Project

CS4125 - Systems Analysis and Design Project

Sean Fitzgibbon - 19273444
Daniel Larkin - 19257503
Colum Maher - 19253443
Adam Butler - 19255967

GitHub Repository

https://github.com/ColumMaher/cs4125_project

Narrative

Concept

This project will involve designing and implementing a car rental system. This system will allow users to rent a car using our service and retrieve the car from a depot. Car rentals would be approved by an administrator, and inspected by an administrator once again when a user returns the rented vehicle after use.

Users would go through the process of registering an account, and logging in to browse car rental listings in the ‘Browse’ section. Once a user finds a car they would like to rent, they can select it and go through the process of adding a deposit and paying rental fees. The user then retrieves the vehicle and once finished, returns it to the depot again.

The GUI that the user is faced with is simple, as we prefer to focus on ensuring that functionality is present over anything else. The GUI would feature everything that is important to the user, such as current listings and feedback on their use (i.e. deposit paid, car reserved etc). It would also include the login form, and the possibility of adding discounts when checking out.

We also have a GUI for administrators, which allows them to add/remove listings from the system, and edit existing listings. They can mark vehicles as unavailable, and also approve user requests/rentals on vehicle listings.

Return users of our system will be eligible for small discounts on new rentals, and administrators would be eligible for a staff discount rate on certain cars.

This system will allow for extensibility down the line, should the business grow beyond expectations. This means we can fit the business needs by expanding the functionality of our system and adding new features, and also altering existing features if it suited the business.

Software Lifecycle

For our project we choose to adopt the agile software lifecycle as the agile model focuses more on flexibility while developing a product rather than on the requirement. This feels like the most sensible option as generally requirements and software evolve throughout the project timeline, using this approach ensures we are better equipped to handle these changes effectively in order to prevent delays in the project. Part of the principles of agile is to deliver valuable iterations regularly with the value of group work and communication is also stressed. This best suits our approach to the project which will be tackling the workload week by week as outlined in the project plan.

The Agile model is a combination of the iterative and incremental model, the appeal of agile over other models is the flexibility provided while developing a product rather than focusing on the requirement. The agile model consists of 'sprints' where the goal is to finish the next iteration of the project, sometimes changing the software or adding new features to meet requirements. The agile approach isn't perfect however as there is generally minimal documentation produced throughout the agile life cycles which can become an issue as the software becomes larger and more complex over time making it difficult to add features and maintain.

The agile software development life cycle can be consists of six phases:

Concept

The first phase involves conceptualizing and outlining the scope of a project. Key requirements will be discussed and documentation will be created to outline them to ensure the requirements are met. Time and cost will be estimated and detailed analysis will be conducted to determine project feasibility.

Inception

In this phase the team establishes any tools or resources needed and the design process is carried out. The project architecture will be built along with user interface mock-ups. Requirements will be completely fleshed out with input from stakeholders to ensure the product functionality meets stakeholder expectations.

Iteration

Finally the product will start to be built, the design will be coded and a prototype UI produced. The goal is to build bare functionality of the product by the end of the first iteration, with additional features added in later iterations. This stage is the most important component of agile development, with software being completed quickly and clients needs satisfied.

Release

Before release a quality assurance team will perform QA testing on the software to ensure it is fully functional. In this phase any bugs are detected and dealt with before the iteration is released into production.

Maintenance

At this point in the agile lifecycle the software is deployed and available to users. All that is needed from the development point of view is ongoing support and squashing any bugs that may appear. It's common practice to train users to use the product also.

Retirement

This phase is necessary if software has become obsolete or incompatible with the organization over time. The users will be notified of project retirement or migrated to a replacement system before support for the product will be removed.

An important task when approaching software design is mitigating risk while ensuring deliverables are met. Keeping in line with the agile principles that value communication and group work we have decided to meet at the same time weekly to complete that week's iteration's workload following the suggested high level project plan’s workflow. Each team member will be flexible as on top of completing their own workload they will provide assistance and feedback to other team members ensuring the project work is kept up to the same standard each week. This way we can deliver our deliverables in a timely manner while also not having to simplify the project to meet deadlines.

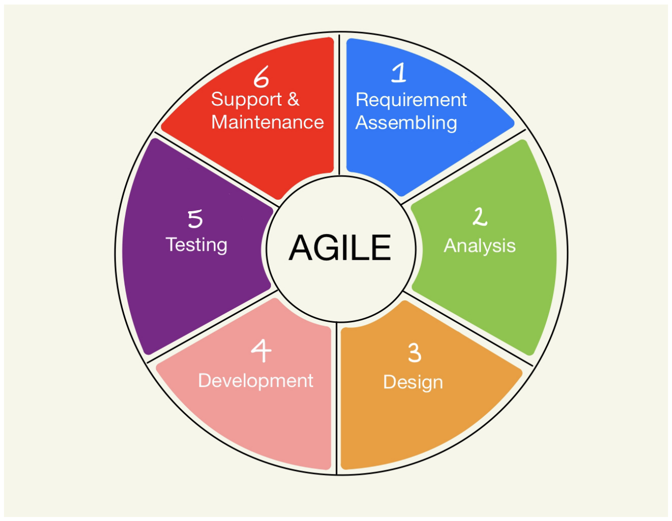


Figure 1: The Agile Development Process

Project Plan

Deliverable	Description	Responsibility	Week
Narrative	Describe the business scenario	Sean	4
Software Lifecycle	Describe and justify the chosen software lifecycle	Colum	4
Establish Roles	Specify the roles people will need to fill to complete this project	Group	4
Requirements	Functional Requirements Use Case Diagram Use Case Description Quality attributes GUI prototypes	Dan/Adam	4

System Architecture	Project Architecture MVC	Colum/Sean	5
Analysis Sketches	Design Time Class, Sequence and State diagrams	Group	6
Code	Code Implementation	Group	7-10
Design		Group	11
Added Value		Colum/Sean	11
Critique		Dan	12
References	List of sources	Group	12

Project Roles

	Role	Description	Designated Team Member
1	Project manager	Sets up group meetings, gets agreement on the project plan, and tracks progress.	Adam Butler
2	Documentation Manager	Responsible for sourcing relevant supporting documentation from each team member and composing it in the report.	Sean/Colum
3	Business Analyst	Responsible for section 4 - Requirements.	Dan/Adam
4	Architect	Defines system architecture	Colum/Sean

5	Systems Analysts	Creates conceptual class model	Sean/Dan
6	Designer	Responsible for recovering design time blueprints	Adam
7	Technical Lead	Leads implementation effort	Colum
8	Programmers	Each team member to develop at least 1 package in the architecture	ALL
9	Tester	Coding of automated test cases	Adam
10	Dev Ops	Must ensure that each team member is competent with development infrastructure, e.g. GitHub, etc.	Dan

Team Background

Name	Domain	Programming Language	Frameworks used
Colum	Data Analytics	R	Dplyr
Sean	Software Automation Engineer	SQL/Production Line Scripting	-
Adam	Software Developer	SQL/Java	AWS
Daniel	IT/Networking	Robotic Scripting	-

Requirements

Use Case Diagrams

Customer

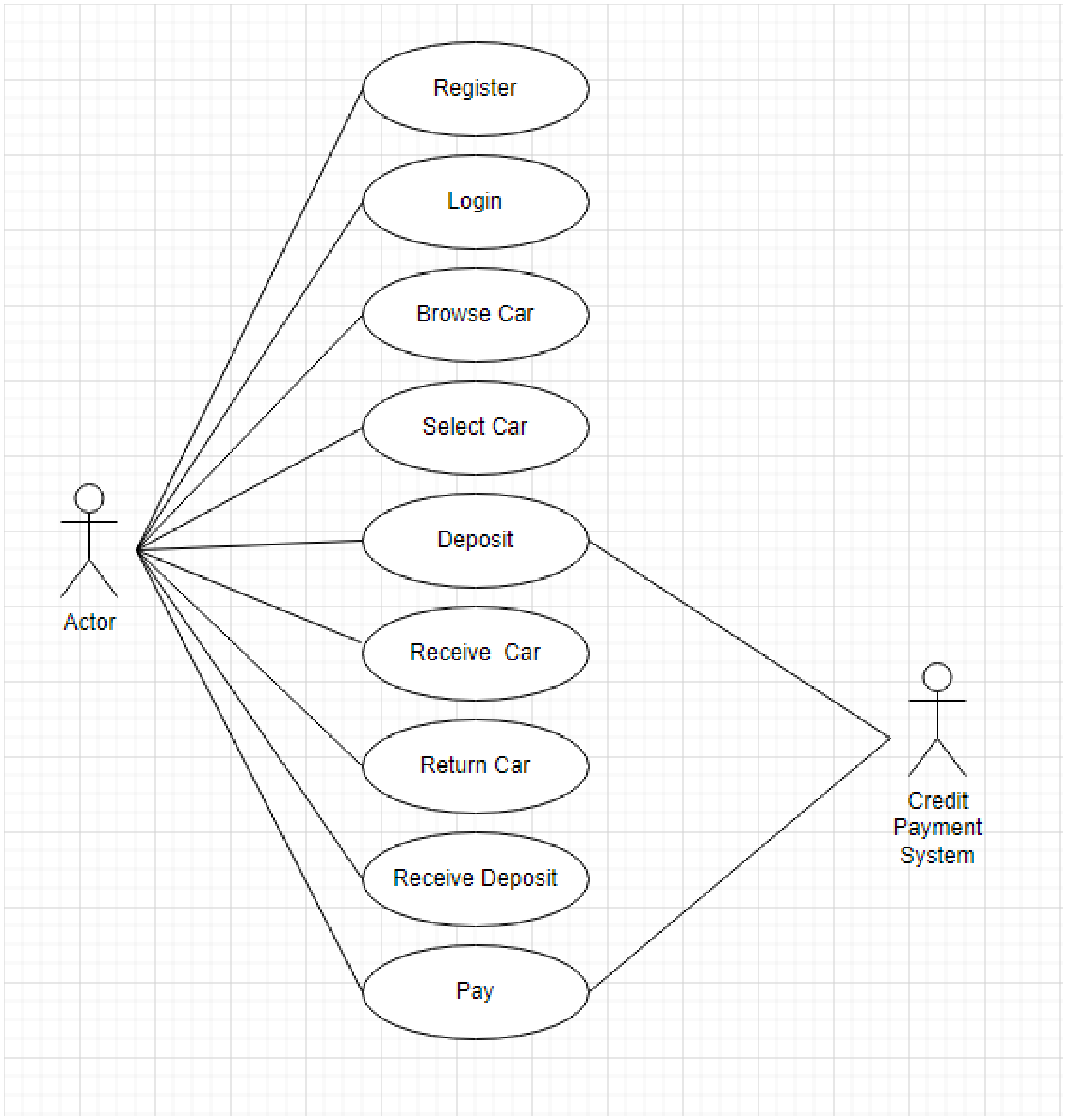


Figure 2: Customer Use Case Diagram

Administrator

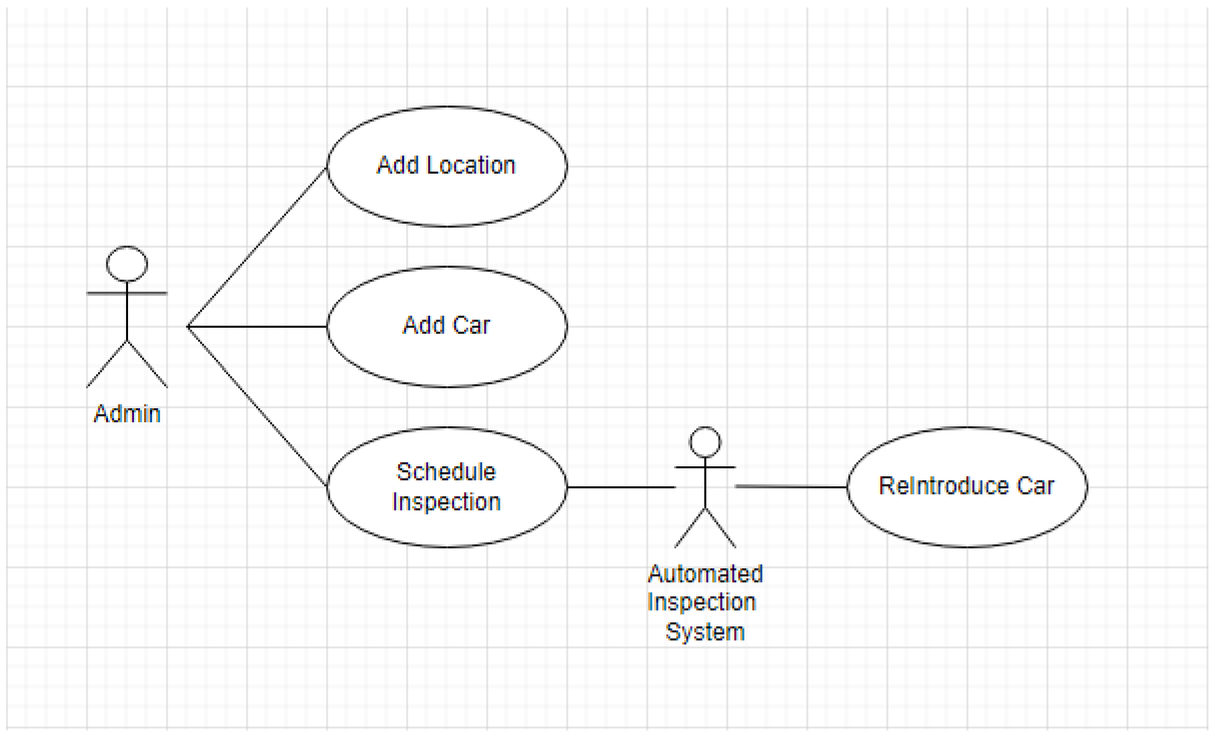


Figure 3: Administrator Use Case Diagram

Use Case Descriptions

Register

User Action:

1. The user enters their credentials and presses Sign Up

System Response:

2. The system checks that the username is not already in use
3. The system adds them to the user database

Alternative Response:

4. In the event that username is taken, the system throws a message stating the username is unavailable.

Non-Functional Requirement:

Security. The user account information needs to be stored securely with encryption.

Login

User Action:

1. The user enters their credentials and presses the Login button

System Response:

2. The system checks that the login credentials given by the user are accurate
3. Logs the user into their account, bringing them to the browse vehicles page

Alternative Response:

4. In the event that the login information is incorrect, the system throws a message stating that the information is incorrect.

Non-Functional Requirement:

Security. The user account information needs to be stored securely with encryption.

Browse Vehicles

User Action:

1. The user browses through the selection of vehicles listed as available in a scrolling menu.

System Response:

2. The system presents the user with a scrolling list of the available vehicles.

Alternative Response:

3. If the user is in an area with no available vehicles, the system will not show any.

Non-Functional Requirement:

Convenience. This menu should be simple with the relevant information that the user needs, like the number of seats in the vehicles, visible.

