType() {
        return "Customer";
    }

    @Override  dhruvupa
    public void update(String message, String email) {
        // TODO Auto-generated method stub
        System.out.println("Sending notification to " + email + ": " + message);
    }

}
```

Figure 42: Concrete Customer class for Customer Observer

Observer: IServiceProviderObserver

IServiceProviderObserver.java

```java
package com.example.demo.Model.ServiceProvider;

public interface IServiceProviderObserver {  1 usage  1 implementation  Achyutam
    void update(String message, String email);  1 implementation  Achyutam
}
```

Figure 43: Interface class for ServiceProvider Observer

Concreate Observer: ServiceProvider

ServiceProvider.java

```java
package com.example.demo.Model.ServiceProvider;
import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.example.demo.DAORepo.ServiceProvider_Repository;
import com.example.demo.Model.Customer.Customer;
import com.example.demo.Model.Customer.ICustomerObserver;
import com.example.demo.Model.User.User;

@Component  dhruvupa +3
public class ServiceProvider extends User implements IServiceProviderSubject, IServiceProviderObserver{

        private String providerId;  2 usages
        private String currentState;  1 usage
        private IServiceProviderStatusState state;  5 usages
        private String status;  1 usage
```

ServiceProvider.java

```java
public class ServiceProvider extends User implements IServiceProviderSubject, IServiceProviderObserver{

        //Observer update method for payment notification
        @Override  Achyutam
        public void update(String message, String email) {
            // TODO Auto-generated method stub
            System.out.println("Sending notification to " + email + ": " + message);
        }
```

Figure 44 : Concrete ServiceProvider class for ServiceProvider Observer

Subject: IPaymentSubject

IPaymentSubject.java

```java
package com.example.demo.Model.Payment;

import com.example.demo.Model.Customer.Customer;
import com.example.demo.Model.ServiceProvider.ServiceProvider;

public interface IPaymentSubject {  1 usage  5 implementations  Achyutam
    void addObserver(Customer observer, ServiceProvider observer2);  1 usage  1 implementation  Achyutam
    void notifyObservers();  2 usages  1 implementation  Achyutam

}
```

Figure 45: Interface class for Payment Subject

Concreate Subject: BasePayment

PaymentService.java BasePayment.java

```java
package com.example.demo.Model.Payment;

import com.example.demo.Model.Customer.Customer;
import com.example.demo.Model.ServiceProvider.ServiceProvider;

import java.util.ArrayList;
import java.util.List;

public abstract class BasePayment implements IPaymentSubject {  7 usages  4 inheritors  Gorav +1
    private String bookingId;  2 usages
    private float amount;  3 usages
    private String status;  3 usages
    private String timestamp;  2 usages

    private final List<Customer> customerObservers = new ArrayList<>();  2 usages
    private final List<ServiceProvider> serviceProviderObservers = new ArrayList<>();  2 usages

    public BasePayment(float amount, String status) {  2 usages  Gorav
        this.amount = amount;
        this.status = status;
    }
```

Figure 46 : Concrete Payment class for Payment Subject

BasePayment.java

```java
public abstract class BasePayment implements IPaymentSubject {  7 usages  4 inheritors  Gorav +1
    public String getTimestamp() {  1 usage  Gorav
        return timestamp;
    }

    public void setTimestamp(String timestamp) {  1 usage  Gorav
        this.timestamp = timestamp;
    }

    @Override  1 usage  Achyutam
    public void addObserver(Customer observer, ServiceProvider observer2) {
        customerObservers.add(observer);
        serviceProviderObservers.add(observer2);
    }

    @Override  2 usages  Achyutam
    public void notifyObservers() {
        for(Customer custObs : customerObservers){
            custObs.update( message: custObs.getName()+" Customer Notified",custObs.getEmail());
        }
        for(ServiceProvider serObs : serviceProviderObservers){
            serObs.update( message: serObs.getName()+"  Service Provider Notified ",serObs.getEmail());
        }
    }
```

Figure 46.1 : Concrete Payment class for Payment Subject

Usage Class: PaymentService and Payment

In ProcessFinalPayment method we are adding customer and Service Provider related to that Payment as a Observer

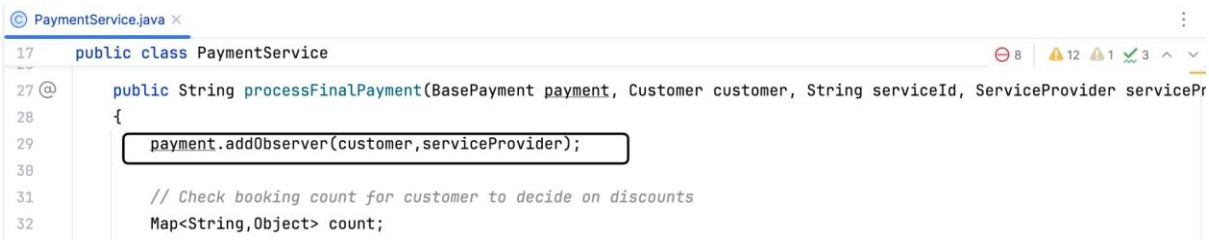


Figure 47: Adding Observer to the List

Once the payment.processPayment is called it will notify both customer and ServiceProvider about the booking confirmation

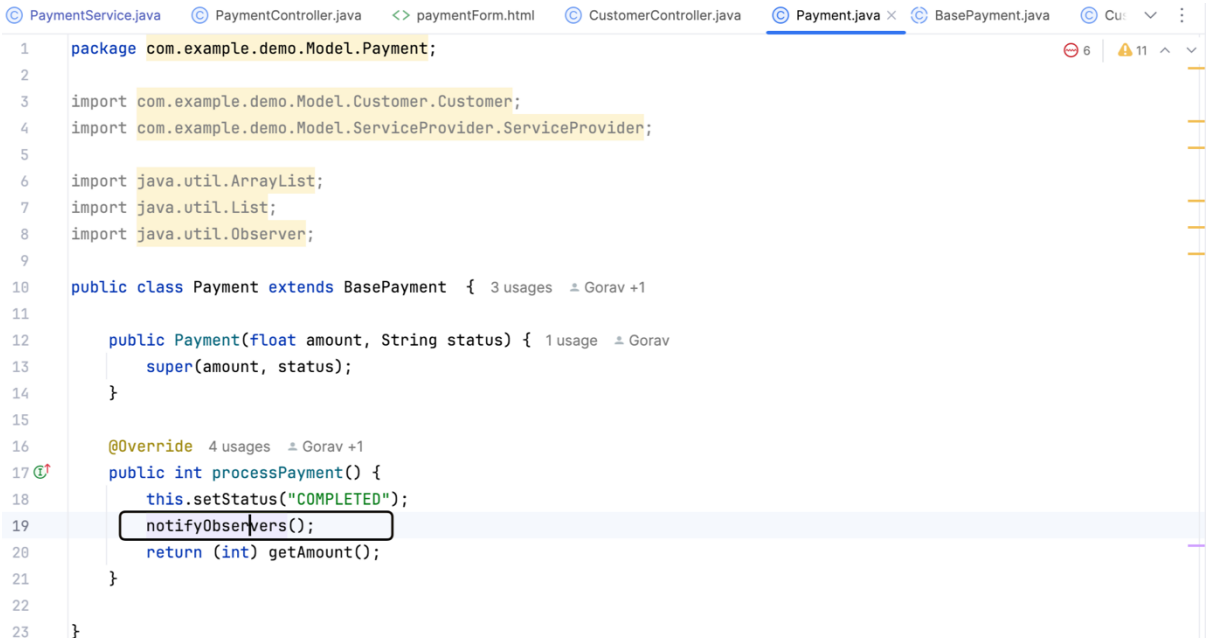


Figure 48: Notifying Observer

8.2.6 Decorator Design Pattern

We have used the decorator design pattern to dynamically add behaviour and responsibilities to the objects without modifying their code. Each payment process, including applying discounts and loyalty points, is implemented as a separate decorator class (DiscountDecorator and LoyaltPointsDecorator), ensuring a clean separation of concern. It also allows the payment operations to be easily extended or modified without altering the core logic.

Base Class: BasePayment.java

Abstract base class to define the payment processing contract and observer notification system.

```java
package com.example.demo.Model.Payment;

import com.example.demo.Model.Customer.Customer;

public abstract class BasePayment implements IPaymentSubject {
    private String bookingId;
    private float amount;
    private String status;
    private String timestamp;

    private final List<Customer> customerObservers = new ArrayList<>();
    private final List<ServiceProvider> serviceProviderObservers = new ArrayList<>();

    public BasePayment(float amount, String status) {
        this.amount = amount;
        this.status = status;
    }

    // Abstract method for processing payment
    public abstract float processPayment();

    // Common methods for all payments
    public String getBookingId() {
        return bookingId;
    }

    public void setBookingId(String bookingId) {
        this.bookingId = bookingId;
    }

    public float getAmount() {
        return amount;
    }

    public void setAmount(float amount) {
        this.amount = amount;
    }

    public String getStatus() {
        return status;
    }
```

```
46
47    public void setStatus(String status) {
48        this.status = status;
49    }
50
51    public String getTimestamp() {
52        return timestamp;
53    }
54
55    public void setTimestamp(String timestamp) {
56        this.timestamp = timestamp;
57    }
58
59
60    @Override
61    public void addObserver(Customer observer, ServiceProvider observer2) {
62        customerObservers.add(observer);
63        serviceProviderObservers.add(observer2);
64    }
65
66    @Override
67    public void notifyObservers() {
68        for(Customer custObs : customerObservers){
69            custObs.update(custObs.getName()+" Customer Notified",custObs.getEmail());
70        }
71        for(ServiceProvider serObs : serviceProviderObservers){
72            serObs.update(serObs.getName()+"  Service Provider Notified ",serObs.getEmail());
73        }
74    }
75
76 }
```

Figure 49: Base class (Superclass) for Payment

Concrete Class: Payment.java

Implements the core payment processing logic and notifies observers upon completion.

```
1  package com.example.demo.Model.Payment;
2
3  import com.example.demo.Model.Customer.Customer;
9
10 public class Payment extends BasePayment  {
11
12     public Payment(float amount, String status) {
13         super(amount, status);
14     }
15
16     @Override
17     public float processPayment() {
18         this.setStatus("COMPLETED");
19         notifyObservers();
20         return (int) getAmount();
21     }
22
23 }
```

Figure 50: Concrete Payment Class (Child Class)

Abstract Decorator: PaymentDecorator.java

Abstract decorator class for adding the dynamic behaviours, for example, discounts, to payment operations.

```java
package com.example.demo.Model.Payment;

public abstract class PaymentDecorator extends BasePayment {
    protected BasePayment decoratedPayment;

    public PaymentDecorator(BasePayment payment) {
        super(payment.getAmount(), payment.getStatus());
        this.decoratedPayment = payment;
    }

    @Override
    public float processPayment() {
        return decoratedPayment.processPayment();
    }
}
```

Figure 51: Payment Decorator class (Super class for other concrete decorators) and child class of Payment.

Decorator 1: DiscountDecorator.java

It applies a 5% discount to the payment amount before processing the payment.

```java
package com.example.demo.Model.Payment;

public class DiscountDecorator extends PaymentDecorator {

    public DiscountDecorator(BasePayment payment) {
        super(payment);
    }

    @Override
    public float processPayment() {
        float discountedAmount = decoratedPayment.getAmount() * 0.95f; // Apply 5% discount
        decoratedPayment.setAmount(discountedAmount);
        System.out.println("5% Discount Applied.");
        return decoratedPayment.processPayment();
    }
}
```

Figure 52: Concrete Decorator 1 (Child class of Payment decorator)

Decorator 2: LoyaltyDecorator.java

It applies a 10% loyalty discount to the payment amount before the processing.

```java
package com.example.demo.Model.Payment;

public class LoyaltyPointsDecorator extends PaymentDecorator {

    public LoyaltyPointsDecorator(BasePayment payment) {
        super(payment);
    }

    @Override
    public float processPayment() {
        float loyaltyDiscountedAmount = decoratedPayment.getAmount() * 0.90f; // Apply 10% discount
        decoratedPayment.setAmount(loyaltyDiscountedAmount);
        System.out.println("10% Loyalty Discount Applied.");
        return decoratedPayment.processPayment();
    }
}
```

Figure 53: Concrete Decorator 2 (Child class of Payment decorator)

---

9. Automated Test Cases:

9.1 Customer Controller:

We are trying to mock the controllers which have created. We have written test cases for each type of Request Mapping controller function.

```java
package com.example.demo.Controller;

import static org.junit.jupiter.api.Assertions.*;

class CustomerControllerTest {

    @InjectMocks
    private CustomerController customerController;

    @Mock
    private CustomerService customerService;

    @Mock
    private Model model;

    @Mock
    private IUserFactory userFactory;

    @Mock
    private SearchServiceService searchServiceService;

    @Mock
    private RedirectAttributes redirectAttributes;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
        customerController.setGlobalCustomername("testCustomer");
    }

    @Test
    void testCheckUser() {
        // Arrange
        when(userFactory.createUser("Customer")).thenReturn(new Customer());

        // Act
        String viewName = customerController.checkUser(model);
```

Figure 54: mocking each function in our controller

```java
@Test
void testCreateCustomer() throws SQLException {
    // Arrange
    Customer customer = new Customer();
    when(customerService.addCustomer(customer)).thenReturn(1);

    // Act
    String response = customerController.createCustomer(customer);

    // Assert
    assertEquals("User Created Successfully", response);
}

@Test
void testAuthenticateCustomer_Success() {
    // Arrange
    String name = "testName";
    String password = "testPassword";
    when(customerService.checkCustomerLogin(name, password)).thenReturn(1);

    // Act
    String viewName = customerController.authenticateCustomer(name, password, model);

    // Assert
    assertEquals("CustomerWelcomeScreen", viewName);
}

@Test
void testAuthenticateCustomer_Failure() {
    // Arrange
    String name = "testName";
    String password = "testPassword";
    when(customerService.checkCustomerLogin(name, password)).thenReturn(0);
    when(userFactory.createUser("Customer")).thenReturn(new Customer());

    // Act
    String viewName = customerController.authenticateCustomer(name, password, model);
```

Figure 55: Test cases for each function in our controller.

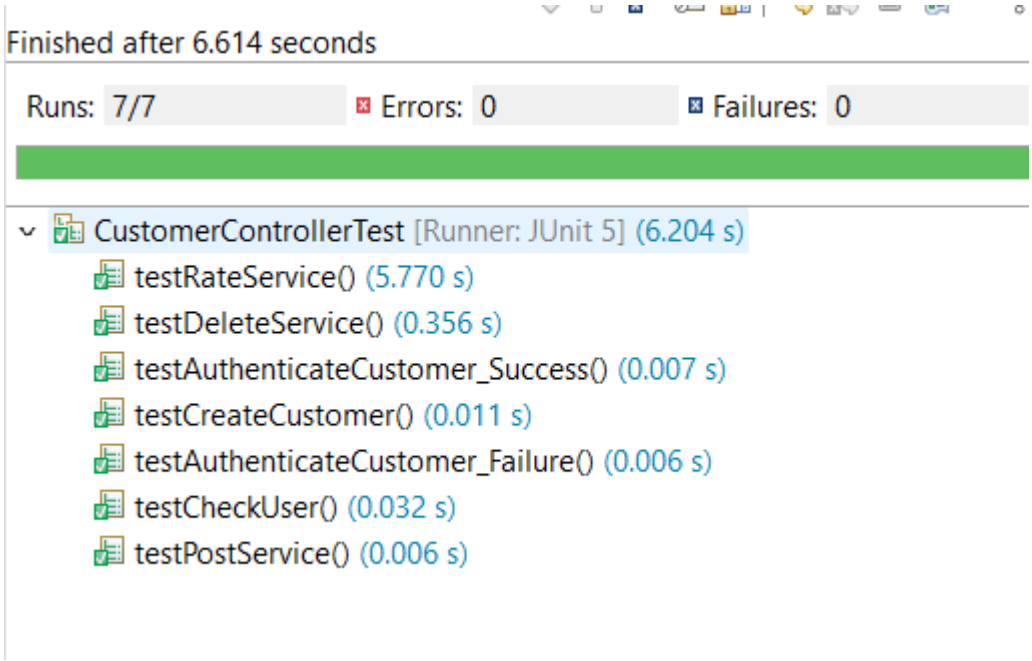


Figure 56: All test cases have passed.

All test cases have passed.

9.2 Service Provider Controller Test file

```
32 class ServiceProvider_ControllerTest {
33
34     @InjectMocks
35     private ServiceProviderController serviceProviderController;
36     @Mock
37     private ServiceProviderService serviceproviderservice;
38     @Mock
39     private IUserFactory userFactory;
40     @Mock
41     private GlobalContext globalcontext;
42     @Mock
43     private WalletService walletService;
44     @Mock
45     private ServiceProviderStateManager serviceProviderStateManager;
46     @Mock
47     private CustomerService customerService;
48
49     @Mock
50     private Model model;
51
52     @Mock
53     private RedirectAttributes redirectAttributes;
54
55     @BeforeEach
56     void setUp() {
57         MockitoAnnotations.openMocks(this);
58     }
59
60     @Test
61     void testLoginForm() {
62         String viewName = serviceProviderController.loginForm(model);
63         assertEquals("ServiceProvider_login", viewName);
64     }
```

Figure 57.1 : Junit test case for service provider controller

```
@Test
void testAddServiceItem() {
    SearchService searchService =new SearchService();
    when(serviceproviderservice.verifyServiceAddition(searchService)).thenReturn(-1);
    String result = serviceProviderController.addServiceItem(searchService);

    when(serviceproviderservice.verifyServiceAddition(searchService)).thenReturn(1);
    String result2 = serviceProviderController.addServiceItem(searchService);

    assertEquals("You can only add one service per ServiceProvider.", result);
    assertEquals("Service Added Sucesfully!", result2);
}

@Test
void testViewBooking() {
    ServiceProviderBookingDTO serviceProviderBookingDTO = new ServiceProviderBookingDTO();
    serviceProviderBookingDTO.setCustomerName("Test");
     List<ServiceProviderBookingDTO> bookings = new ArrayList<>();
     bookings.add(serviceProviderBookingDTO);

     when(serviceproviderservice.getBookedServices()).thenReturn(bookings);
     String view =  serviceProviderController.viewBooking(model);
     assertEquals("ServiceProviderBookedServices", view);