Select Car

User Action:

1. The user presses on a vehicle they want to choose to confirm their selection.

System Response:

2. The system will present the user with a page to pay a deposit on the vehicle to ensure its return.

Alternative Response:

3. N/A.

Non-Functional Requirement:

Reliability. This step is important to the user and not too complicated from a software point of view so it is important that it works well.

Pay Deposit

User Action:

1. The user will enter their payment information and will pay a deposit on the vehicle they wish to rent

System Response:

2. The system will display fields for the user to input their payment information.
3. The system will check if the payment information is valid.

4. If the information is valid, the user will be charged and the vehicle they have chosen will be removed from the list of available vehicles.
5. The system will inform the user where they must go in order to pick up their vehicle.

Alternative Response:

6. If the user does not have the funds to pay the deposit, the system will give a message stating the transaction has failed.
7. If the user has input their payment information incorrectly, the system will give a message saying the payment information is invalid.

Non-Functional Requirement:

Security. It is critical that users' payment information is stored in a secure encrypted manner.

Receive Car

User Action:

1. User goes to the car park at which their vehicle is stored.
2. The User collects the keys for the vehicle from the on-site lockbox.
3. The User goes to the parking space shown on the app to find their vehicle.

System Response:

4. The system will display the screen showing the time the user has had the vehicle and current cost for their rental period, along with a button to end the trip.

Alternative Response:

5. If during the trip, the vehicle detects it is low on fuel it will notify the user via the program that they must refuel in order to be able to reach a drop-off point.

Non-Functional Requirement:

Reliability. It is important that this part of the program works consistently as it would be highly inconvenient for the user and the staff if a vehicle is left away from a drop-off point and has to be brought back.

Return Car

User Action:

1. The user presses the “End Trip” button.

System Response:

2. The System will display a map with directions to the nearest drop-off location.
3. Upon reaching the drop-off point the System will say which parking space to leave the vehicle.
4. Upon parking the vehicle, the System will inform the user which on-site lockbox to leave the keys in.

Alternative Response:

5. User does not return the vehicle back to the drop-off point.
6. If the vehicle is not returned to drop-off, it will enable the vehicle’s hazard lights, and slow to a stop.
7. In this event, a staff member will have to go and collect the vehicle.

Non-Functional Requirement:

Usability. It is important that the user is able to easily find their way to the drop-off.

Receive Deposit

User Action:

1. The user must return the vehicle to receive the deposit.

System Response:

2. Return the deposit to the payment method it was sent from.

Alternative Response:

3. If the user does not return the vehicle, the deposit is not returned.

Non-Functional Requirement:

Security. It is critical that users' payment information is stored in a secure encrypted manner.
Reliability. It is very important for users to get their deposit back reliably.

Final Payment

User Action:

1. No user action is required as they entered their payment information previously.

System Response:

2. The System will charge the payment method previously entered for the required amount calculated based on the trip length.

Alternative Response:

3. N/A.

Non-Functional Requirement:

Security. It is critical that users' payment information is stored in a secure encrypted manner.

Reliability. It is very important that the payment system does not fail for the System to be relied on for a successful business model.

GUI Prototypes

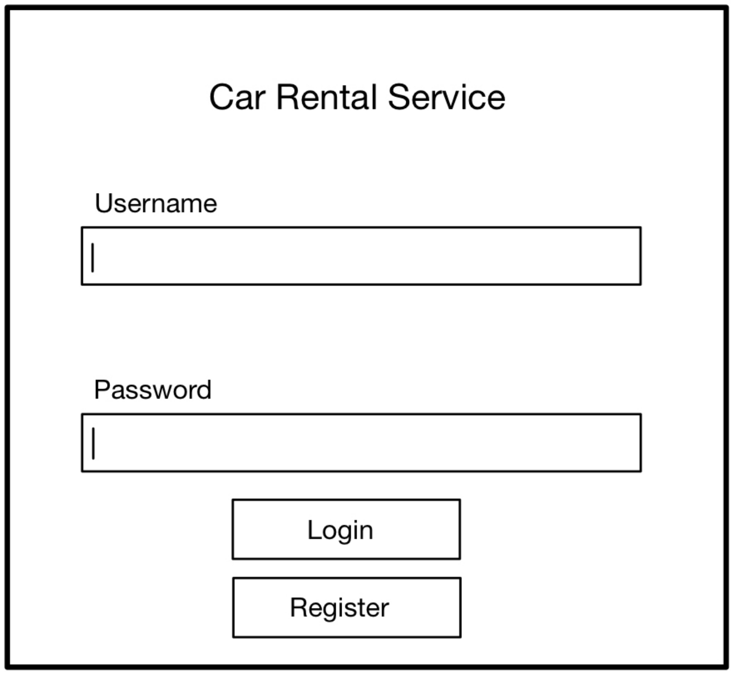


Figure 4: Login Screen Mockup

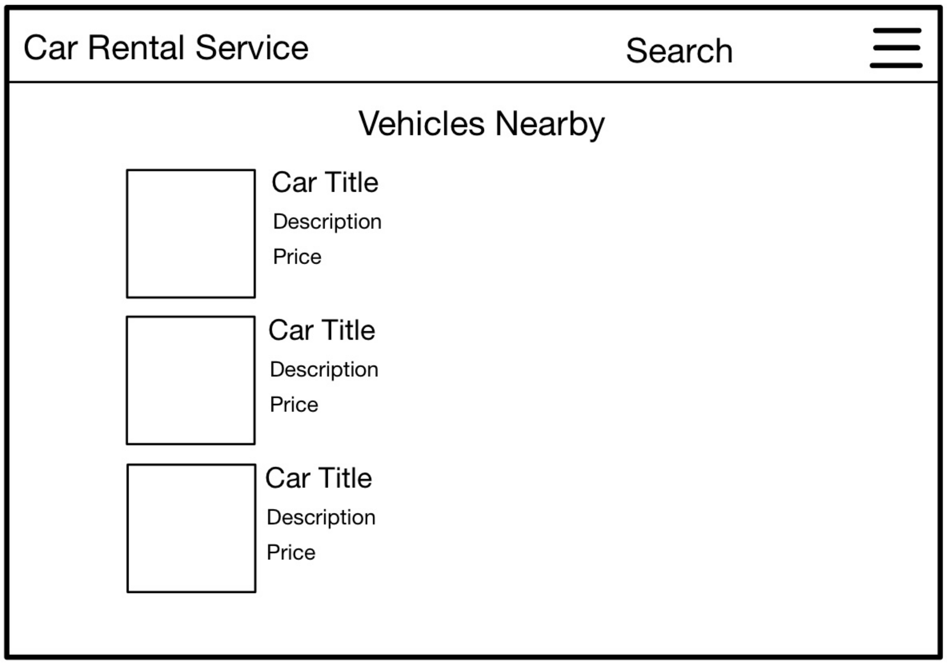


Figure 5: Browse Car Screen

Architecture

Model View Controller (MVC)

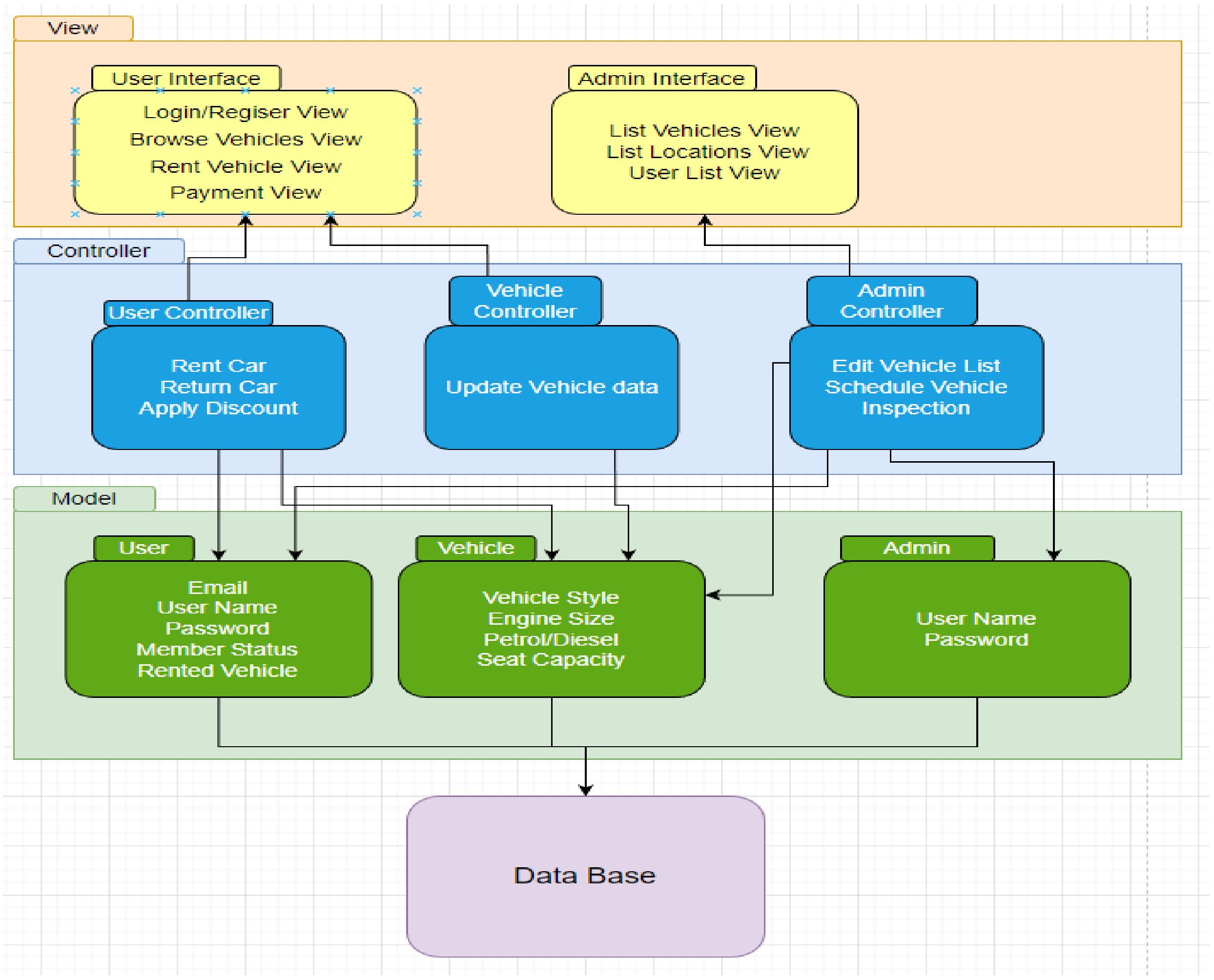


Figure 6: High level package diagram following MVC architecture.

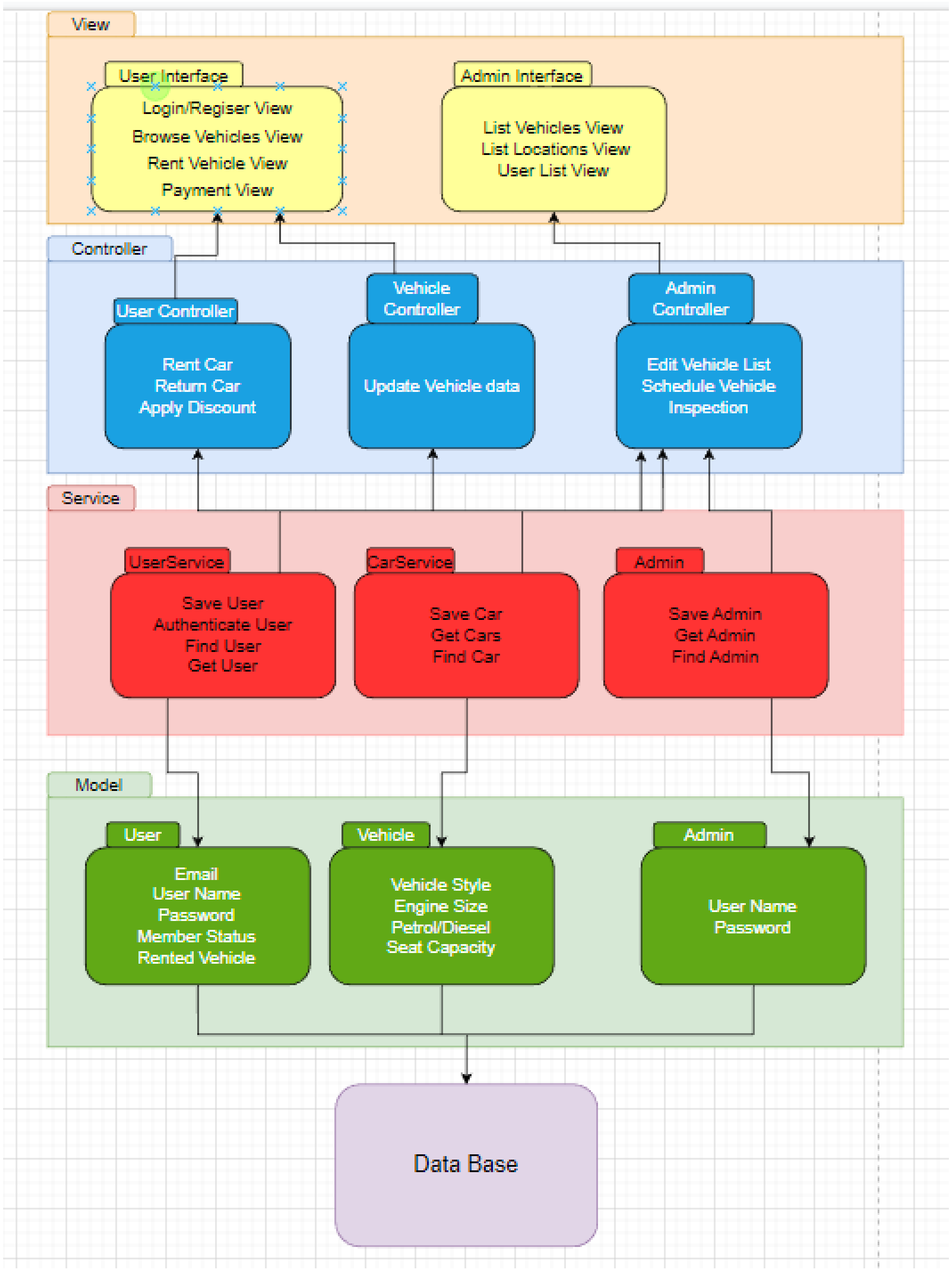


Figure 6.1: Refactor Architecture Diagram with Service Package

Above we see the MVC (Model, View, Controller) structure. This structure separates an application into three major types of components:

- Model: houses the main functionality of the application
- View: presents the user with a user interface, front end
- Controller: Manages the updating of views

By using the MVC model, we allow for maintenance and portability when updating and deploying the system. We have a clear separation of business logic, UI logic, and input logic. We also have to handle different views for different users, as seen in the MVC architecture package diagram. A user has one UI, and an admin has a separate UI. These UIs interact differently with the view and controller components of the system.