}
```

Figure 57.2: Junit test case for service provider controller

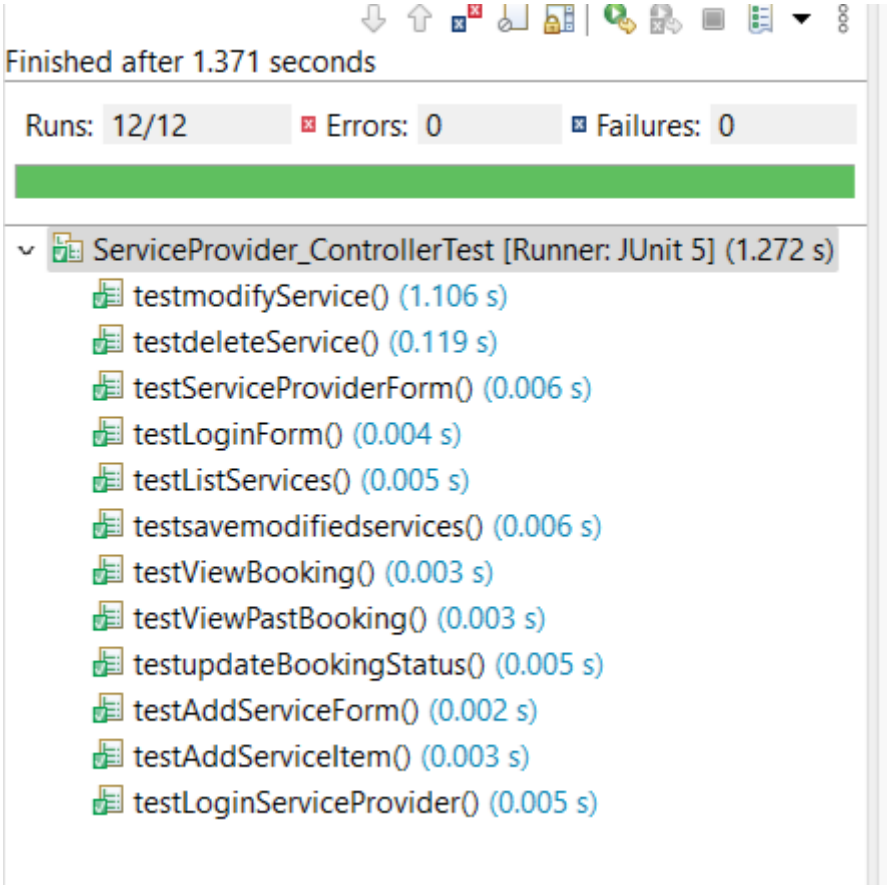


Figure 58: Junit test case result for service provider controller

All Test Cases Passed

10. Version Control

Figure 59.1: GitHub Commits and contribution

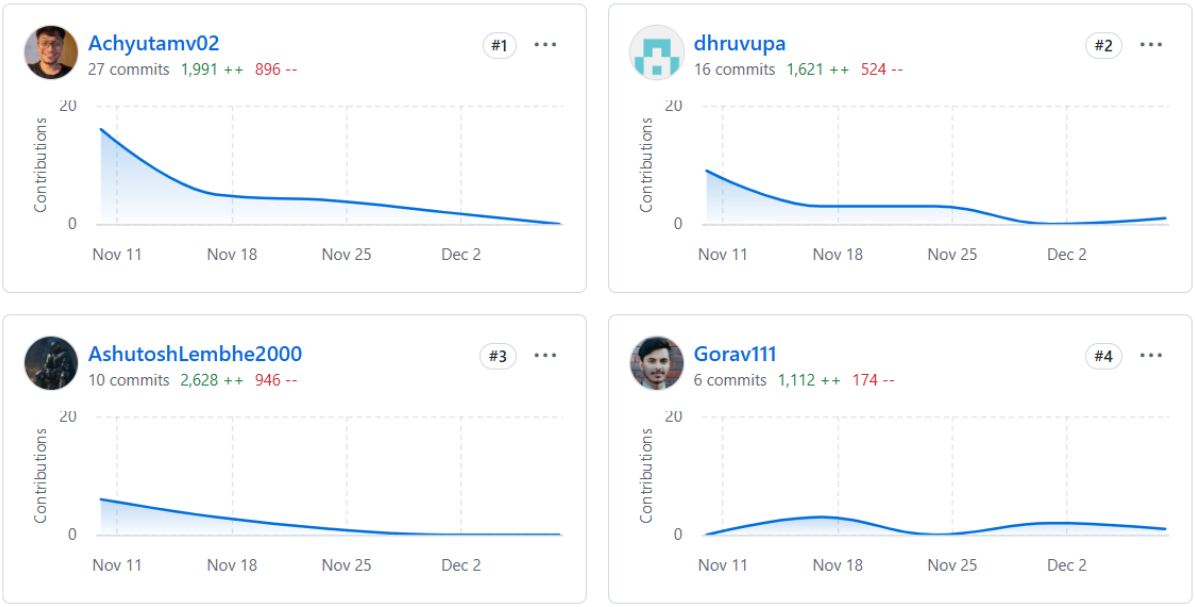


Figure 59.2: GitHub Commits and contribution for each teammate

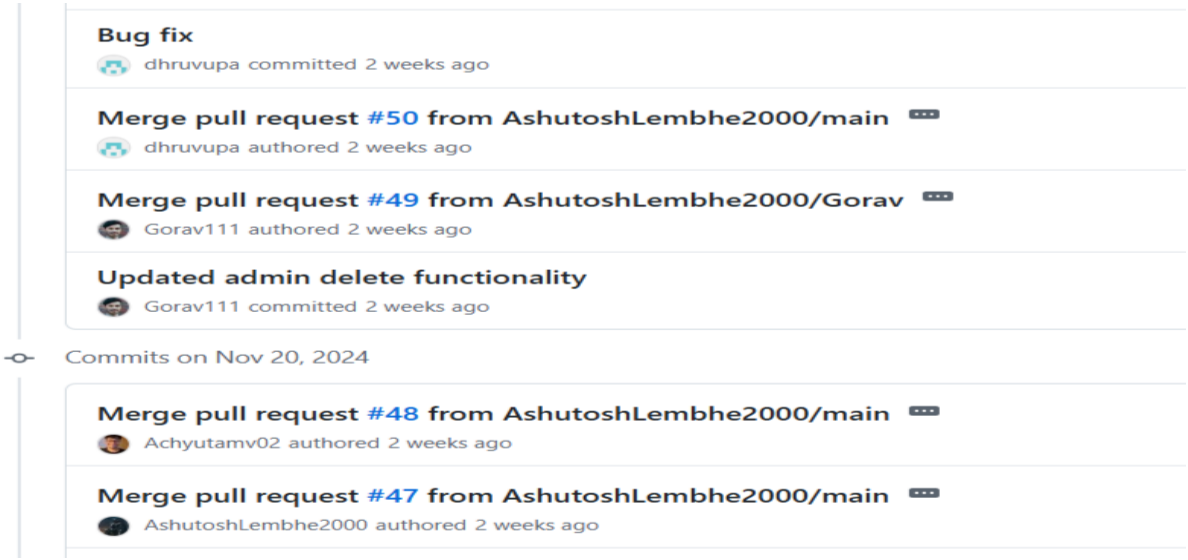


Figure 59.3: GitHub Commits history

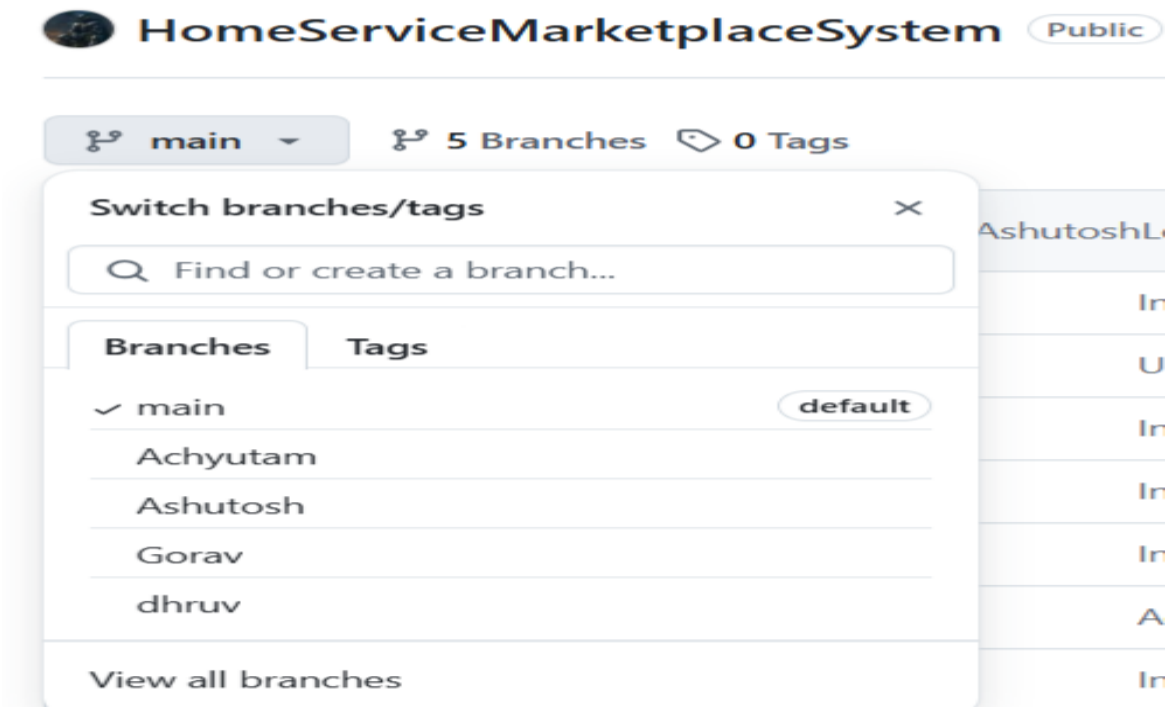


Figure 59.4: GitHub team member's branch

11. API Implemented:

11.1 Customer Controller:

Table 10: Customer Controller API's

Method	Path	Description
GET	http://localhost:8080/Customer/customerLogin	Show customer login screen
GET	http://localhost:8080/Customer/customerRegistrationForm	Show customer registration Screen
POST	http://localhost:8080/Customer/customerRegistrationSuccess	We register the customer with two outputs
POST	http://localhost:8080/Customer/customerLoginSuccess?username=Ashutosh Lembhe&password=test	Brings us to customer welcome screen.
GET	http://localhost:8080/Customer/allservices	Returns the search services screen
GET	http://localhost:8080/Customer/filterservices?City=Limerick	Returns the same screen with filtered services view
GET	http://localhost:8080/Customer/rateService	Returns a screen with rating service
POST	http://localhost:8080/Customer/postRating	Submits rating for the service provider
GET	http://localhost:8080/Customer/book	Shows a booking Screen
GET	http://localhost:8080/Customer/CustomerCurrentBookedServices	Returns a screen with customers current booking with an option to cancel them.

POST	http://localhost:8080/Customer/CancelBooking/{bookingId}/{bookingStatus}	Returns the same screen with selected service as cancelled.

11.2 Service Provider Controller

Table 11: Service Provider Controller API's

Method	Path	Description
GET	http://localhost:8080/ServiceProvider/ServiceProviderRegistrationForm	Shows service provider registration screen.
POST	http://localhost:8080/ServiceProvider/ServiceProviderRegistrationSuccess	Registers the user. If user already exist navigates to login screen.
GET	http://localhost:8080/ServiceProvider/ServiceProviderLoginForm	Shows service provider Login Screen.
POST	http://localhost:8080/ServiceProvider/loginsucess	Login service provider and shows welcome screen.
GET	http://localhost:8080/ServiceProvider/ServiceProviderWelcomeScreen	Shows service provider welcome screen.
GET	http://localhost:8080/ServiceProvider/addServiceForm?	Shows screen for adding service provider services.
POST	http://localhost:8080/ServiceProvider/addServiceItem	Adds service provider services.
GET	http://localhost:8080/ServiceProvider/ViewBooking	Shows screen for service provider bookings.
GET	http://localhost:8080/ServiceProvider/PastBookings?	Shows screen for service provider past booking.
GET	http://localhost:8080/ServiceProvider/ListServices?	Shows all the services provided by service provider, with an option to modify and delete services.
POST	http://localhost:8080/ServiceProvider/ModifyService/Dhruv%20Upadhyay/1	Opens a form to modify service provider services.
POST	http://localhost:8080/ServiceProvider/DeleteService/Dhruv%20Upadhyay/1	Deletes service provider service.
POST	http://localhost:8080/ServiceProvider/SaveModifiedService	Saves modified services of service provider.
POST	http://localhost:8080/ServiceProvider/UpdateBookingStatus	Updates booking and service provider status when service provider accepts or rejects booking.

11.3 Payment Controller

Table 12: Payment Controller API's

Method	Path	Description
GET	http://localhost:8080/payment/VerifyOTP?bookingId=BOOK1732808007634	Show OTP Verification page with the bookingID
POST	http://localhost:8080/payment/VerifyOTPSuccess	If the Entered OTP is Correct ServiceProvider should be navigated to WelcomeScreen
GET	http://localhost:8080/payment/makePayment?customerCity=Limerick&servic	Shows the payment form page to pay for the service

	eCity=Limerick&serviceId=4&serviceStatus=Free&servicePrice=10.0&serviceSkill=Plumber&serviceRating=5.0&serviceProviderName=Dhruv&serviceCategory=Plumber	
POST	http://localhost:8080/payment/processPayment	Shows the payment status after payment is successfully executed
GET	http://localhost:8080/payment/getWalletBalance	Gets the balance of the wallet with respect to the userId and user_type

11.4 Admin Controller

Table 13: Admin Controller API's

Method	Path	Description
GET	http://localhost:8080/Admin/adminRegistrationForm	Shows the registration form for the admin
POST	http://localhost:8080/Admin/adminRegistrationSuccess	Registers the admin. If successfully registered, displays the success message
GET	http://localhost:8080/Admin/manage-accounts	Shows the screen to manage (delete) all the customers and the service providers.
GET	http://localhost:8080/Admin/manage-accounts/customer/2?	Deletes a particular customer with customerId = 2
GET	http://localhost:8080/Admin/manage-accounts/service-provider/3?	Deletes a particular service provider with serviceId = 3

11.5 Wallet Controller

Table 14: Wallet Controller API's

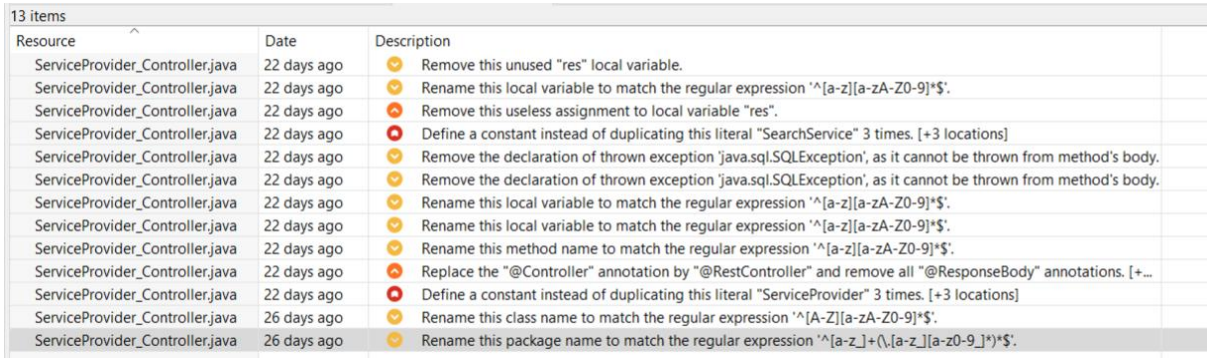
Method	Path	Description
GET	http://localhost:8080/wallet/3/CUSTOMER	Gets the wallet object with the given userID and user_type
POST	http://localhost:8080/wallet/addFunds	Adds the funds to the customer wallet

12. Added Values

12.1 Software Metrics:

We are using Sonar Lint to remove bad code smells and have done refactoring on our code.

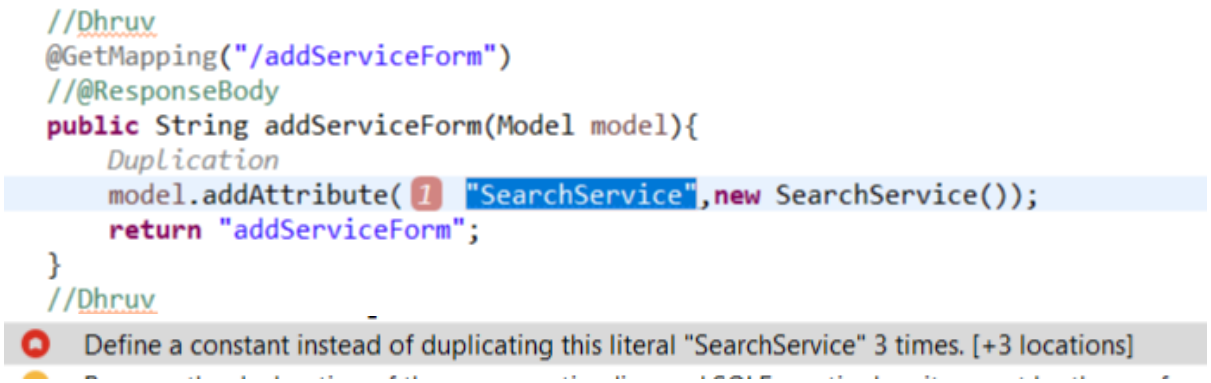
ServiceProvider Controller before Refactoring:



13 items

Resource	Date	Description
ServiceProvider_Controller.java	22 days ago	Remove this unused "res" local variable.
ServiceProvider_Controller.java	22 days ago	Rename this local variable to match the regular expression '^[a-z][a-zA-Z0-9]*$'.
ServiceProvider_Controller.java	22 days ago	Remove this useless assignment to local variable "res".
ServiceProvider_Controller.java	22 days ago	Define a constant instead of duplicating this literal "SearchService" 3 times. [+3 locations]
ServiceProvider_Controller.java	22 days ago	Remove the declaration of thrown exception 'java.sql.SQLException', as it cannot be thrown from method's body.
ServiceProvider_Controller.java	22 days ago	Remove the declaration of thrown exception 'java.sql.SQLException', as it cannot be thrown from method's body.
ServiceProvider_Controller.java	22 days ago	Rename this local variable to match the regular expression '^[a-z][a-zA-Z0-9]*$'.
ServiceProvider_Controller.java	22 days ago	Rename this local variable to match the regular expression '^[a-z][a-zA-Z0-9]*$'.
ServiceProvider_Controller.java	22 days ago	Rename this method name to match the regular expression '^[a-z][a-zA-Z0-9]*$'.
ServiceProvider_Controller.java	22 days ago	Replace the "@Controller" annotation by "@RestController" and remove all "@ResponseBody" annotations. [+...
ServiceProvider_Controller.java	22 days ago	Define a constant instead of duplicating this literal "ServiceProvider" 3 times. [+3 locations]
ServiceProvider_Controller.java	26 days ago	Rename this class name to match the regular expression '^[A-Z][a-zA-Z0-9]*$'.
ServiceProvider_Controller.java	26 days ago	Rename this package name to match the regular expression '^[a-z_]+(\.[a-z_][a-z0-9_]*)*$'.

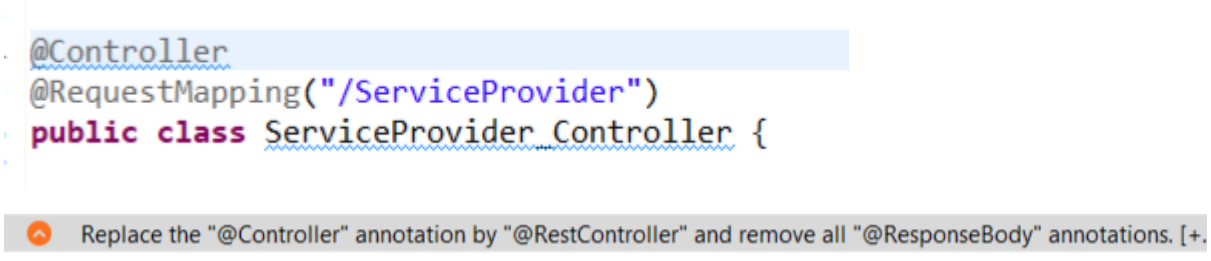
Figure 60: Sonar Lint bad code smell warnings



```
//Dhruv
@GetMapping("/addServiceForm")
//@ResponseBody
public String addServiceForm(Model model){
    Duplication
    model.addAttribute( 1  "SearchService",new SearchService());
    return "addServiceForm";
}
//Dhruv
```

Define a constant instead of duplicating this literal "SearchService" 3 times. [+3 locations]

Figure 61.1: Sonar Lint bad code smell warning and code reference



```
@Controller
@RequestMapping("/ServiceProvider")
public class ServiceProvider_Controller {
```

Replace the "@Controller" annotation by "@RestController" and remove all "@ResponseBody" annotations. [+.

Figure 61.2: Sonar Lint bad code smell warning and code reference

In the above screenshot, there are some samples of Sonar Lint refactoring suggestions, like to use @RestController instead of @Controller and remove @ResponseBody, define constant literals instead of duplicating it.

ServiceProvider Controller after Refactoring:

```
//Dhruv
@GetMapping("/addServiceForm")
public String addServiceForm(Model model){
    model.addAttribute(SEARCH_SERVICE,new SearchService());
    return "addServiceForm";
}
```

Figure 62.1: Sonar Lint bad code smell refactored code

```
22 @RestController
23 @RequestMapping("/ServiceProvider")
24 public class ServiceProviderController {
25
26     public static final String SEARCH_SERVICE = "SearchService";
27     public static final String SERVICE_PROVIDER = "ServiceProvider";
28
```

Figure 62.2: Sonar Lint bad code smell refactored code

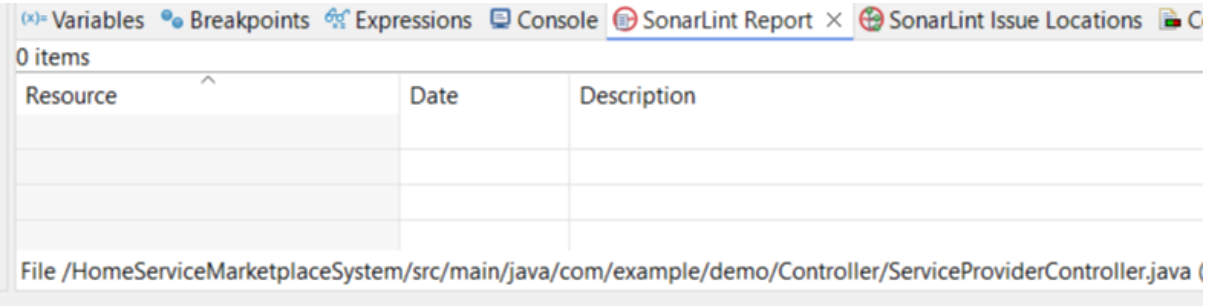


Figure 63: Sonar Lint, no warnings after refactoring

In above screenshot we have implemented all the suggestions recommended by Sonar Lint, few screenshot attached above for reference.

Customer Controller before refactoring:

Using the sonar lint we have identified 12 bad code smells including changing the annotations to the correct one and removing the wrong ones, to changing the correct data type for variable.

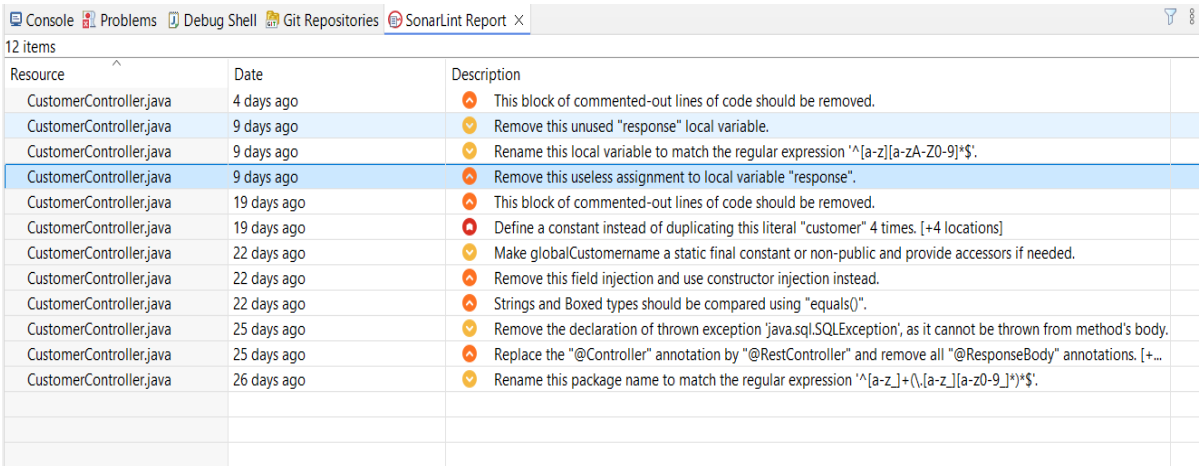


Figure 64: Bad code smells for Customer Controller.

```
@Controller
@RequestMapping("/Customer")
public class CustomerController {

    public String globalCustomername;

    public String getGlobalCustomername() {
        return globalCustomername;
    }

    public void setGlobalCustomername(String globalCustomername) {
        this.globalCustomername = globalCustomername;
    }

    private final CustomerService customerService;
    private IUserFactory userFactory;
    @Autowired
    private SearchServiceService searchServiceService;

    @Autowired
    public CustomerController(CustomerService customerService,IUserFactory userFactory) {
        this.customerService = customerService;
        this.userFactory=userFactory;
    }

    /*---------------Customer Registration----------------------------*/
    @GetMapping("/customerRegistrationForm")
    public String checkUser(Model model){
        //User customer = userFactory.createUser("Customer");
        userFactory=new ConcreteUserFactory();
```

Figure 65: Bad code in customer controller with wrong annotations and other bad syntax

```
@PostMapping("/CancelBooking/{bookingId}/{bookingStatus}")
@ResponseBody
public RedirectView deleteService(@PathVariable("bookingId") String bookingId,@PathVariable("bookingStatus") S
    int res = customerService.cancelSelectedBooking(bookingId,bookingStatus);
    if(res!=0) {
        redirectAttributes.addFlashAttribute("message", "Booking Status Rejected Successfully!!");
        return new RedirectView("/Customer/CustomerCurrentBookedServices");
    }
    redirectAttributes.addFlashAttribute("message", "Booking status cannot be rejected");
    return new RedirectView("/Customer/CustomerCurrentBookedServices");
}
```

Figure 66: Post Mapping bad code smells

```
@GetMapping("/filterservice")
public String listFilteredServices(
        @RequestParam(required = false) String skill,
        @RequestParam(required = false) String city,
        @RequestParam(required = false) String status,
        Model model) {

    // Fetch the filtered services for the customer
    List<SearchService> services = customerService.searchService(skill, city, status);
    Customer customer=customerService.getCustomerByName(this.getGlobalCustomername());
    model.addAttribute("customer",customer);
    model.addAttribute("services", services);
    return "filter-services"; // Refers to the Thymeleaf view
}
```

Figure 67: Get Mapping Bad code smells

Customer Controller after refactoring:

We can see that all the bad code smells have gone after implementation of the required changes. We have changed the @Controller to @RestController and removed @Autowired and added constructor. Removed @ResponseBody. Added @RequestBody, changed to public static data type to keep the data constant. And created a variable to replace strings on our model.addAttributes methods.

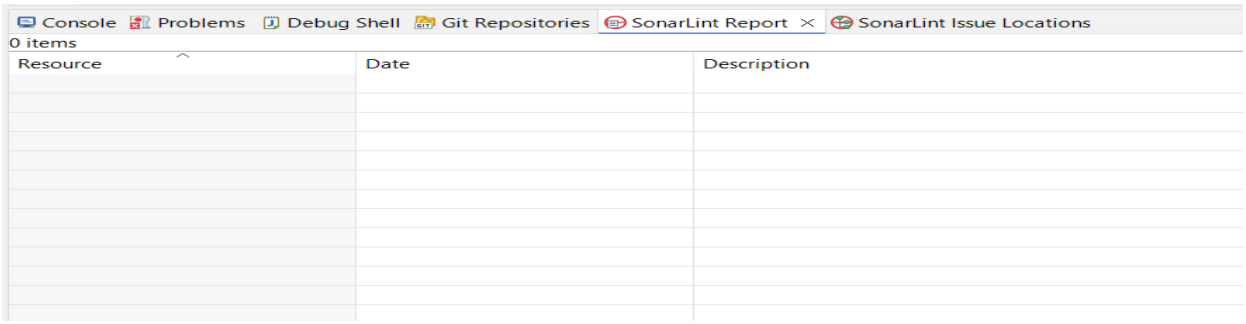


Figure 68: Bad code smells fixed for customer controller

```
18 @RestController
19 @RequestMapping("/Customer")
20 public class CustomerController {
21
22     public static String globalCustomername;
23     public static final String customerString="customer";
24     public String getGlobalCustomername() {
25         return globalCustomername;
26     }
27
28     public void setGlobalCustomername(String globalCustomername) {
29         this.globalCustomername = globalCustomername;
30     }
31
32     private final CustomerService customerService;
33     private IUserFactory userFactory;
34
```

Figure 69: Corrected Syntax for Controller and RequestMapping

Removed Autowired annotation

```
private SearchServiceService searchServiceService;

public CustomerController(CustomerService customerService,IUserFactory userFactory,SearchServiceService searchServiceService) {
    this.customerService = customerService;
    this.userFactory=userFactory;
    this.searchServiceService =searchServiceService;
}
```

```
@PostMapping("/customerRegistrationSuccess")
public String createCustomer(@RequestBody Customer customer) {
     int response= customerService.addCustomer(customer);
     if(response==0) {
          return "User Already Exists go to login";
     }
     else
     {
          return "User Created Successfully";
     }
}
/*---------------Customer Registration----------------------------*/
```

Figure 70: Removed Autowired and added @RequestBody for customer.