Model will control the user, vehicle and admin logic, handling user/vehicle input and admin login. View will handle the user interface, displaying a UI for the type of user logged in, either a normal user or an administrator. Controllers will deal with the updating of these components by handling information requests or data being sent by the user/administrator. We have a controller listed for user, vehicle and admin, each with their own handling of information and data. This architecture allows us to decouple key components of the system, giving the ability to make changes easily down the line, i.e. creating a new user interface or adding features to existing user interface.

For the deployment and hosting of our application and database we will be using the Amazon WebServices (AWS) platform as a DBMS for our MySQL Database. Since we are building the web-app in Java we will be using the Spring Boot API which is powerful in terms of following good design principles as it helps us to follow MVC architecture principles and implement business logic following design principles.

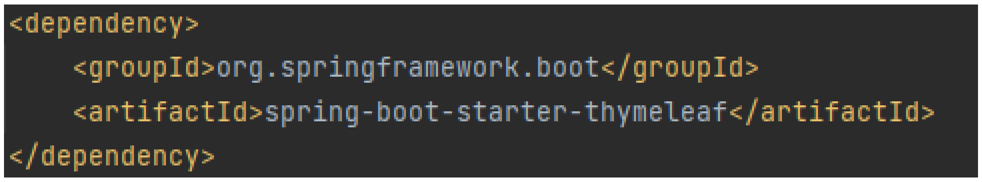
Spring Framework

Spring is an open source Java framework which can be used to develop any Java application, and includes extensions designed to easily build web-applications. This framework is organized in a modular fashion, and there are a lot of packages and classes, but not all of them are important.

The web framework for Spring is a MVC framework. It allows us to separate our classes into the Model-View-Controller split as discussed above. This enables us to decouple our program sufficiently and allow us to easily add modules using different patterns as the need arises in the future.

Spring also provides various APIs which we can use to enhance the complexity of the project. For example, the use of JDBC (Java Database Connectivity) allows us to make use of repositories and mappings throughout our program to create and access persistent data in an efficient manner.

We can also make use of libraries such as Thymeleaf, which is a server-side template engine for the creation of web environments. Thymeleaf processes HTML, CSS, JavaScript and XML, and in our project it allows us to access variables from our Java code and display them in our web-application. Thymeleaf is easily integrated into Spring, as seen in Figure 7 below.



```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Figure 7

For this reason, Spring is enabling fast development, or Rapid Application Development (RAD). This allows us to generate a system faster than if we were to not utilize a framework.

Analysis

List of Candidate Objects

User
Vehicle
Admin
Location
Order

Sketch of the conceptual class diagram

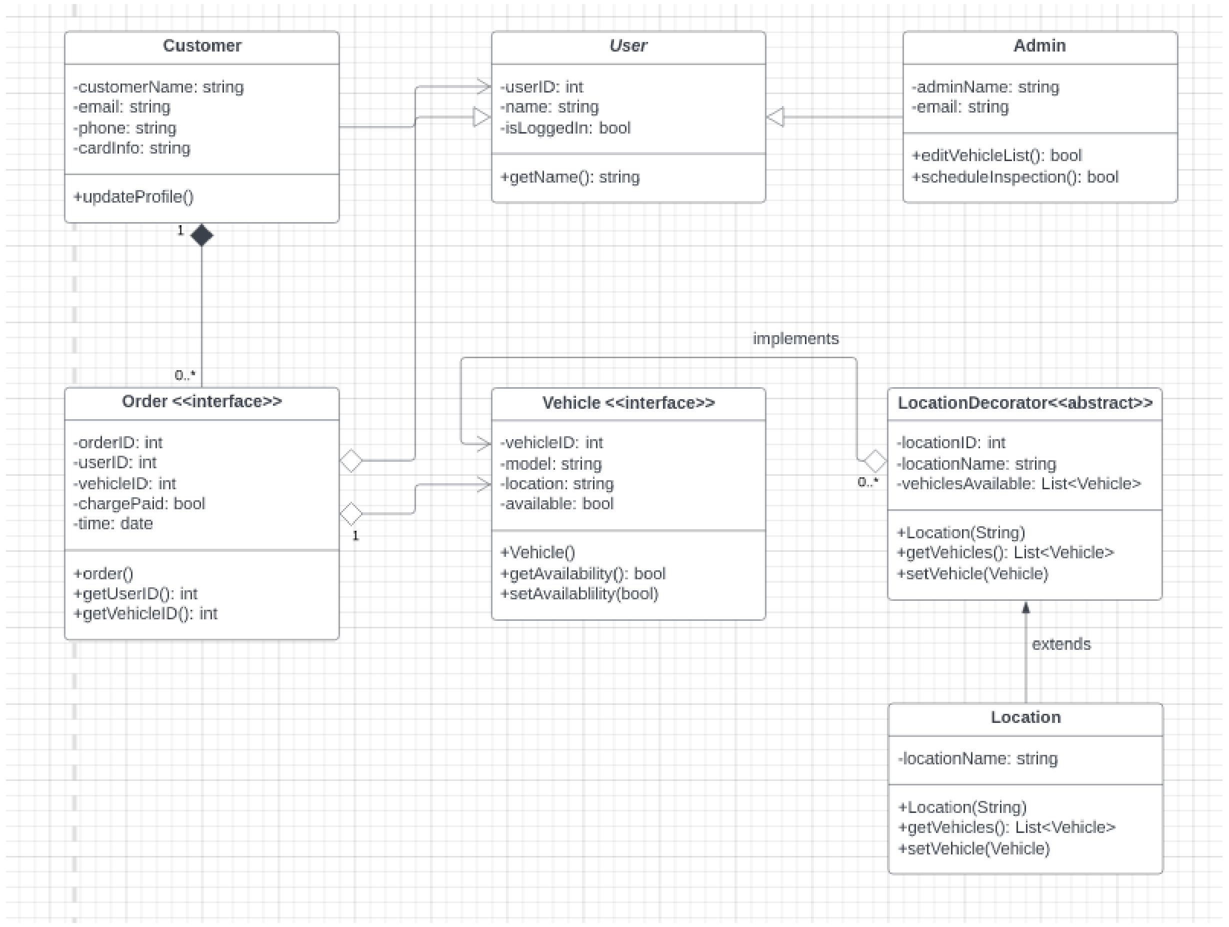


Figure 8: Conceptual Class diagram

Sequence or Communication diagram

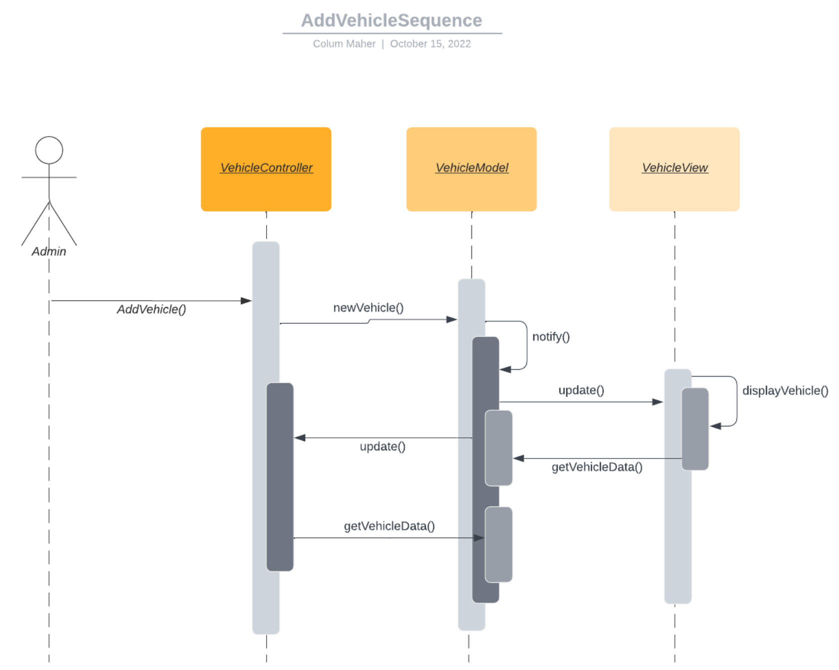


Figure 9: Sequence Diagram of adding a vehicle

State Chart

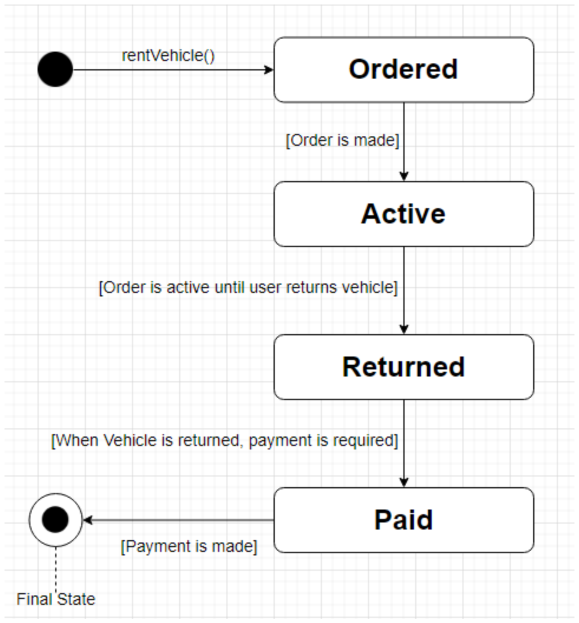

rentVehicle()
Ordered
[Order is made]
Active
[Order is active until user returns vehicle]
Returned
[When Vehicle is returned, payment is required]
Paid
[Payment is made]
Final State

Figure 10: State Chart

Activity Diagram

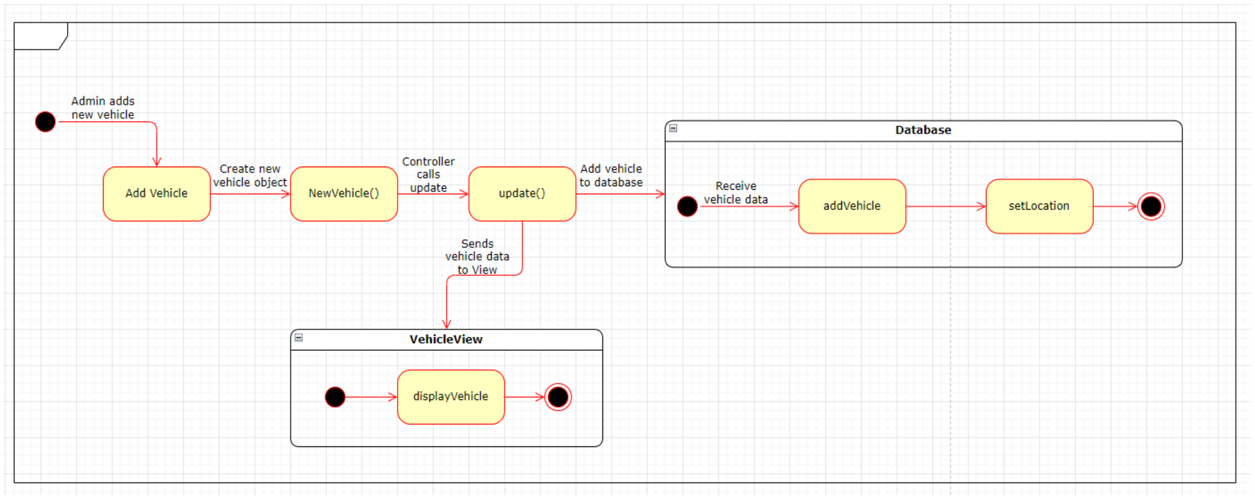


Figure 11: Activity Diagram

Entity Relationship Diagram

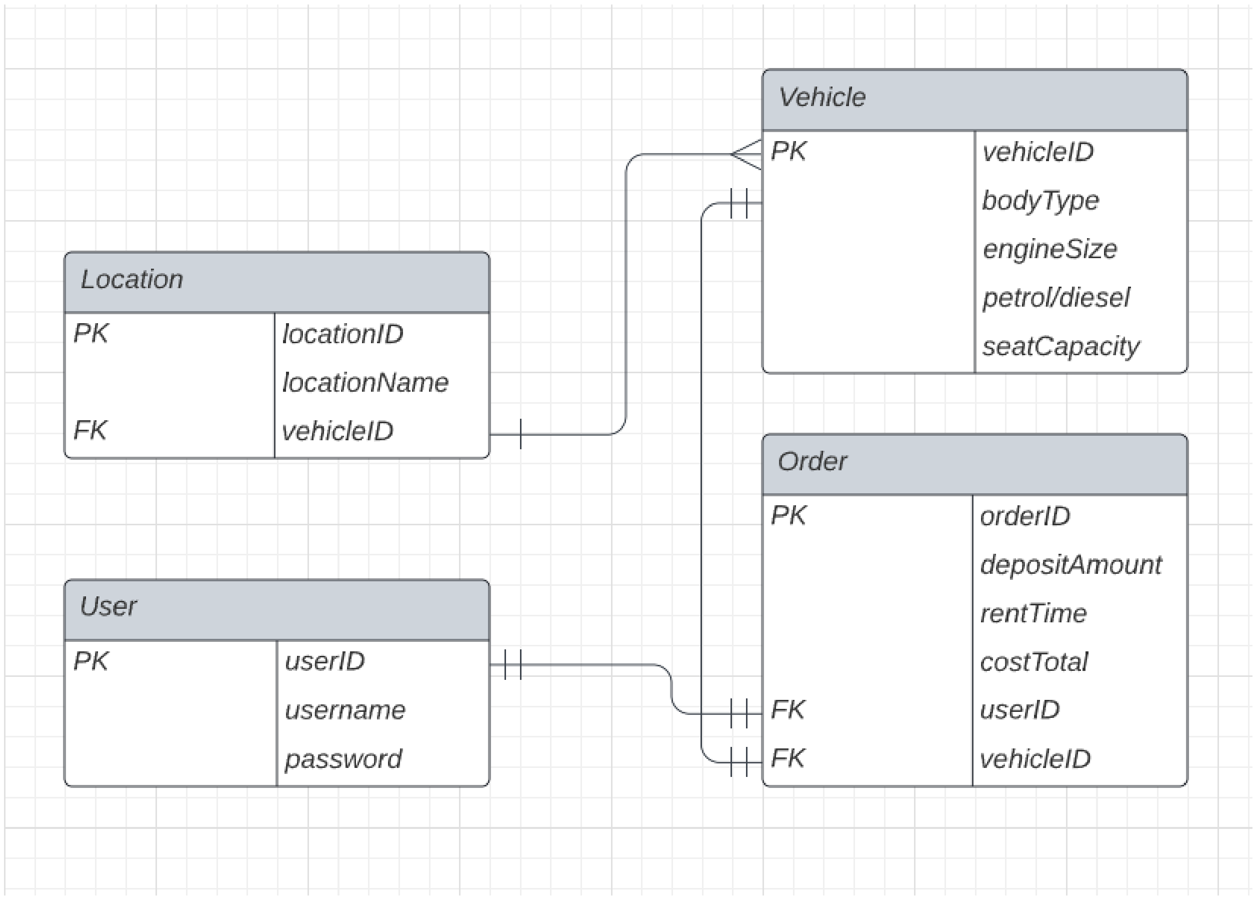


Figure 12: Entity Relationship Diagram

Transparency and Traceability

Package/Class count

Package	Num. of Classes
Controller	5
Model	19
Repository	3
Service	8
Templates	10
Static.css	1

Total Packages	Total Classes
6	46

Package Class Details

Package	Classes	Author	Lines of Code
Controller	CarController CarViewController OrderController RegisterLoginController UserController	Colum Sean Adam	28 163 31 78 29
Model	ActiveState Builder Car Director HatchBackCarBuilder Location LuxuryCarBuilder Observer	Colum Sean Daniel Adam	21 10 82 22 35 34 34 5

	Orders OrderState PaidState ReturnedState SportCarBuilder Subject SUVCarBuilder User VehicleUpdatePublisher		72 15 18 19 35 7 34 59 58
Repository	CarRepository UserRepository OrderRepository	Colum Sean	12 15 12
Service	CarService CarServer CustomerFactory OrderServer OrderService UserFactory UserServer UserService	Sean Colum	15 30 19 29 13 10 36 15
Templates	Car-list Error Index Login Order Paid paymentScreen	Daniel Colum Adam	65 12 22 23 92 39 88
Static.css	Style	Colum Adam Daniel	179

Overall Contribution

Team Member	Lines Contributed
Colum Maher	700
Sean Fitzgibbon	450
Daniel Larkin	300
Adam Butler	300

Code

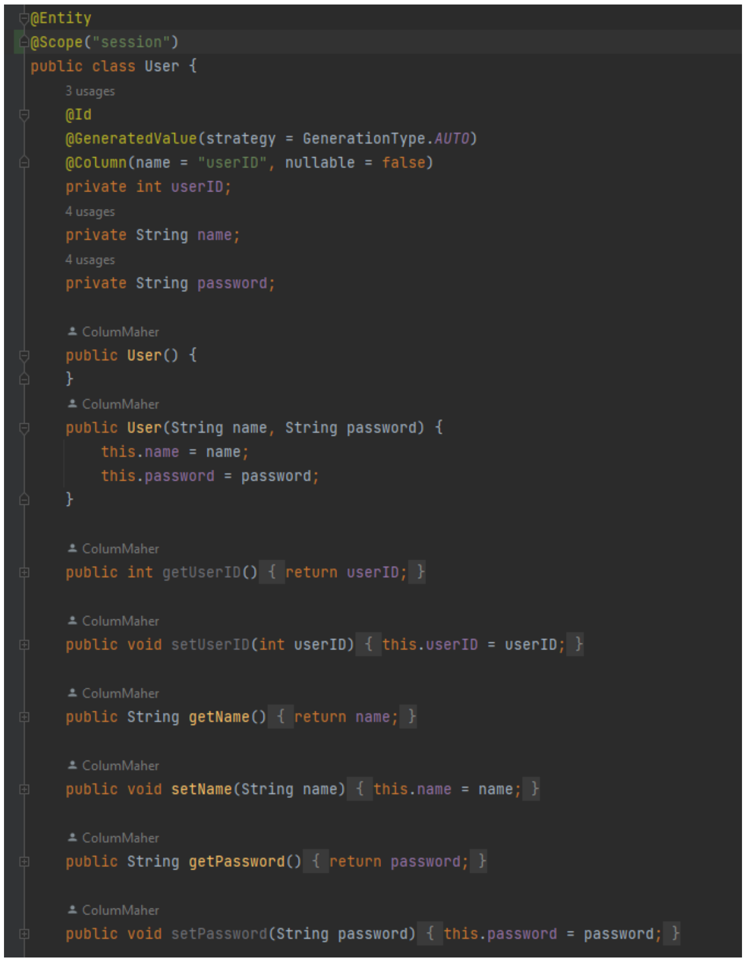
MVC

The Model View Controller (MVC) design pattern specifies that an application consists of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects. The Model contains only the pure application data. The View presents the model's data to the user. The Controller exists between the view and the model. It listens to view triggered events like a button click or form submission and executes the appropriate reaction to these events.

MVC enables **separation of concerns;** it helps break up the frontend and backend code into separate components.

Model

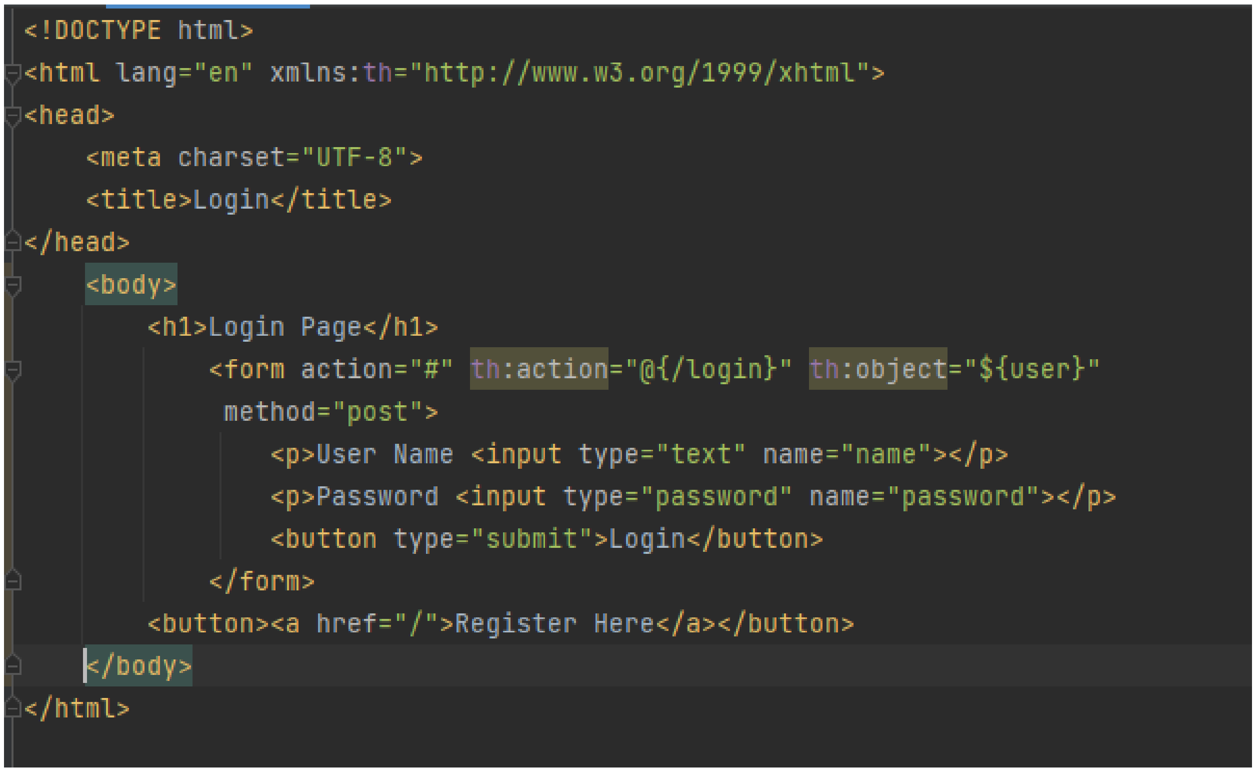
Example of a model class for our application users, denoted by the Entity annotation.