```
@GetMapping("/customerRegistrationForm")
public String checkUser(Model model){
    userFactory=new ConcreteUserFactory();

    model.addAttribute(customerString,userFactory.createUser("Customer"));
    return "CustomerRegistration";
}
```

Figure 71: Change customer String to CustomerString variable

12.2 UI using REST Architectural Pattern:

- **Thymeleaf Page**:
  This is our user interface (UI) layer. Thymeleaf is a server-side template engine for Java-based web applications. It enables us to create dynamic web pages that interact with our backend.
- **RESTful APIs**:
  The image shows a RESTful controller in Spring Boot, annotated with `@RestController` and `@RequestMapping("/Customer")`. This controller handles HTTP requests and responses in a RESTful manner.

Combining Thymeleaf and REST:

- **UI Layer**: Your Thymeleaf page serves as the front-end where users interact with your application. It can make HTTP requests (GET, POST, etc.) to the backend to fetch or send data.
- **REST Pattern**: The RESTful APIs exposed by your Spring Boot controller handle the HTTP requests from the UI. They perform operations like retrieving data, creating new records, updating existing ones, and so on.

Example Workflow:

1. **User Interaction**: A user interacts with the Thymeleaf page, for instance, by filling out a form and submitting it.
2. **HTTP Request**: The form submission triggers an HTTP request to one of your REST endpoints in the `CustomerController`.
3. **Backend Processing**: The `CustomerController` processes the request, performs the necessary business logic, and interacts with the database through services like `CustomerService` and `SearchServiceService`.
4. **Response**: Once the processing is complete, the REST controller sends back an HTTP response, which Thymeleaf can use to update the UI accordingly.

UI:

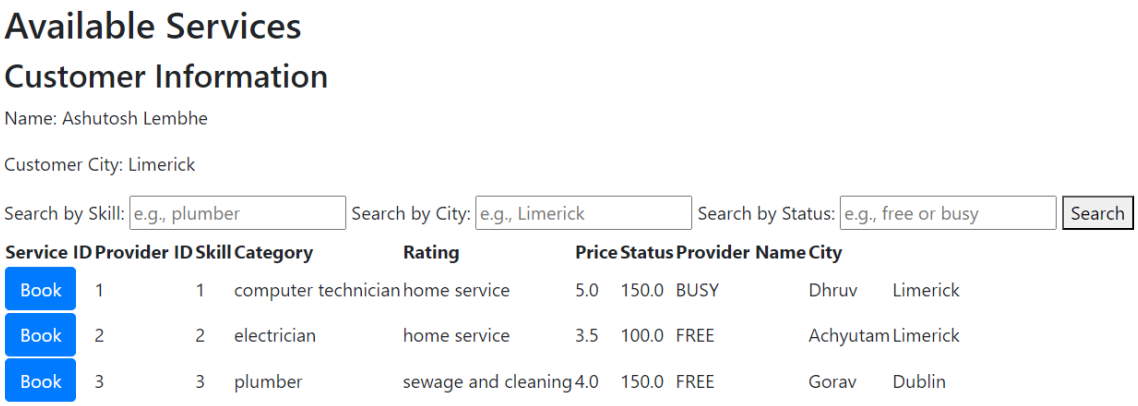


Figure 71: Search service UI on local host

REST Architectural Patterns:

```
19
20  @Controller
21  @RequestMapping("/Customer")
22  public class CustomerController {
23
24      public static String globalCustomername;
25      public static final String customerString="customer";
26      public String getGlobalCustomername() {
27          return globalCustomername;
28      }
29
30      public void setGlobalCustomername(String globalCustomername) {
31          this.globalCustomername = globalCustomername;
32      }
33
34      private final CustomerService customerService;
35      private IUserFactory userFactory;
36
```

```
/*                Customer Registration                    ------------------------*/
@GetMapping("/customerRegistrationForm")
public String checkUser(Model model){
    User customer = userFactory.createUser("Customer");
    model.addAttribute("customer",customer);
    return "CustomerRegistration";
}