```java
@Entity
@Scope("session")
public class User {
    3 usages
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "userID", nullable = false)
    private int userID;
    4 usages
    private String name;
    4 usages
    private String password;

    ColumMaher
    public User() {
    }
    ColumMaher
    public User(String name, String password) {
        this.name = name;
        this.password = password;
    }

    ColumMaher
    public int getUserID() { return userID; }

    ColumMaher
    public void setUserID(int userID) { this.userID = userID; }

    ColumMaher
    public String getName() { return name; }

    ColumMaher
    public void setName(String name) { this.name = name; }

    ColumMaher
    public String getPassword() { return password; }

    ColumMaher
    public void setPassword(String password) { this.password = password; }
```

Figure 13: User model class.

View
Here is an example of a view in our project. In our views we use Thymeleaf, which is a template engine that works for web and non-web environments. It is best suited for serving HTML at the view layer of MVC-based web applications and integrates with the spring framework. In the example below we use thymeleaf for a form for logging in, returning a user object loaded with the credentials provided by the user.



```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
    <body>
        <h1>Login Page</h1>
            <form action="#" th:action="@{/login}" th:object="${user}"
             method="post">
                <p>User Name <input type="text" name="name"></p>
                <p>Password <input type="password" name="password"></p>
                <button type="submit">Login</button>
            </form>
        <button><a href="/">Register Here</a></button>
    </body>
</html>
```

Figure 14: Login View

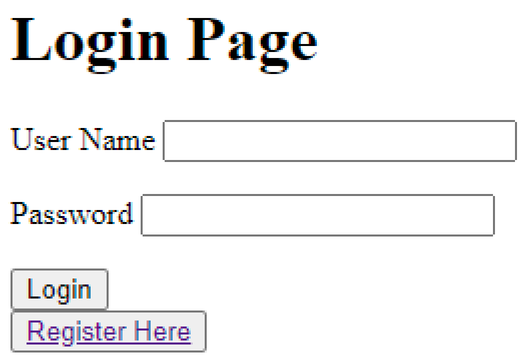
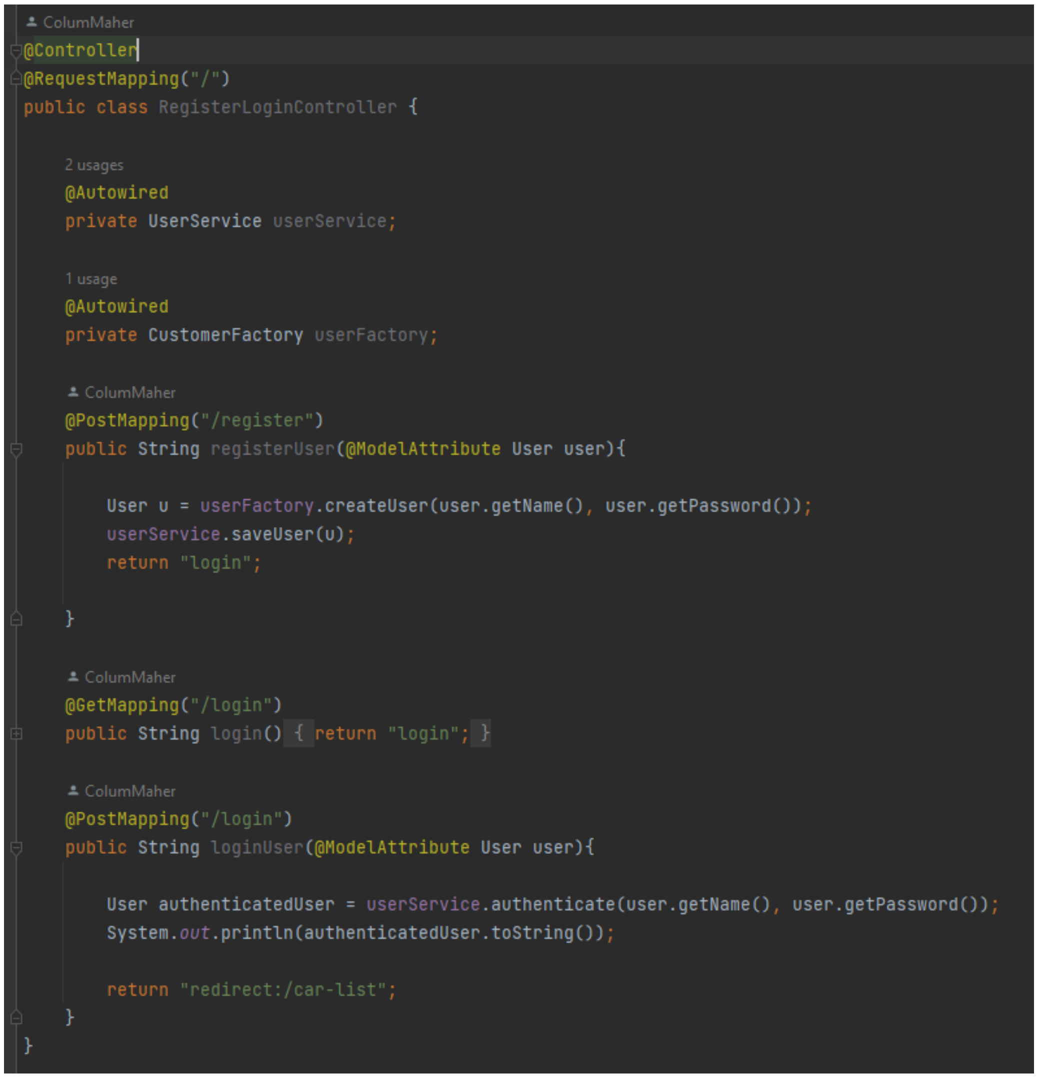


Figure 15: Login View GUI

Controller
The Controller is the brain of the application, pulling, modifying and providing data to the user. In the registration form the user provides a username and password, which is placed in a user object using the factory method and is added to the database through the user service class.



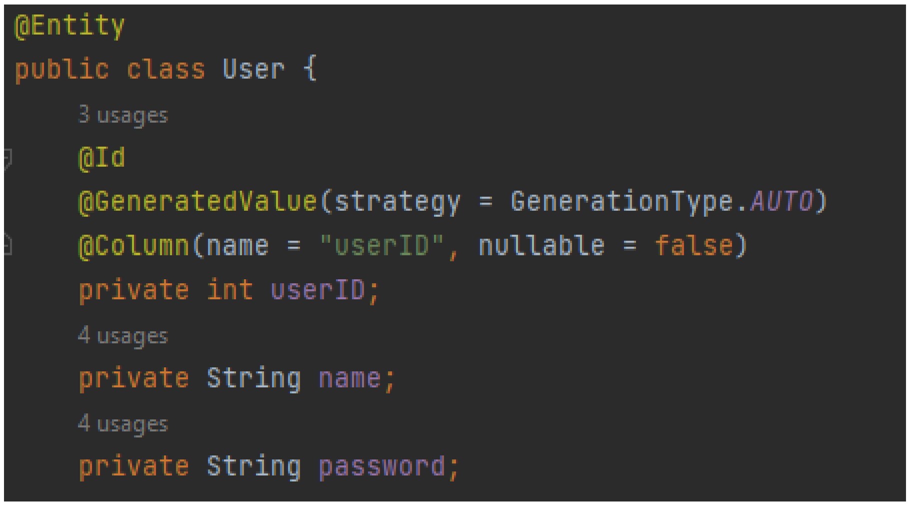
```
ColumMaher
@Controller
@RequestMapping("/")
public class RegisterLoginController {

    2 usages
    @Autowired
    private UserService userService;

    1 usage
    @Autowired
    private CustomerFactory userFactory;

    ColumMaher
    @PostMapping("/register")
    public String registerUser(@ModelAttribute User user){