@PostMapping("/customerRegistrationSuccess")
@ResponseBody
public String createCustomer(Customer customer) throws SQLException {
    int response= customerService.addCustomer(customer);
    if(response==0) {
        return "User Already Exists go to login";
    }
    else
    {
        return "User Created Successfully";
    }
}
/*---------------Customer Registration---------------------------*/
```

Figure 72: Implementation Rest Architectural patterns using GetMapping and PostMapping

13. Recovered architecture and design blueprints.

13.1 Design-time package diagram.

We have a new service package Wallet. And few new classes such as row mapper classes, which will be used by the repository to map each row to the correct attribute in our model. This data will be further used for displays in view.

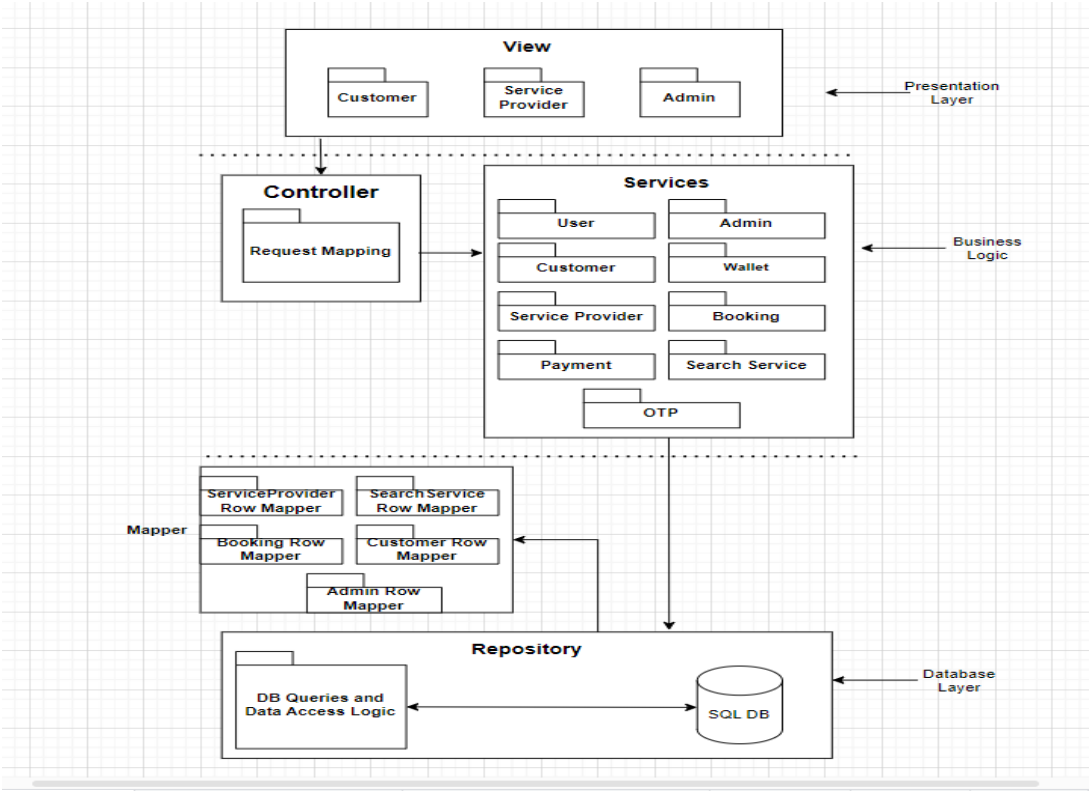
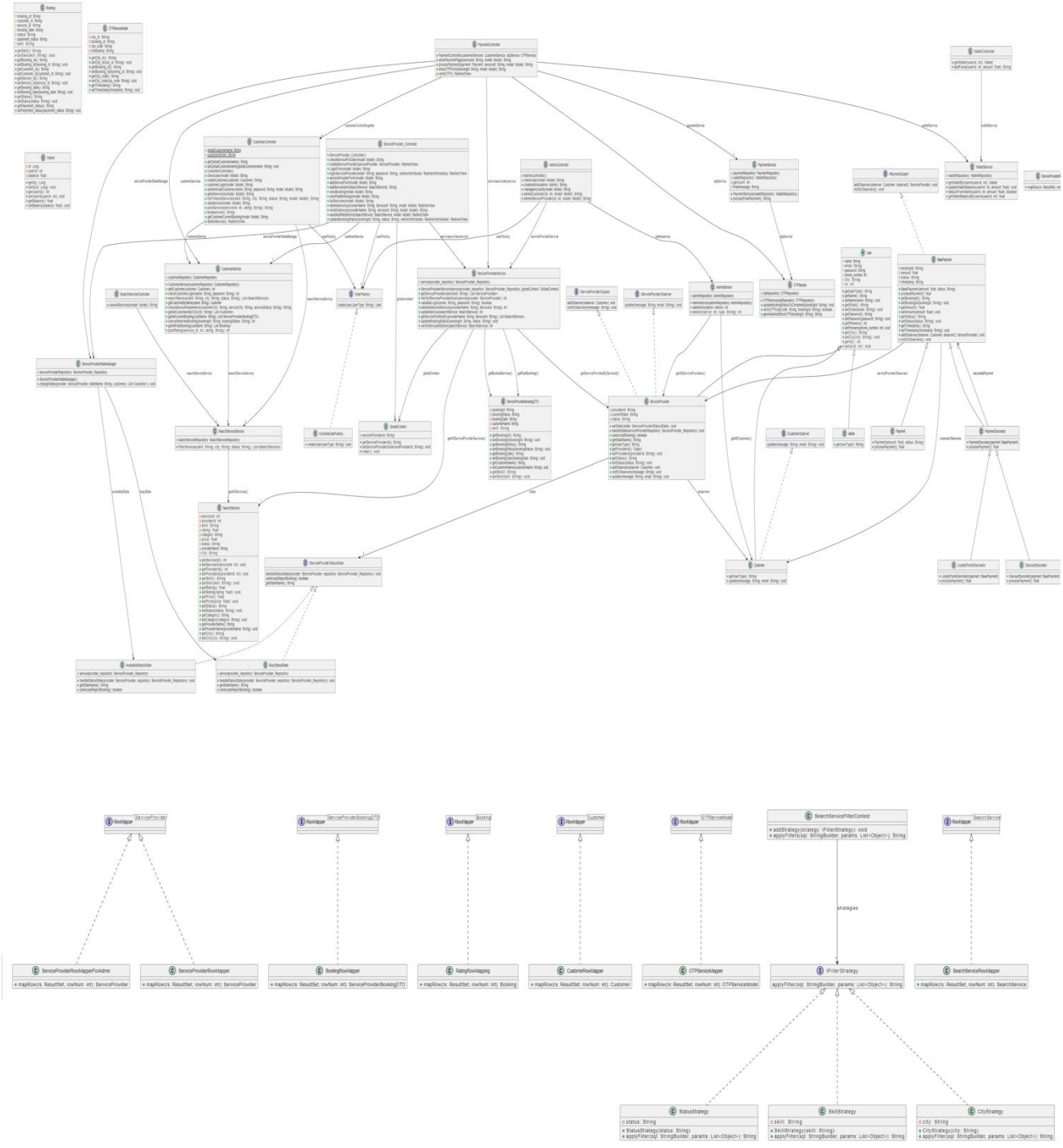


Figure 73: Revised Architectural MVC

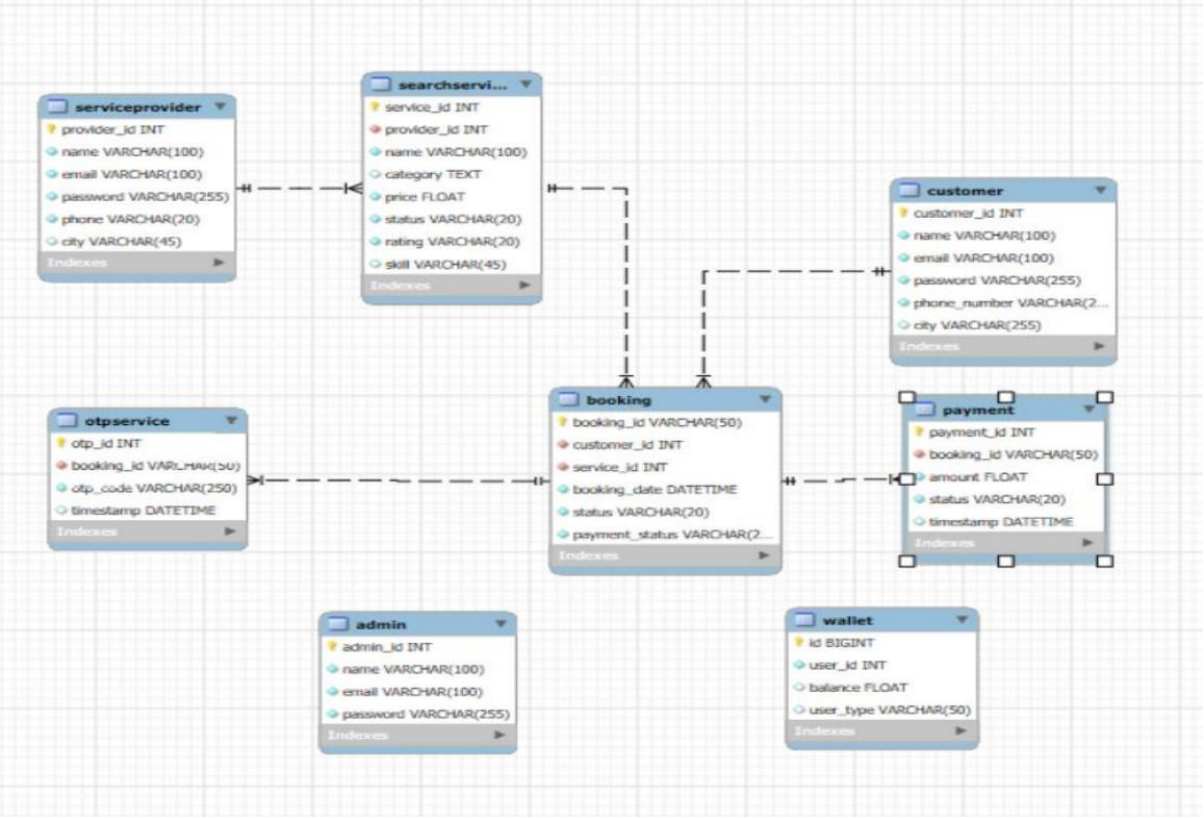
13.2 Design time class diagram.



We are splitting our class diagram into different design patterns that we have implemented. This way it is easier to understand what major changes and design patterns were implemented in our project. Design Time Combined Class diagram drive link:

https://drive.google.com/file/d/1oGeenkRY0oPGME1ULcyawBnQ0y_Vflrl/view?usp=sharing

13.3 Design Time Entity Relationship Diagram



We are splitting our class diagram into different design patterns that we have implemented. This way it is easier to understand what major changes and design patterns were implemented in our project.

1. **Factory Design Pattern**:
   Definition: The factory method defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
   Implementation The application uses ConcreteUserFactory to create different types of users. It is used to replace class constructors, abstracting the process of object generation so that the type of object instantiated can be determined at run-time. Object creation is being done in the controller class. This design pattern follows the **Open Close Principle as it is open to extend to new classes but close to modification to old code. Also follows Find what varies and encapsulate it.**

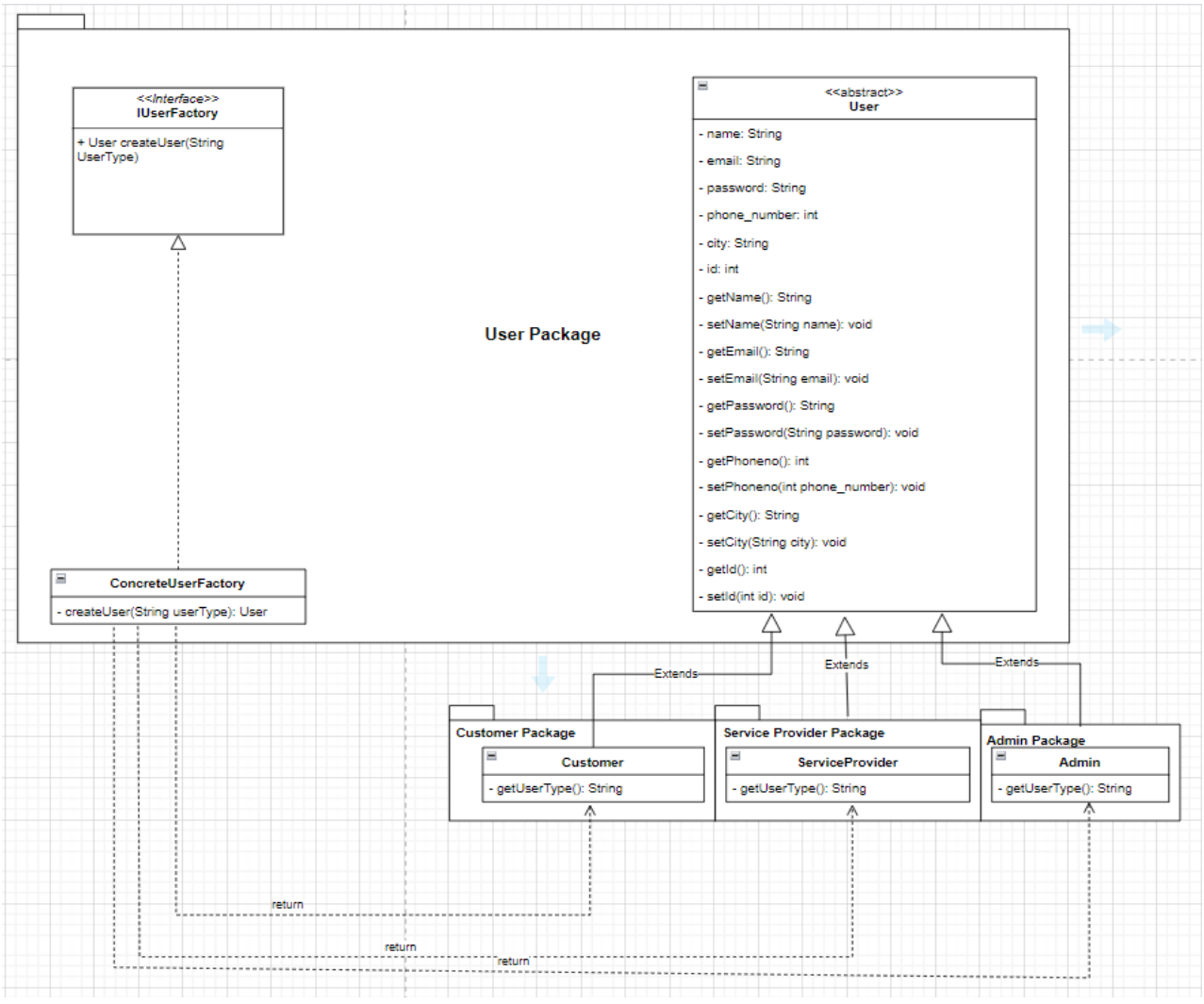


Figure 74: Factory Design Pattern Design Time Class Diagram

2. **Strategy Design Pattern:**

Definition: The intent of the Strategy Pattern is to define a family of algorithms, encapsulate each algorithm, and make them interchangeable. The Strategy Pattern lets the algorithm vary independently from clients that use it. In addition, the pattern, defines a group of classes that represent a set of possible behaviours. These behaviours can then be used in an application to change its functionality.

Implementation: Scalability and extensibility are two key factors for any business. As part of the Home Service MarketPlace System to support these two attributes, we have designed a 'FilterStrategy', that houses an array of algorithms which decide the filtering strategy to be used for searching services based on parameters such as city, skill, status. These algorithms are designed as individual classes such as 'CityStrategy', 'SkillStrategy', 'StatusStrategy', which are in turn the concrete implementations of the interface 'IFilterStrategy'. The idea is that these algorithms are made available to the clients via an interface class, in our case the 'IFilterStrategy' class, and at any given point in time the implementation of the concrete classes aka the algorithms' implementation can vary, but the client does not get affected by this, as it is just utilizing the abstraction of these classes. **This is a pattern that supports the thought 'program to interfaces, and not implementation'**.

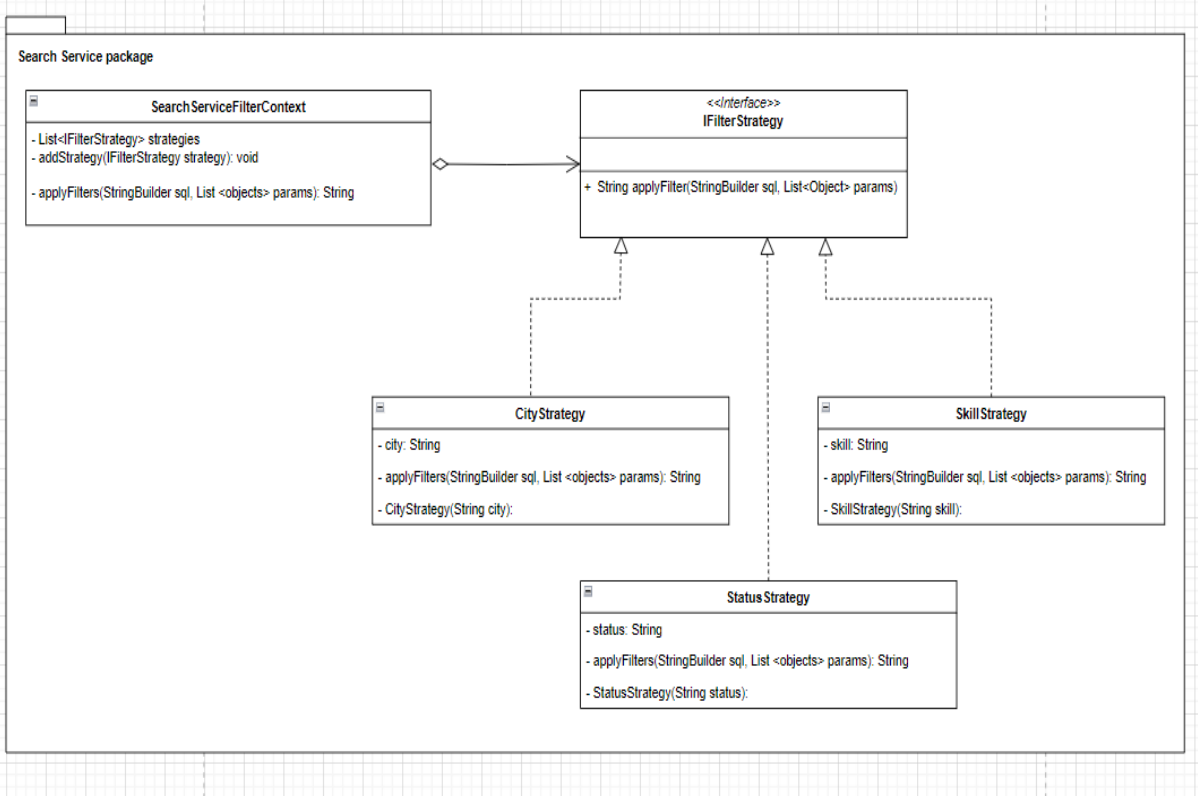


Figure 75: Strategy Design Pattern Design time class diagram

3. **State Design Pattern:**

Definition: The State Design Pattern allows an object to change its behaviour when its internal state changes. This pattern is useful when the object's behaviour depends on its state, and at the same time, the state may change during the object's lifetime.

Implementation: In Our application, we have handled service provider busy or available state with the help of state design pattern, when the booking is accepted or rejected the status of service provider changes to busy or available state. State design pattern follows **Open closed principle** as it makes the system open for extension and closed for modification. New states can be added by implementing a new state class without modification and existing states do not need to be changed. We are also achieving **single responsibility principle** by state design pattern as this keeps the logic of each state distinct.