        User u = userFactory.createUser(user.getName(), user.getPassword());
        userService.saveUser(u);
        return "login";

    }

    ColumMaher
    @GetMapping("/login")
    public String login() { return "login"; }

    ColumMaher
    @PostMapping("/login")
    public String loginUser(@ModelAttribute User user){

        User authenticatedUser = userService.authenticate(user.getName(), user.getPassword());
        System.out.println(authenticatedUser.toString());

        return "redirect:/car-list";
    }
}
```

Figure 16: Controller for registration and logging in operations.

Repository

Repository in Java is a pattern used to store data in a program, allowing for persistence in a system. This is achieved through the use of a data repository instantiated with variables in the code, and we use a mapping layer to map the data to the corresponding location in storage (i.e. a database).

In our program, we use a repository to store both user data and vehicles in a given location. An example of this can be seen below in Figure 17.

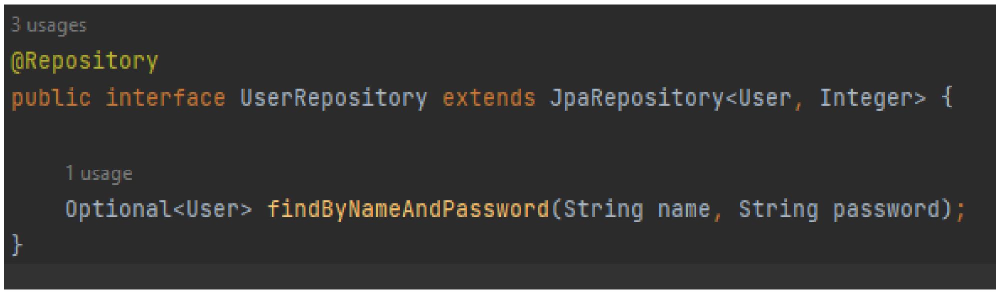


```
@Entity
public class User {
    3 usages
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "userID", nullable = false)
    private int userID;
    4 usages
    private String name;
    4 usages
    private String password;
```

Figure 17

Here we use Entity annotation to specify that this class will be an entity in our database, and we specify our variables which will be stored in the database according to the schema we set above.

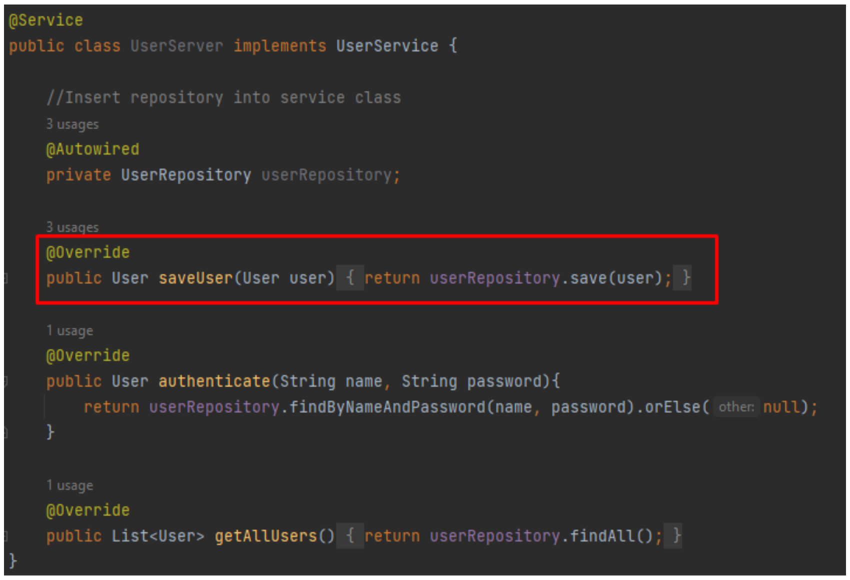
UserRepository is then created as an interface extending the JPA Repository, seen below in Figure 18.



```
3 usages
@Repository
public interface UserRepository extends JpaRepository<User, Integer> {

    1 usage
    Optional<User> findByNameAndPassword(String name, String password);
}
```

Figure 18

Using this data repository of users, we can add data, and authenticate login attempts based on the stored data. The following code snippet in Figure 19 shows the function for saving a new user to the database, based on inputted user details.

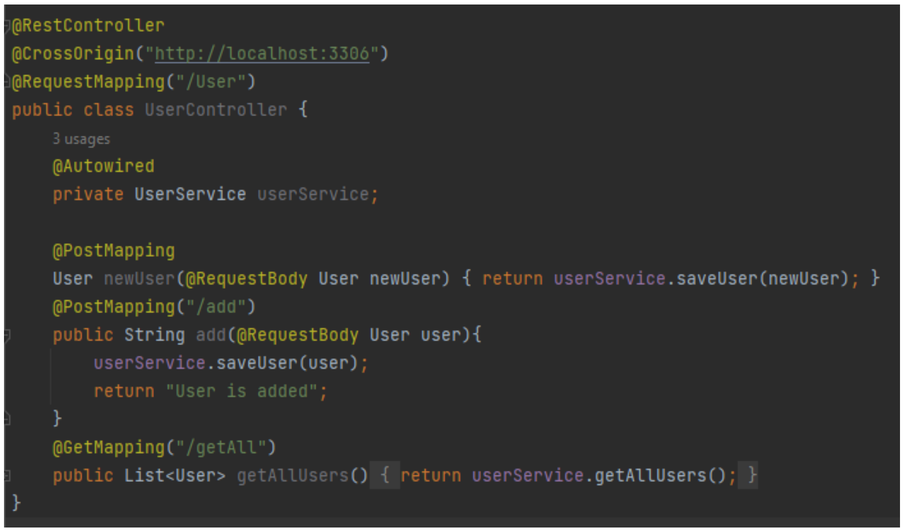


```
@Service
public class UserServer implements UserService {

    //Insert repository into service class
    3 usages
    @Autowired
    private UserRepository userRepository;

    3 usages
    @Override
    public User saveUser(User user) { return userRepository.save(user); }

    1 usage
    @Override
    public User authenticate(String name, String password){
        return userRepository.findByNameAndPassword(name, password).orElse( other: null);
    }

    1 usage
    @Override
    public List<User> getAllUsers() { return userRepository.findAll(); }
}
```

Figure 19

By using repositories in our program, we make testing of the program much simpler. The big benefit of this approach, however, is the creation of an abstract layer between data access and business logic layers, which makes our means of data access more loosely coupled in the end.

We can access the correct storage space using mappings. Below in Figure 20 is a snippet of a mapping function, which maps to the User space.

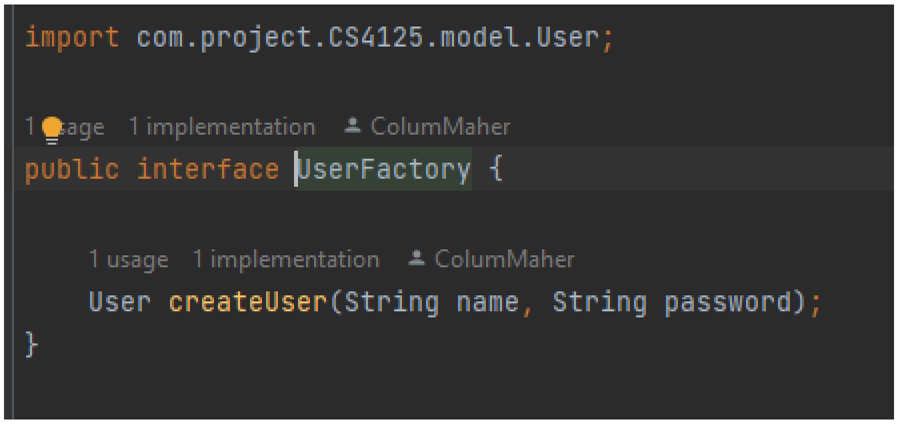


```
@RestController
@CrossOrigin("http://localhost:3306")
@RequestMapping("/User")
public class UserController {
    3 usages
    @Autowired
    private UserService userService;

    @PostMapping
    User newUser(@RequestBody User newUser) { return userService.saveUser(newUser); }
    @PostMapping("/add")
    public String add(@RequestBody User user){
        userService.saveUser(user);
        return "User is added";
    }
    @GetMapping("/getAll")
    public List<User> getAllUsers() { return userService.getAllUsers(); }
}
```

Figure 20

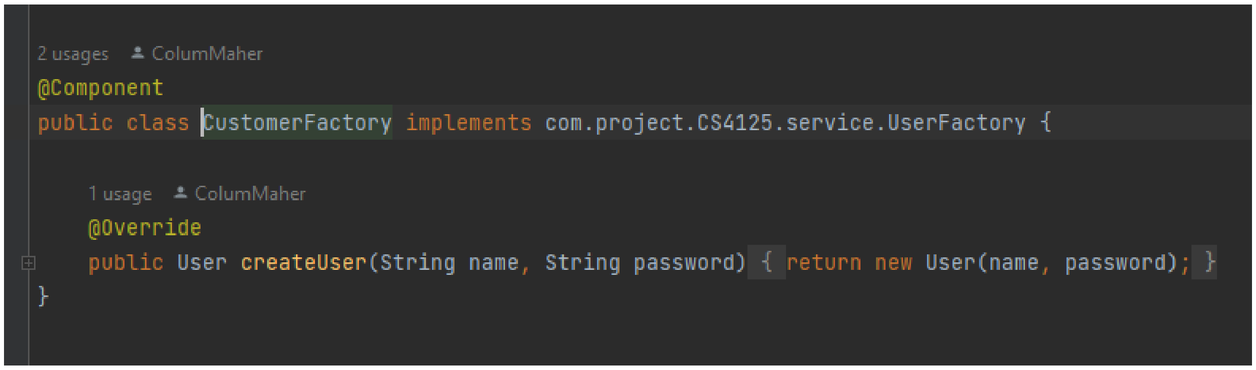
Factory Design Pattern

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass. The Factory pattern entails replacing direct object construction calls using the **NEW** operator, but it's being called from within the factory method. At first glance this just moves the constructor call to a different part of the program, one benefit is being able to override the factory method in a subclass and change the class of objects being created by the method.



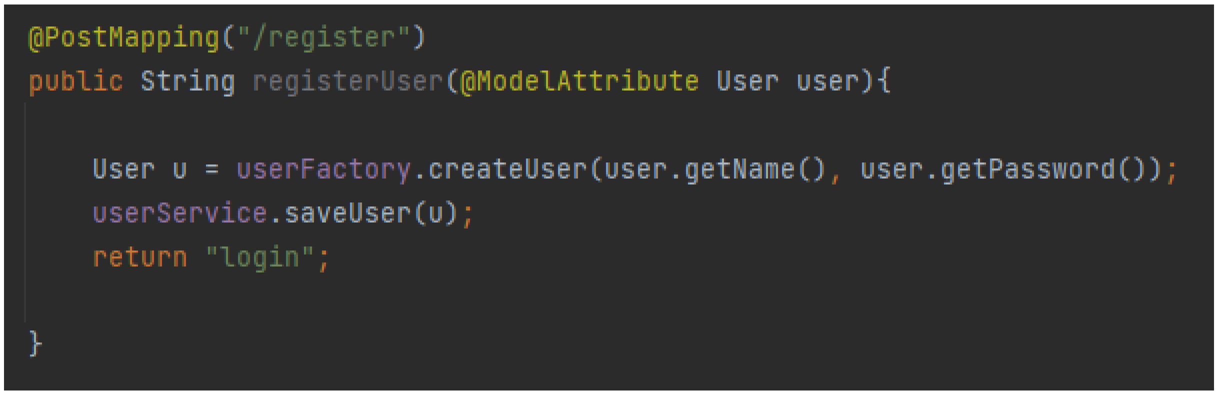
```
import com.project.CS4125.model.User;

1 usage  1 implementation  ColumMaher
public interface UserFactory {

    1 usage  1 implementation  ColumMaher
    User createUser(String name, String password);
}
```

Figure 21: Factory Method interface



```
2 usages  ColumMaher
@Component
public class CustomerFactory implements com.project.CS4125.service.UserFactory {

    1 usage  ColumMaher
    @Override
    public User createUser(String name, String password) { return new User(name, password); }
}
```

Figure 22: Simple Factory Class



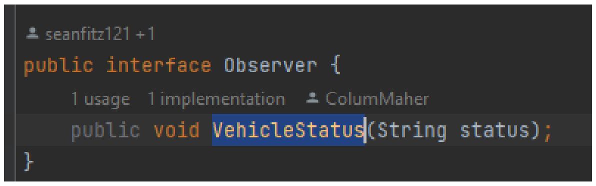
```
@PostMapping("/register")
public String registerUser(@ModelAttribute User user){

    User u = userFactory.createUser(user.getName(), user.getPassword());
    userService.saveUser(u);
    return "login";

}
```

Figure 23 Factory method call

Here we have a factory interface implemented by a simple factory class that creates a user model object which is used to save new users to our database.

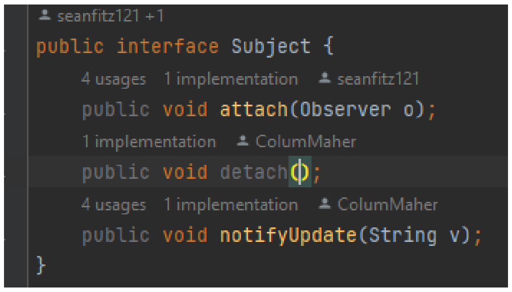
Observer Design Pattern

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. The pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It allows you to add observers without modifying the subject or other observers.



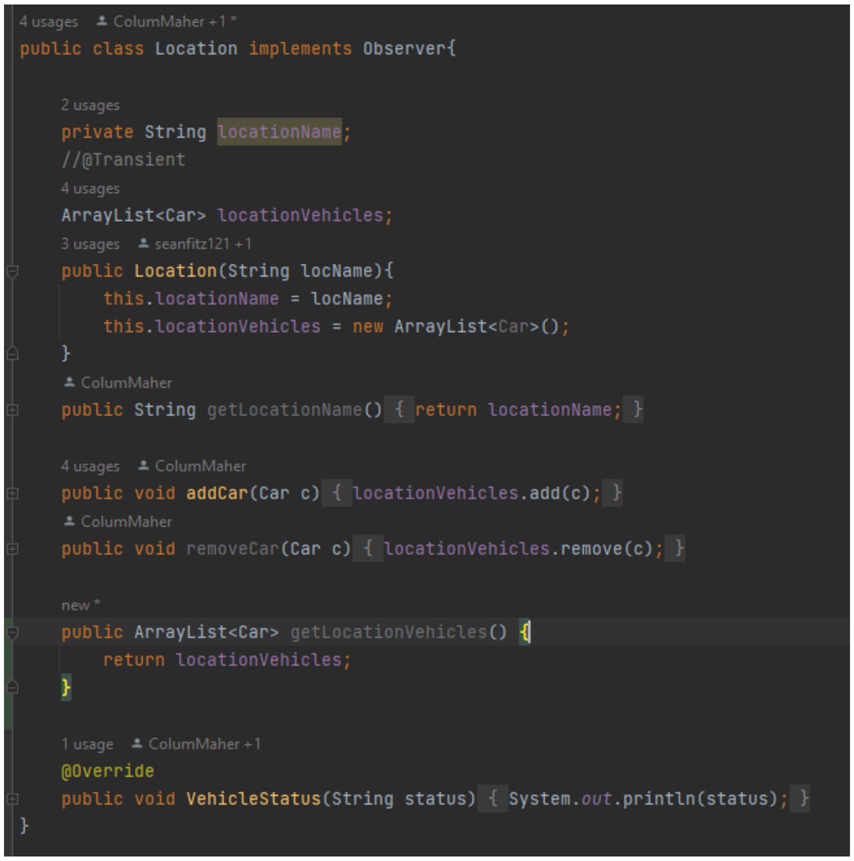
```
seanfitz121 +1
public interface Observer {
    1 usage   1 implementation   ColumMaher
    public void VehicleStatus(String status);
}
```

Figure 24 Observer Interface



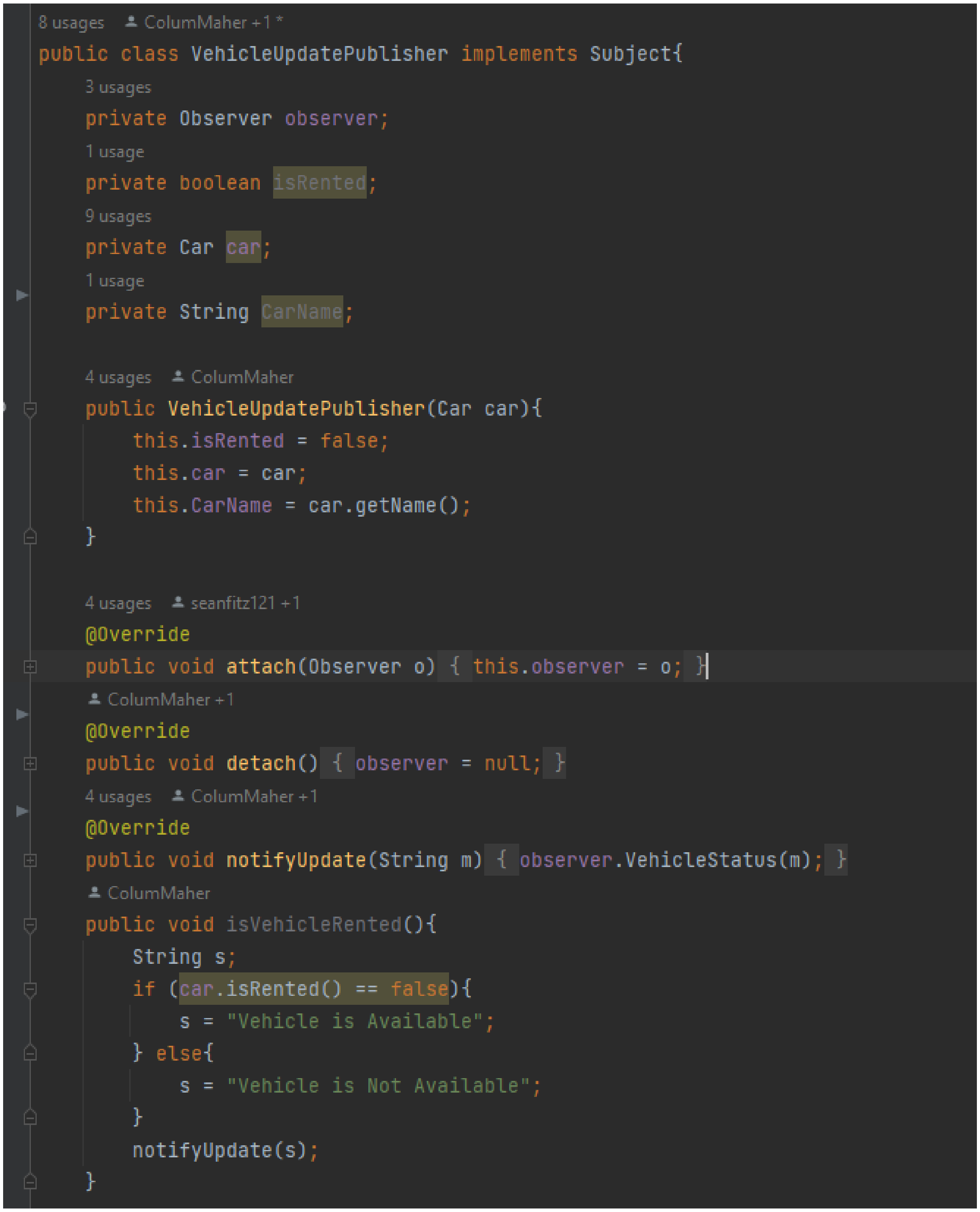
```
seanfitz121 +1
public interface Subject {
    4 usages   1 implementation   seanfitz121
    public void attach(Observer o);
    1 implementation   ColumMaher
    public void detach();
    4 usages   1 implementation   ColumMaher
    public void notifyUpdate(String v);
}
```

Figure 24 Subject Interface



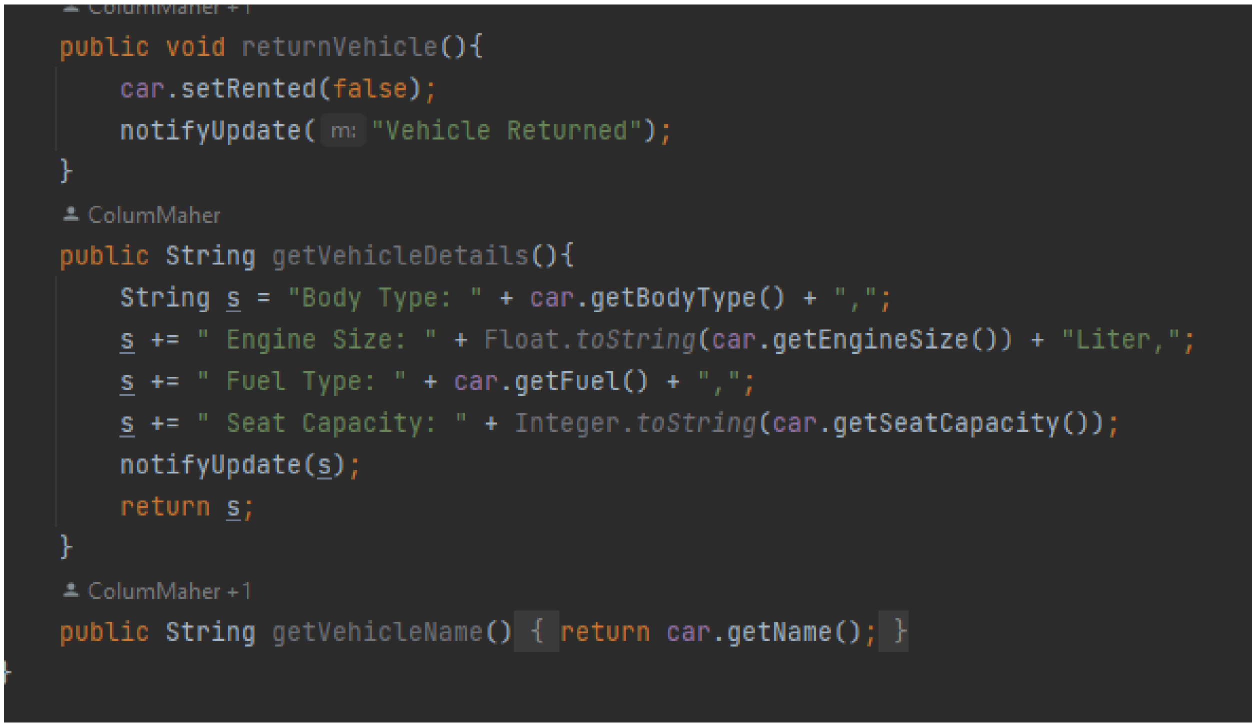
```
4 usages  ColumMaher +1 *
public class Location implements Observer{

    2 usages
    private String locationName;
    //@Transient
    4 usages
    ArrayList<Car> locationVehicles;
    3 usages  seanfitz121 +1
    public Location(String locName){
        this.locationName = locName;
        this.locationVehicles = new ArrayList<Car>();
    }
    ColumMaher
    public String getLocationName() { return locationName; }

    4 usages  ColumMaher
    public void addCar(Car c) { locationVehicles.add(c); }
    ColumMaher
    public void removeCar(Car c) { locationVehicles.remove(c); }

    new *
    public ArrayList<Car> getLocationVehicles() {
        return locationVehicles;
    }

    1 usage  ColumMaher +1
    @Override
    public void VehicleStatus(String status) { System.out.println(status); }
}
```

Figure 25 Observer Implementation



```
8 usages   ColumMaher +1 *
public class VehicleUpdatePublisher implements Subject{
    3 usages
    private Observer observer;
    1 usage
    private boolean isRented;
    9 usages
    private Car car;
    1 usage
    private String CarName;

    4 usages   ColumMaher
    public VehicleUpdatePublisher(Car car){
        this.isRented = false;
        this.car = car;
        this.CarName = car.getName();
    }

    4 usages   seanfitz121 +1
    @Override
    public void attach(Observer o) { this.observer = o; }
      ColumMaher +1
    @Override
    public void detach() { observer = null; }
    4 usages   ColumMaher +1
    @Override
    public void notifyUpdate(String m) { observer.VehicleStatus(m); }
      ColumMaher
    public void isVehicleRented(){
        String s;
        if (car.isRented() == false){
            s = "Vehicle is Available";
        } else{
            s = "Vehicle is Not Available";
        }
        notifyUpdate(s);
    }
```

Figure 26 Subject implementation



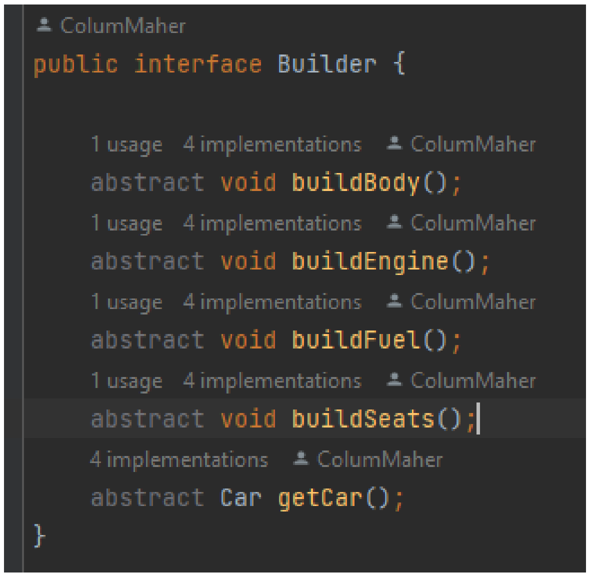
```
    public void returnVehicle(){
        car.setRented(false);
        notifyUpdate( m: "Vehicle Returned");
    }
    ColumMaher
    public String getVehicleDetails(){
        String s = "Body Type: " + car.getBodyType() + ",";
        s += " Engine Size: " + Float.toString(car.getEngineSize()) + "Liter,";
        s += " Fuel Type: " + car.getFuel() + ",";
        s += " Seat Capacity: " + Integer.toString(car.getSeatCapacity());
        notifyUpdate(s);
        return s;
    }
    ColumMaher +1
    public String getVehicleName() { return car.getName(); }
}
```

Figure 27 Subject implementation.

In our project the subject is the vehicles and the observer is the location. A location receives updates of the cars at that location when it is rented or returned. A vehicle can attach and detach observers as needed if they were rented in one location and returned to another. Location and vehicle have a one to many relationship as a vehicle can only be at one location at a time but many vehicles can be at the same location.

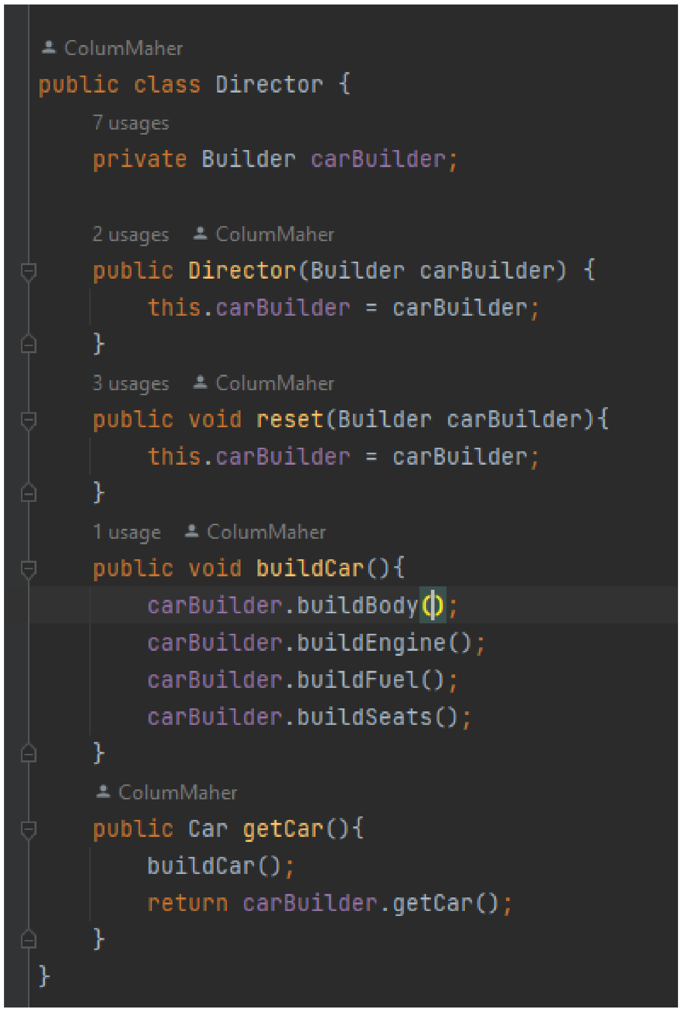
Builder Design Pattern

The Builder pattern is a creational design pattern that lets you construct complex objects. The pattern allows you to produce different types and representations of an object using the same construction code. The builder pattern also features a director, the director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.



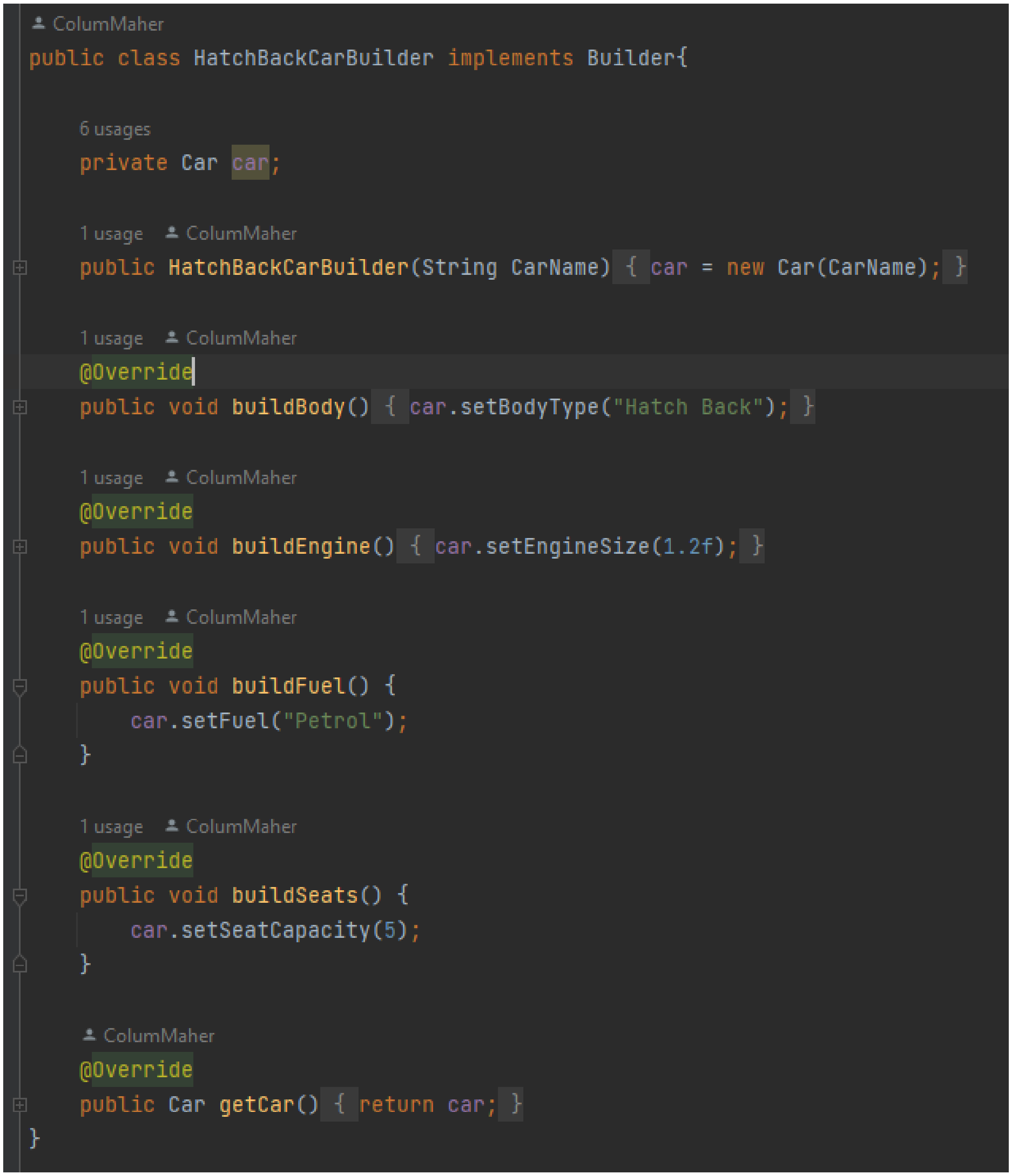
```
ColumMaher
public interface Builder {