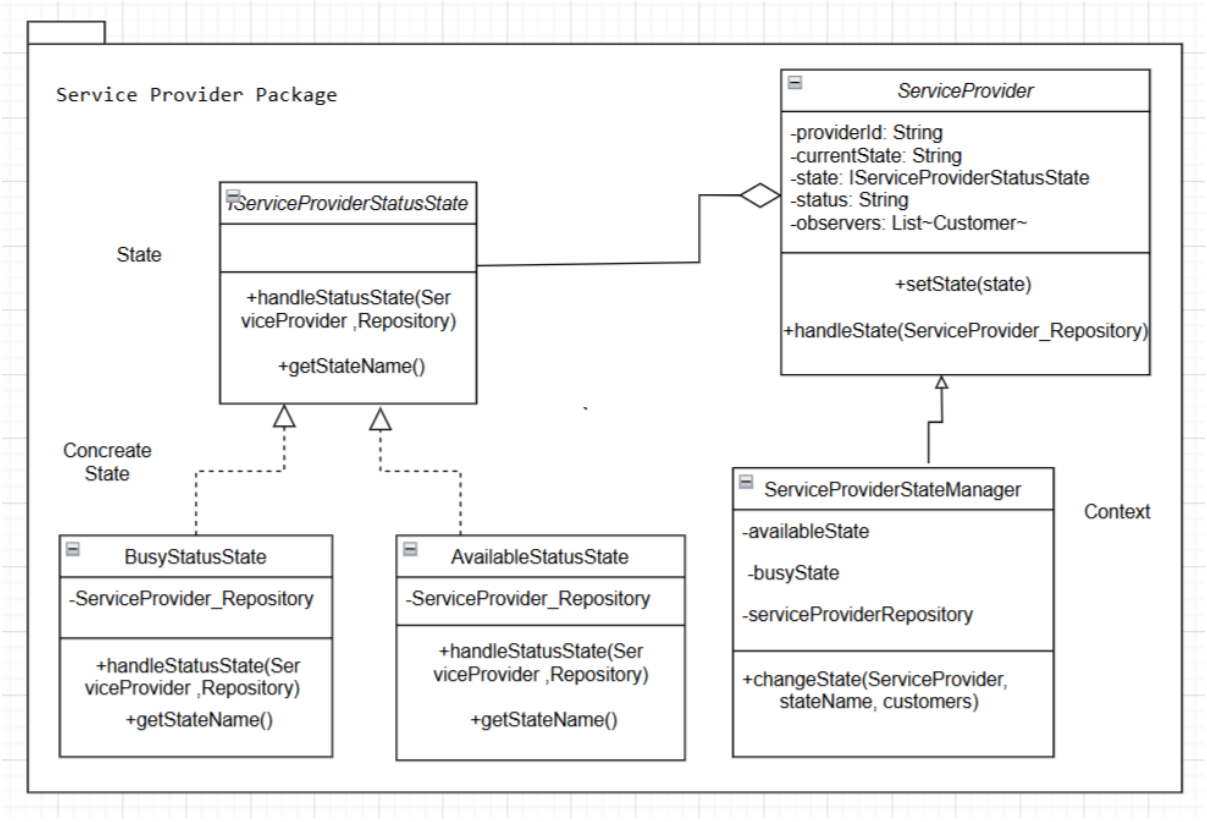


Figure 76: State design pattern design time class diagram

4. **Observer Design Pattern:**

   Implementation: Our application implements observer design pattern for notifying customers of that city about the status of service provider. Whenever status of service provider is changed all customers are notified. Here Service provider acts as a subject and customers act as an observer. Observer design pattern uses **Program to Interface, not implementation**. We are defining observer behaviour in interface; this means that subject interacts with observer through interface.

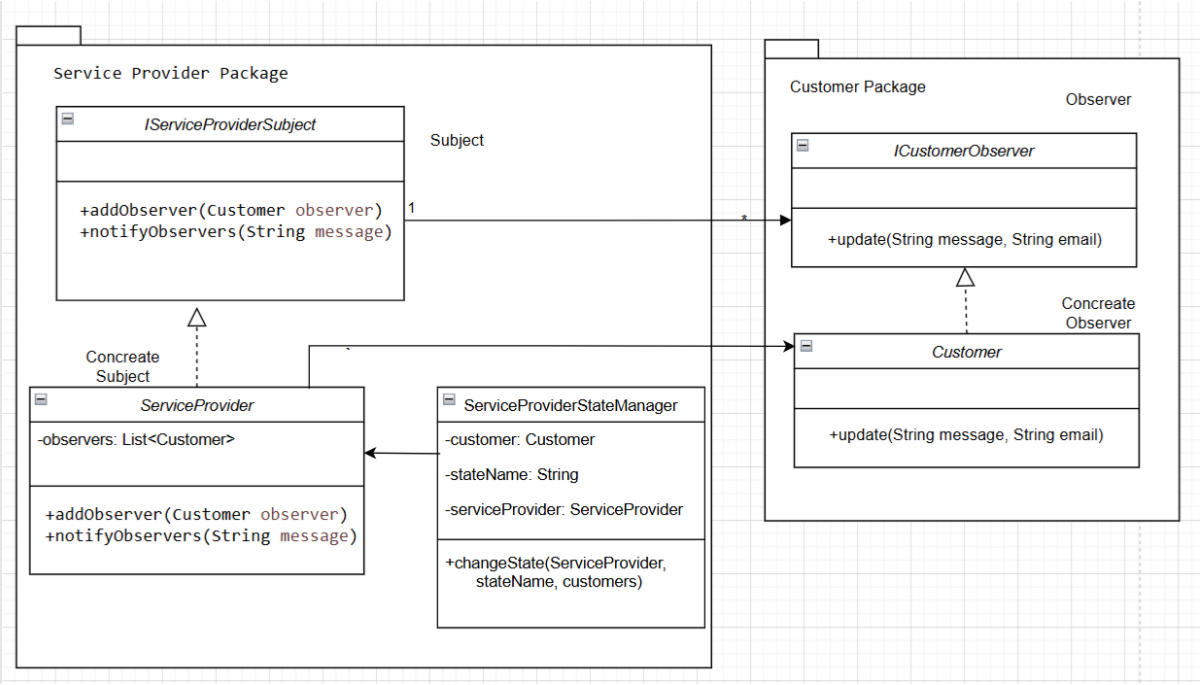


Figure 77: Observer design pattern design time class diagram

5. **Decorator Design Pattern:**

Implementation: We have implemented the Decorator Design Pattern to enhance the flexibility and modularity of the payment process. The core payment logic is encapsulated in a base Payment class, which is responsible to handle basic payment operations. To add more functionality, we have created decorator classes like DiscountDecorator and LoyaltyPointsDecorator. These decorators extend the functionality of the base Payment class by wrapping it and adding extra behaviours. The decorator design pattern follows principles such as **Single Responsibility Principle (SRP)** where each class has a specific responsibility, **Open/Close Principle (OCP)** where we can add new decorators without modifying the existing classes, **Liskov Substitution Principle (LSP)** where any instance of BasePayment can be replaced with a subclass or decorator without altering the behaviour.

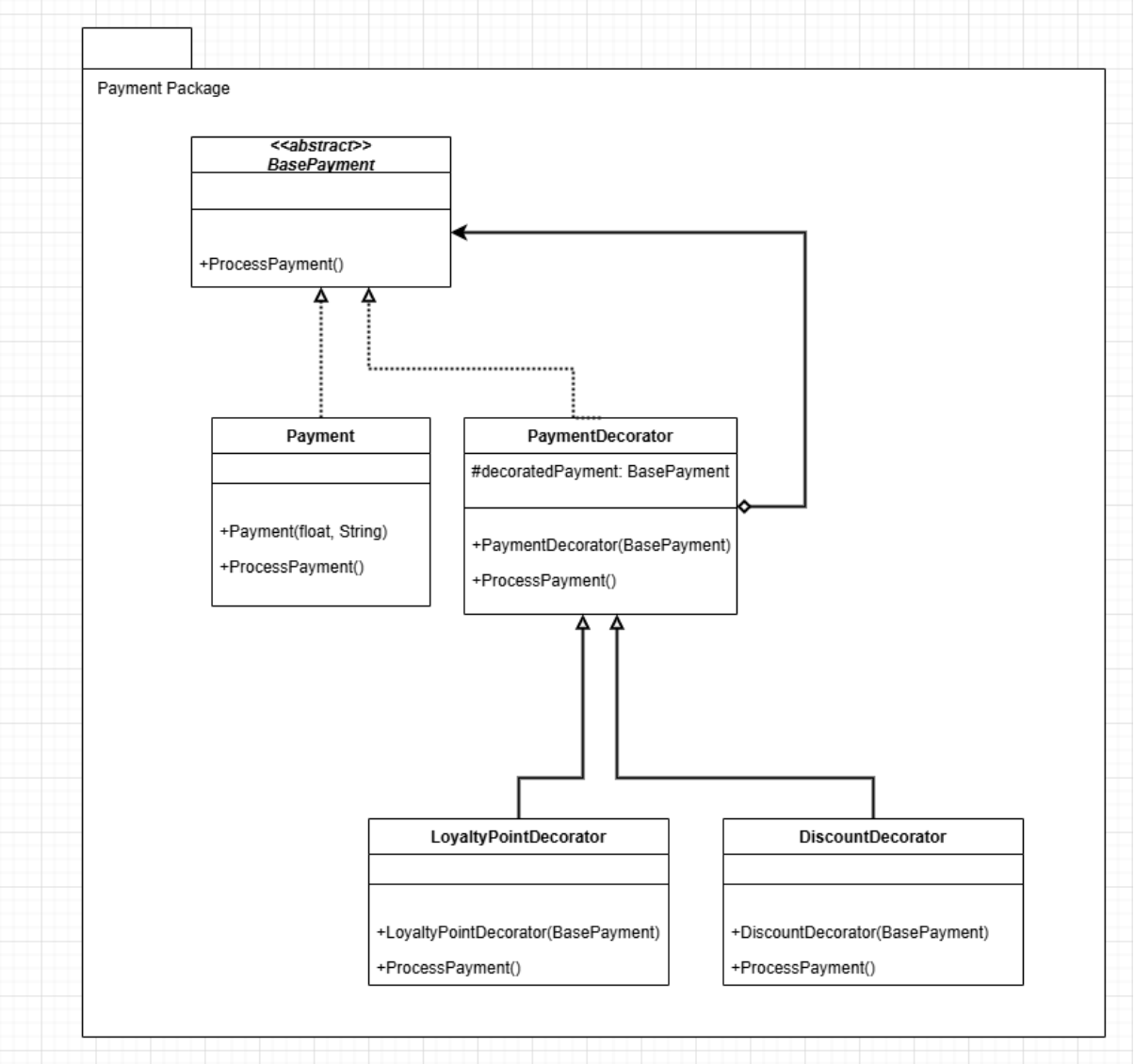


Figure 78: Decorator Design Pattern Design Time class Diagram.

6. Observer Design Pattern

Implementation: We have implemented Observer design pattern to Notify Customer and ServiceProvider about booking confirmation. Whenever status of Payment changes to COMPLETED for that particular booking both customer and Service Provider will get Notified Here Payment act as a Subject, ServiceProvider and Customer will be Observer. Observer design pattern uses **Program to Interface, not implementation**. We are defining observer behaviour in interface; this means that subject interacts with observer through interface.

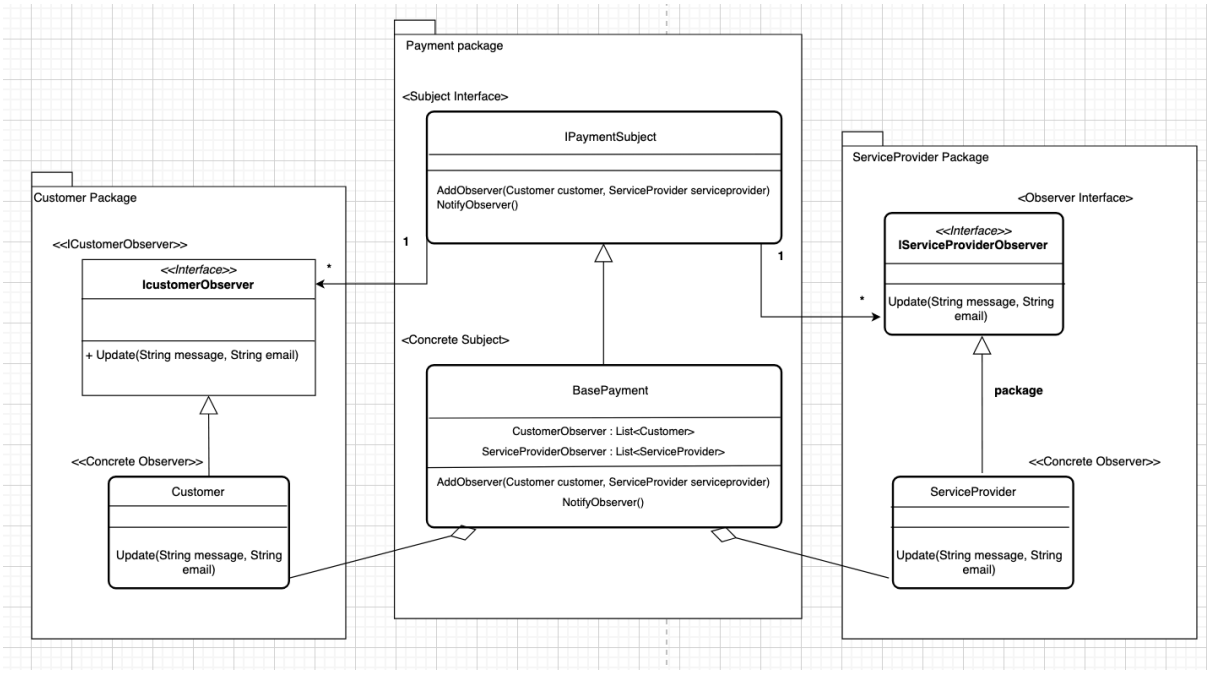


Figure 79: Observer design pattern design time class diagram

13.4 Deployment Diagram

Our project has multiple layers. The local server is host:8080 on which we will see our application hosted. The spring boot has embedded tomcat in it, which helps run the application. And our environment which is our IDE through which we start our application. Then we have our database and different classes in our packages.

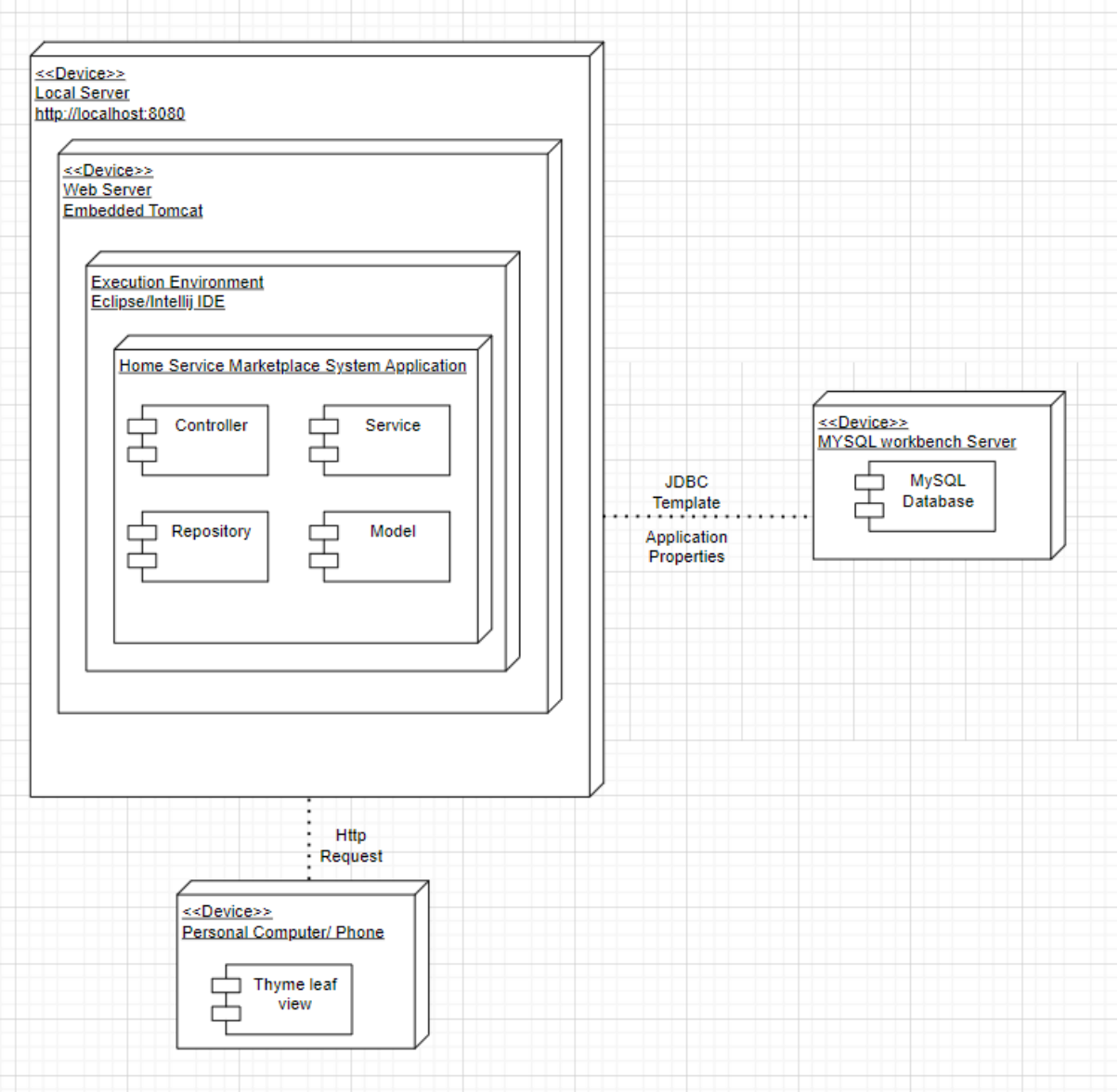


Figure 80: Design Time Deployment Diagram

13.5 Design Time Sequence Diagram

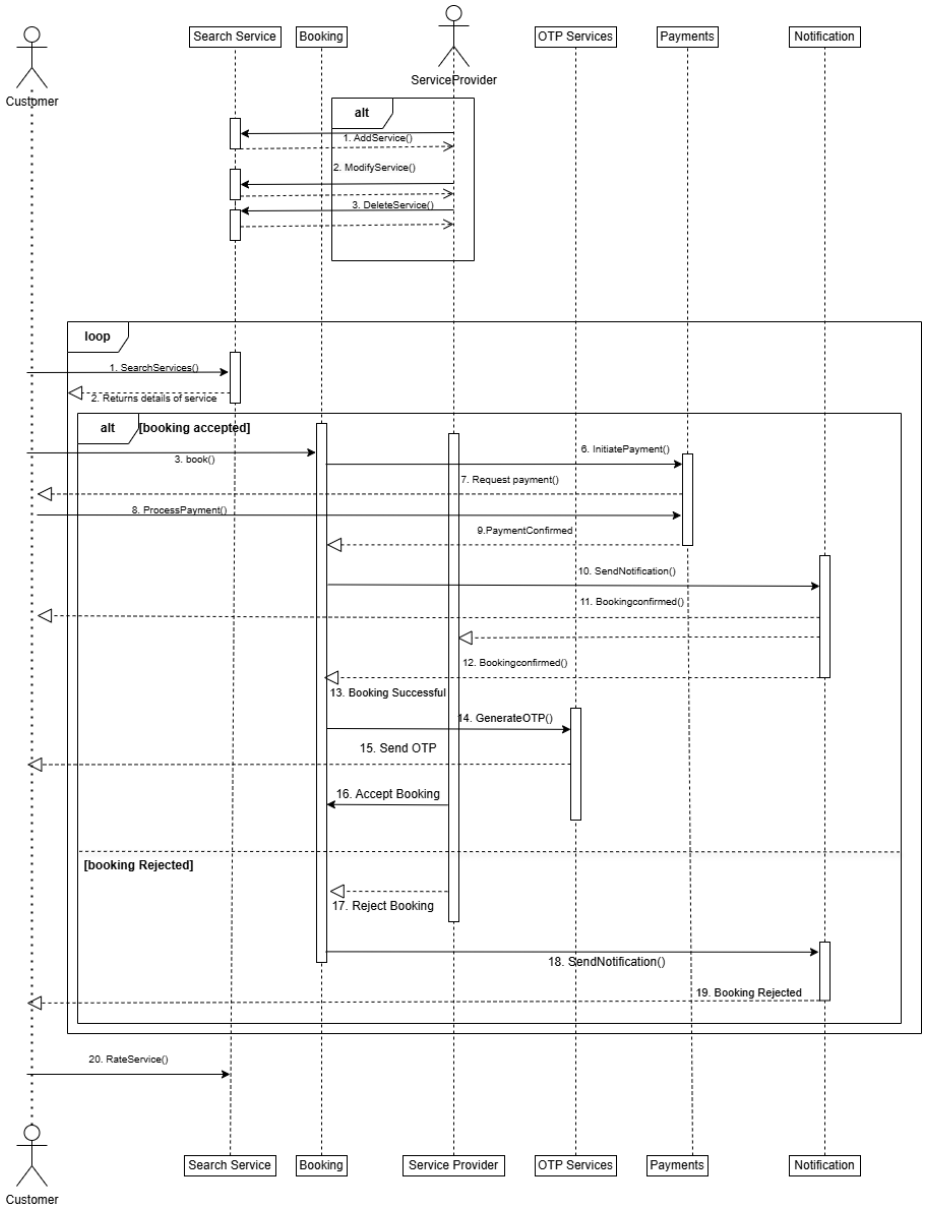


Figure 81: Design Time Sequence Diagram

13.6 Design Time Component Diagram:

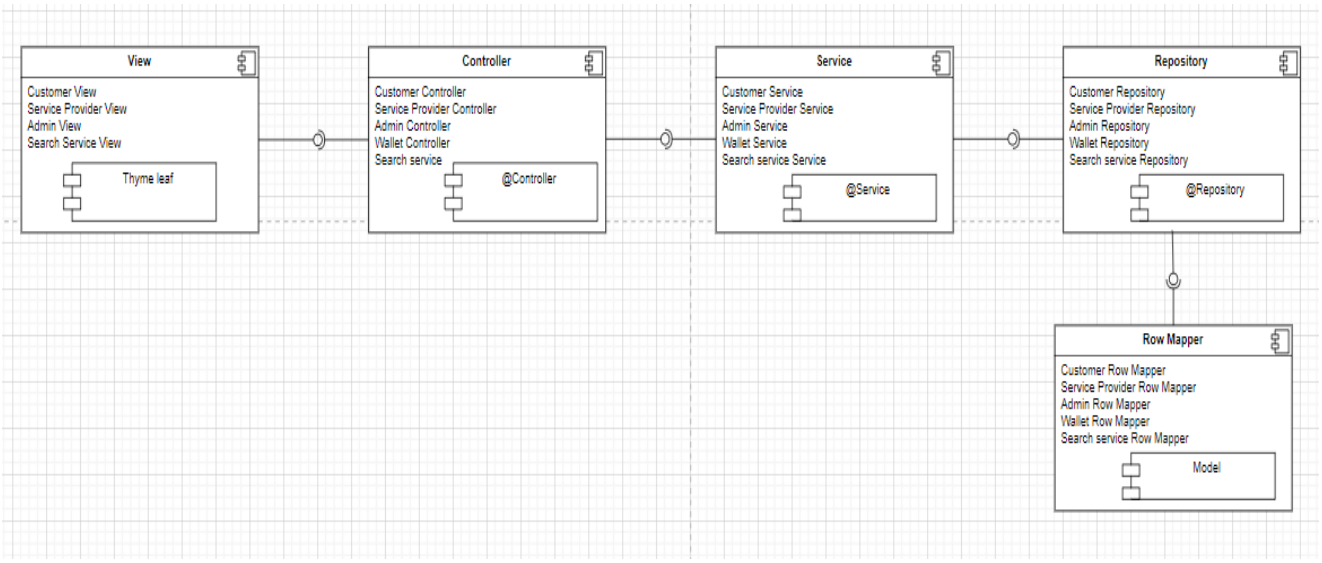


Figure 82: Design Time Component Diagram.

14. Critique

This project is a huge learning curve for each team member. It has provided us with the opportunity to understand team dynamics, work smartly, and understand software design from a completely new perspective. In the early stages of the project, the time spent understanding the strengths of others played a crucial role. It has greatly helped us in task allocations that allow team members to play their respective strengths and work effectively together. All programmers were new to the use of Java for web development, and we saw it as a challenge to learn spring boots from scratch. When using Java and spring boot, we found it difficult to implement design patterns, which confused with certain patterns. Over time, we read, brainstorm and slowly implement 5 design patterns into various areas of system business logic. We learned to draw several types of UML diagrams and understood how each diagram clearly defines and describes each system in its own way. The diagrams that had been reconstructed from implementation clearly show the gaps in our initial understanding of the system. When working in Spring boot, we used controllers, services, and repository to better structure code, allowing us to reuse the functionalities shared between subsystems. Overall, the project opened new ways of learning, and as future software engineers, we are confident that we have made a small but decisive step to put what we learnt into practice.