    1 usage  4 implementations  ColumMaher
    abstract void buildBody();
    1 usage  4 implementations  ColumMaher
    abstract void buildEngine();
    1 usage  4 implementations  ColumMaher
    abstract void buildFuel();
    1 usage  4 implementations  ColumMaher
    abstract void buildSeats();
    4 implementations  ColumMaher
    abstract Car getCar();
}
```

Figure 28: Builder interface



```
ColumMaher
public class Director {
    7 usages
    private Builder carBuilder;

    2 usages  ColumMaher
    public Director(Builder carBuilder) {
        this.carBuilder = carBuilder;
    }
    3 usages  ColumMaher
    public void reset(Builder carBuilder){
        this.carBuilder = carBuilder;
    }
    1 usage  ColumMaher
    public void buildCar(){
        carBuilder.buildBody();
        carBuilder.buildEngine();
        carBuilder.buildFuel();
        carBuilder.buildSeats();
    }
    ColumMaher
    public Car getCar(){
        buildCar();
        return carBuilder.getCar();
    }
}
```

Figure 29: Director which calls Builder and its building methods



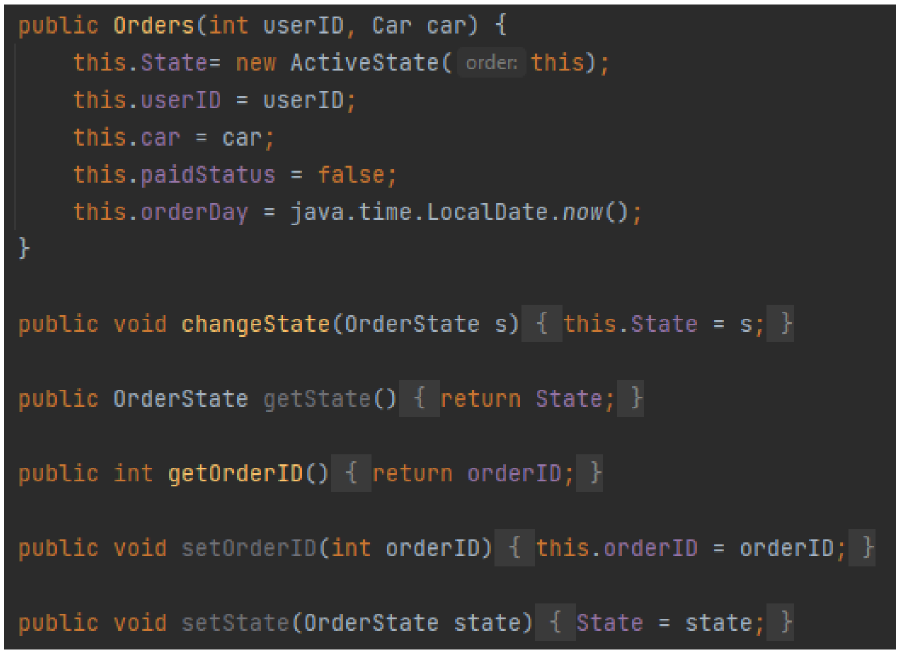
```
ColumMaher
public class HatchBackCarBuilder implements Builder{

    6 usages
    private Car car;

    1 usage   ColumMaher
    public HatchBackCarBuilder(String CarName) { car = new Car(CarName); }

    1 usage   ColumMaher
    @Override
    public void buildBody() { car.setBodyType("Hatch Back"); }

    1 usage   ColumMaher
    @Override
    public void buildEngine() { car.setEngineSize(1.2f); }

    1 usage   ColumMaher
    @Override
    public void buildFuel() {
        car.setFuel("Petrol");
    }

    1 usage   ColumMaher
    @Override
    public void buildSeats() {
        car.setSeatCapacity(5);
    }

    ColumMaher
    @Override
    public Car getCar() { return car; }
}
```

Figure 30: An example of a builder.

As the builder pattern allows you to build various representations of a product we decided to use it for the creation of vehicle objects. Using builders we can support the creation of different types of vehicles like small form hatchbacks, SUVs and sports cars.

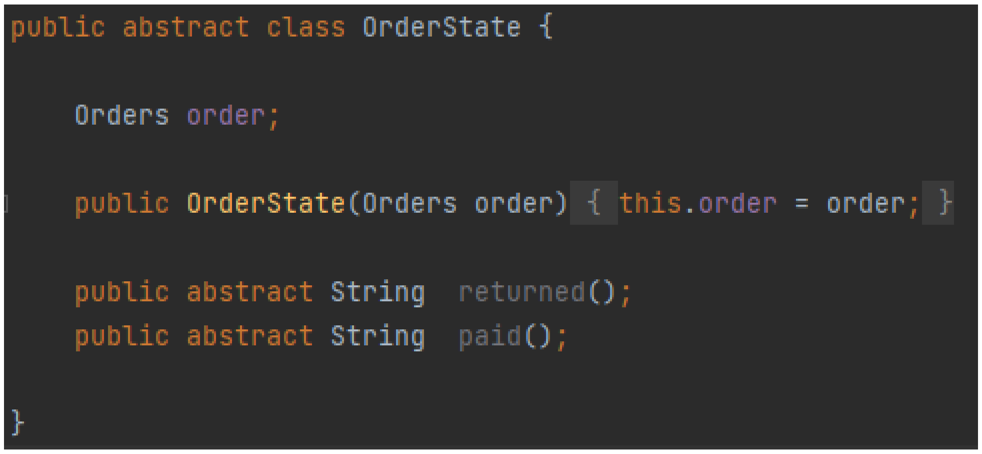
State Design Pattern

The State Design Pattern extracts state-related behaviors into separate state classes and forces the original object to delegate the work to an instance of these classes, instead of acting on its own. We planned to use the design pattern to set the different states of an order.



```java
public Orders(int userID, Car car) {
    this.State= new ActiveState( order: this);
    this.userID = userID;
    this.car = car;
    this.paidStatus = false;
    this.orderDay = java.time.LocalDate.now();
}

public void changeState(OrderState s) { this.State = s; }

public OrderState getState() { return State; }

public int getOrderID() { return orderID; }

public void setOrderID(int orderID) { this.orderID = orderID; }

public void setState(OrderState state) { State = state; }
```

Figure 31: Order page



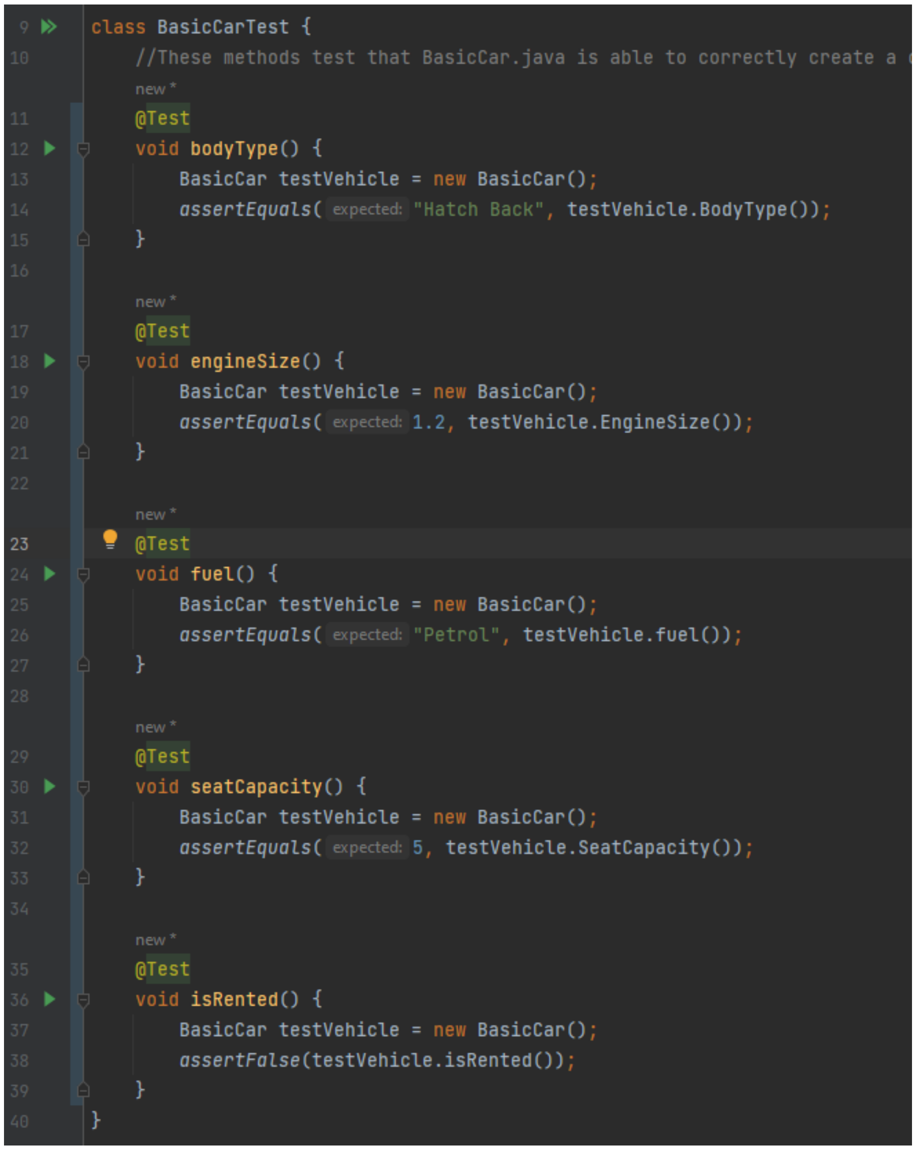
```java
public abstract class OrderState {

    Orders order;

    public OrderState(Orders order) { this.order = order; }

    public abstract String  returned();
    public abstract String  paid();

}
```

Figure 32: OrderState class

We used an ActiveState page along with a Paid and Returned. The idea was to set the state of an order to active when a customer selects their car for rental which would act as a default state. This was to be followed with either returned or paid depending on the customer's actions. Unfortunately we did not have enough time for the implementation of the design pattern.

Testing

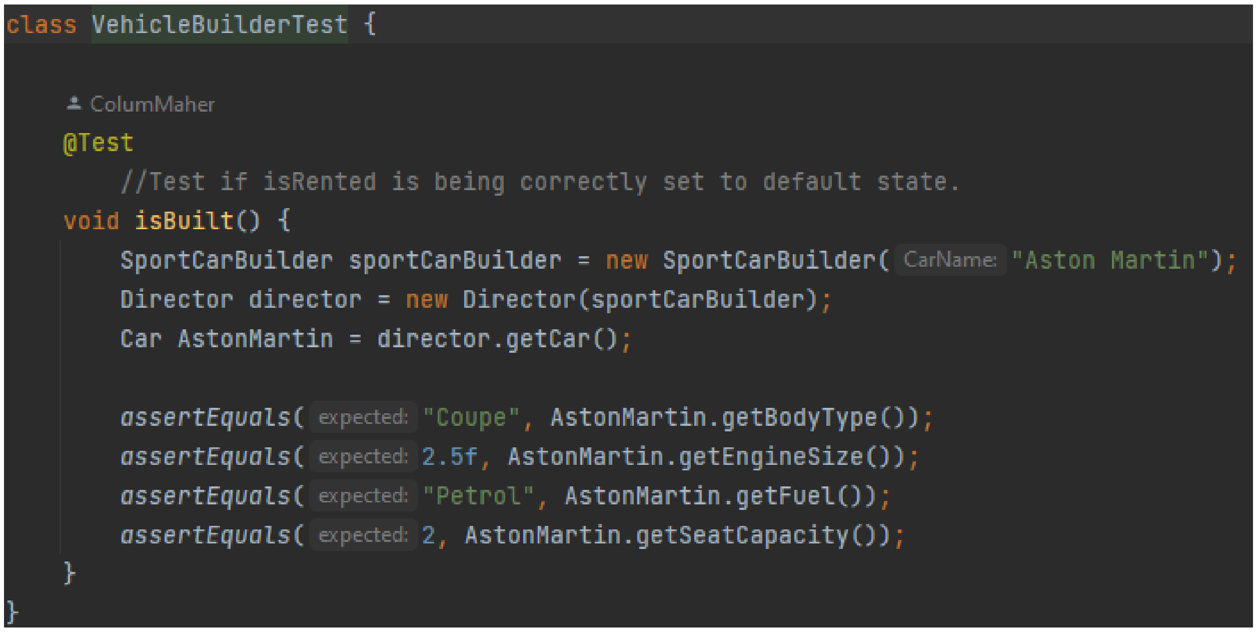
In order to make sure that our code was correctly creating vehicles we used intelliJ’s JUnit integration to generate tests from our code that checks the values assigned by the sample model. While in the end, we did end up changing our approach to categorizing vehicles, thus removing the sample vehicle class BasicCar.java, it did help us when creating the vehicles.