One thing we fell short is we were not able to implement all the use cases. Also, we were not able to write more test/Junit cases to cover our entire project.

15. Reflection

Ashutosh Lembhe: Coming from a no code/low code background, this project has been a great opportunity for me to develop my coding skills particularly in Java and learn more about how to develop web applications using Spring boot. The practical implementation of design patterns, creating deployment diagrams and integrating user interfaces with backend. I also got the opportunity to write the Junit test cases which helped me understand how to write automated test cases and dive deep into implementation of end-to-end project like this. I learned how to use post-man to mimic the API's and test them as well. I also developed my first front end view using thyme leaf which. I was also able to understand how to create my own database in MySQL from scratch. I was also able to understand how version control works.

The project has not only given me experience in technical skills but also from a teamwork and team collaboration perspective. I enjoyed working with my teammates on brainstorming on new tasks and problems.

Dhruv Upadhyay: This project was a new learning opportunity for me. It made me confident with my Java and spring boot concepts and design patterns. Starting this project from scratch from building class diagram to implementing design patterns, all this made me more confident with software development. This project was great in implementing all the concepts of Software Design subject in practical use. This project has changed my perspectives about software, now I look at software product, I find myself thinking more about its architecture. In this project I have implemented Junit testing, and I was also InCharge of GitHub which helped team collaborations and made me confident with the tool. We also implemented Sonar Lint, which helped me learn about bad code smells, and refactoring techniques.

Working closely with a team gave me experience of team collaboration and taught me to take ownership of task.

Achyutam Verma: The project has been quite challenging and very rewarding too, for a professional like me who comes from a testing background. It has really given me an insight into Spring Boot and its capabilities, especially in respect to MVC architecture, integration with RESTful API, and JDBC implementation. Each of these concepts was new, making the journey enriching and transformative. I also learned the implementation of Spring Boot in Java projects using JDBC, which really emphasized for me how to construct a secure and scalable application. Further, I got hands-on experience with MySQL database integrations into Java projects that were very instrumental in enhancing my database management skills.

Through effective teamwork, Regular group discussions were instrumental in solving complex problems, sharing innovative solutions, and making sure that everyone was aligned with the objectives of the project.

Gorav Sharma: As a fresher, this project was an invaluable opportunity to kickstart my journey in software development. While I had some exposure to Java and JavaScript, working with Spring Boot was completely new to me. Learning to build a web application from scratch provided hands-on experience with implementing design patterns, creating deployment diagrams, and integrating a user-friendly interface with robust backend services.

Additionally, I got the chance to explore advanced concepts like the Decorator Design Pattern to manage the payment operations effectively. Collaborating with my team not only helped me gain

insights into different approaches to problem-solving but also improved my understanding of team dynamics and ownership of tasks.

Overall, this project has significantly boosted my confidence in software development and broadened my understanding of building scalable and efficient applications.

16. GitHub Link for Code.

Code link in GitHub: https://github.com/AshutoshLembhe2000/HomeServiceMarketplaceSystem

17. References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional.

[2] https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/

[3] https://refactoring.guru/design-patterns

[4] Previous year report.

[5] Udemy.

[6] YouTube, for design patterns and spring boot concepts.

[7] https://docs.spring.io/spring-boot/index.html

[8] Figure 7: https://medium.com/@jacoblogan98/understanding-the-mvc-with-rails-c5222e7e81d2

[9] https://reqtest.com/en/knowledgebase/agile-development-process