```
class BasicCarTest {
    //These methods test that BasicCar.java is able to correctly create a
    new *
    @Test
    void bodyType() {
        BasicCar testVehicle = new BasicCar();
        assertEquals( expected: "Hatch Back", testVehicle.BodyType());
    }

    new *
    @Test
    void engineSize() {
        BasicCar testVehicle = new BasicCar();
        assertEquals( expected: 1.2, testVehicle.EngineSize());
    }

    new *
    @Test
    void fuel() {
        BasicCar testVehicle = new BasicCar();
        assertEquals( expected: "Petrol", testVehicle.fuel());
    }

    new *
    @Test
    void seatCapacity() {
        BasicCar testVehicle = new BasicCar();
        assertEquals( expected: 5, testVehicle.SeatCapacity());
    }

    new *
    @Test
    void isRented() {
        BasicCar testVehicle = new BasicCar();
        assertFalse(testVehicle.isRented());
    }
}
```

Figure 33: Assignment testing for BasicCar.java

Once we had changed our approach to creating the vehicles, we rewrote our tests to confirm that it was correctly assigning the values.

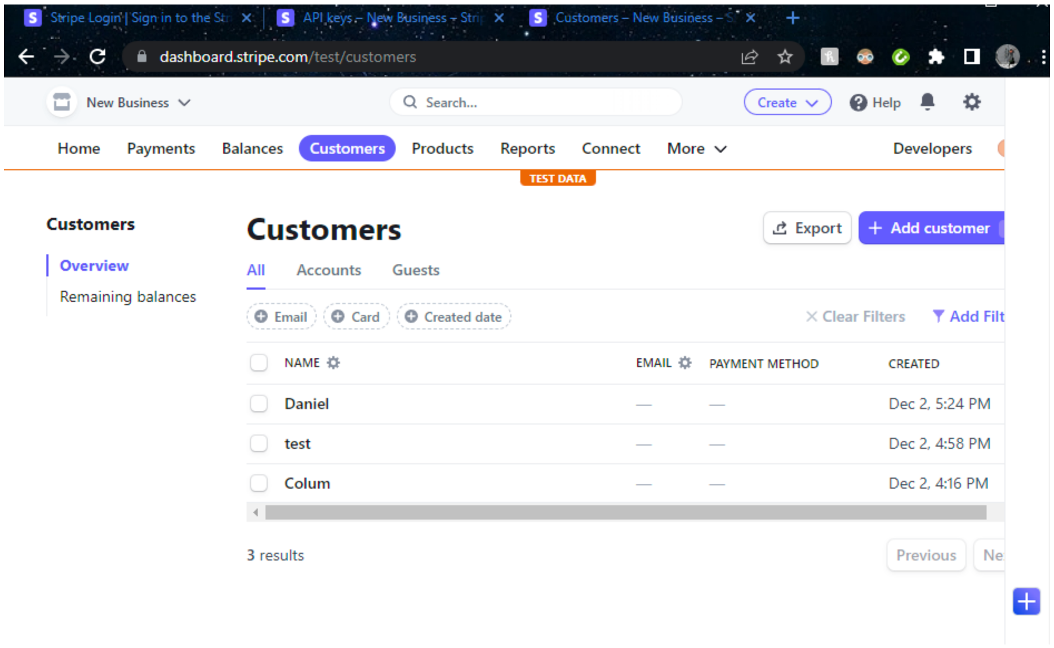

```
class VehicleBuilderTest {

    ColumMaher
    @Test
        //Test if isRented is being correctly set to default state.
    void isBuilt() {
        SportCarBuilder sportCarBuilder = new SportCarBuilder( CarName: "Aston Martin");
        Director director = new Director(sportCarBuilder);
        Car AstonMartin = director.getCar();

        assertEquals( expected: "Coupe", AstonMartin.getBodyType());
        assertEquals( expected: 2.5f, AstonMartin.getEngineSize());
        assertEquals( expected: "Petrol", AstonMartin.getFuel());
        assertEquals( expected: 2, AstonMartin.getSeatCapacity());
    }
}
```
Figure 34: More testing

Added Value

Stripe

Stripe is an easy to use suite of APIs powering online payment processing and commerce solutions for internet businesses. It lets merchants accept credit and debit cards or other payments. It is used by some of the world’s largest companies, including Amazon and Shopify.

Stripe is very simple to implement. We just had to implement dependencies in intellij and then go to the Stripe website to receive the API key and we had it in the project. The documentation on the website is very helpful and simple to understand.

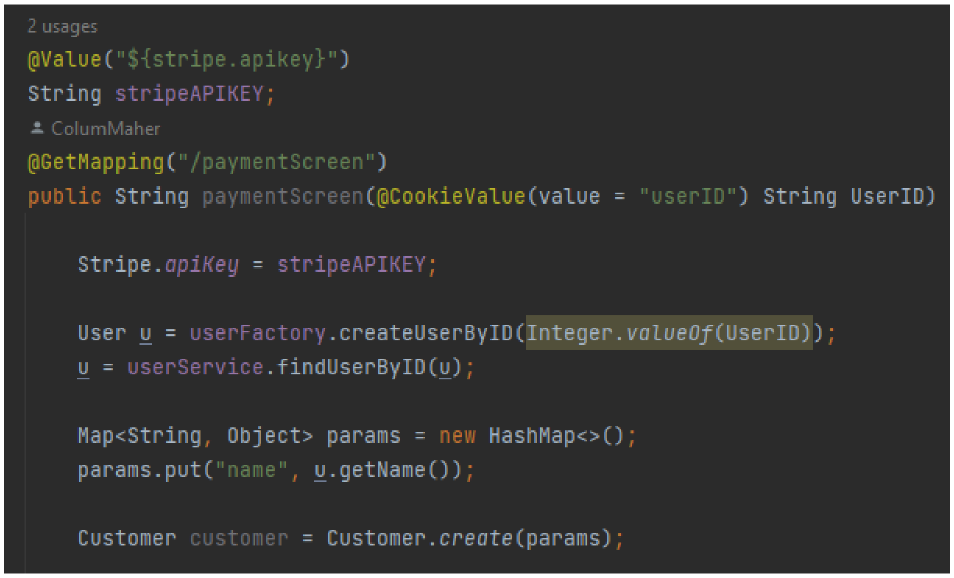


Customers

NAME	EMAIL	PAYMENT METHOD	CREATED
Daniel	—	—	Dec 2, 5:24 PM
test	—	—	Dec 2, 4:58 PM
Colum	—	—	Dec 2, 4:16 PM

3 results

Figure 35: Customers in Stripe dashboard



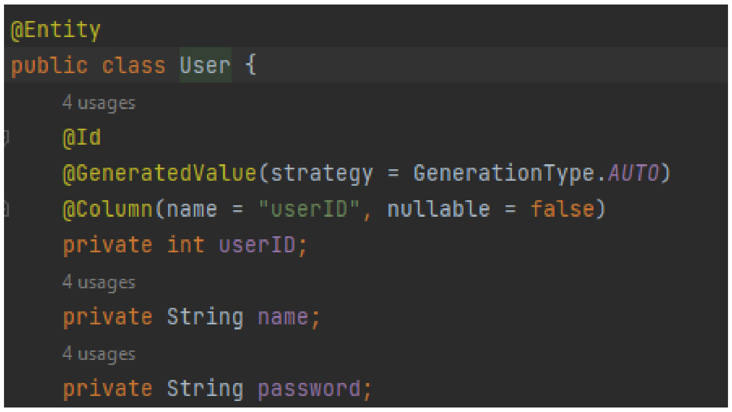
```
2 usages
@Value("${stripe.apikey}")
String stripeAPIKEY;
ColumMaher
@GetMapping("/paymentScreen")
public String paymentScreen(@CookieValue(value = "userID") String UserID)

    Stripe.apiKey = stripeAPIKEY;

    User u = userFactory.createUserByID(Integer.valueOf(UserID));
    u = userService.findUserByID(u);

    Map<String, Object> params = new HashMap<>();
    params.put("name", u.getName());

    Customer customer = Customer.create(params);
```

Figure 36: Implementing the API, Customer is part of the stripe API

Object Relational Mapping (ORM)

Object Relational Mapping (ORM) is a method of creating a link between object-oriented systems and a relational database. Spring-ORM is a design pattern which can be utilized to access a relational database from Java, an object-oriented language. This enables persistence in our program. The API we used for this is the Java Persistence API (JPA). This API links our system to our relational MySQL database.

We used JPA in order to create user, car and order objects and map them to our database. In doing this, we created persistent data and had the ability to store and access this data efficiently. An example of the use of JPA persistence is seen below in Figure 37

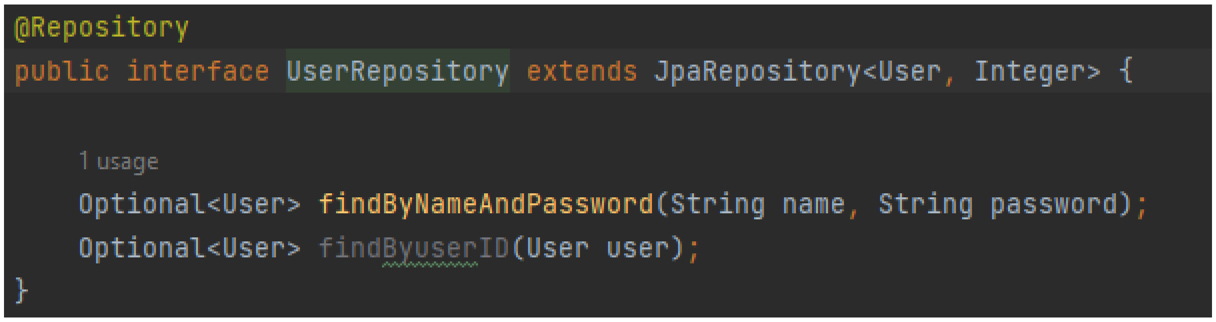


```
@Entity
public class User {
    4 usages
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "userID", nullable = false)
    private int userID;
    4 usages
    private String name;
    4 usages
    private String password;
```

Figure 37

In Figure 37, we mark the user class as an Entity using '@Entity', and this will allocate fields for each variable we declare in the user class. Fields can be ignored using '@Transient', as seen in the Order class. '@Id', '@GeneratedValue', and '@Column' are annotations used to specify the structure of our fields and table allocation for userID. It will auto generate an ID for userID. '@Id' will specify our primary key, '@GeneratedValue' will declare how the key is generated.

We also use elements of JPA for creating repositories. A repository is a data access object (DAO). We see an example of this in Figure 38 below.



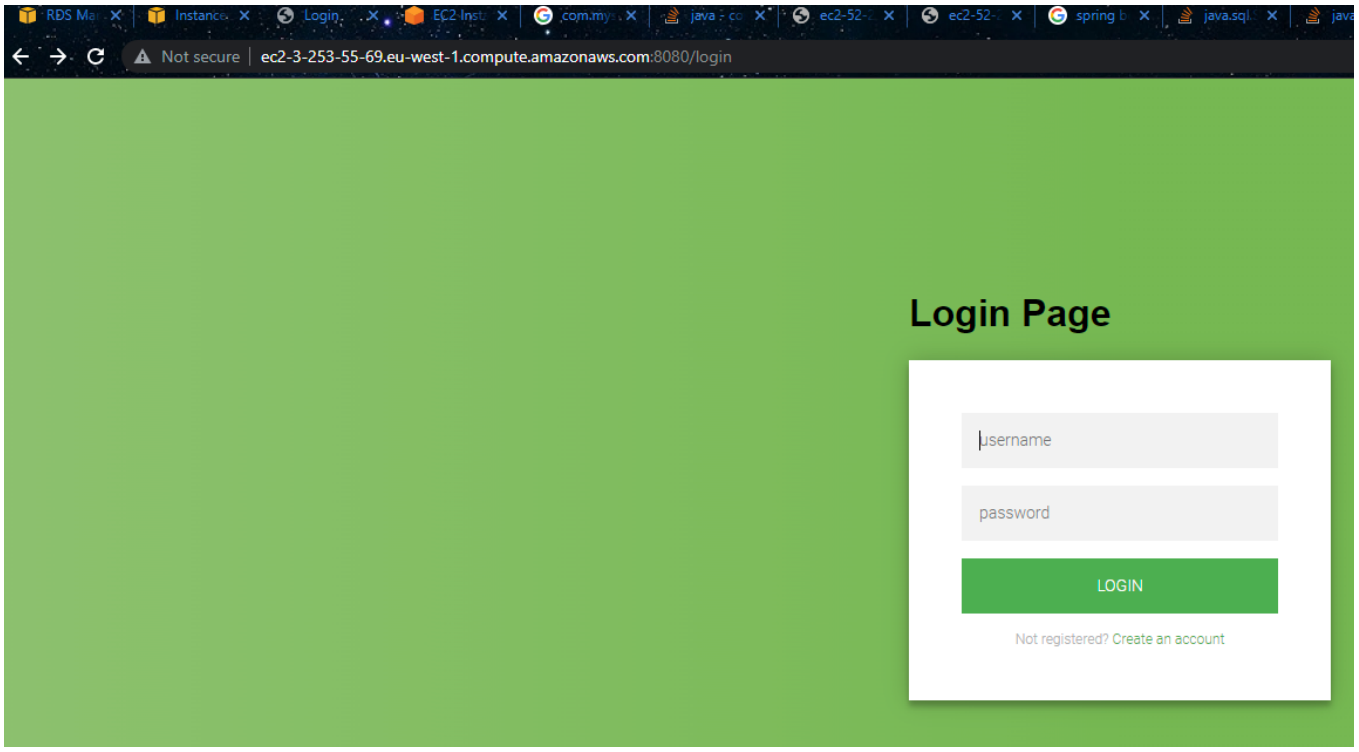
```
@Repository
public interface UserRepository extends JpaRepository<User, Integer> {

    1 usage
    Optional<User> findByNameAndPassword(String name, String password);
    Optional<User> findByuserID(User user);
}
```

Figure 38

Here we create a UserRepository of User objects, and use this in our User Service class 'UserService.java'.

Amazon Web Services

In order to host our web app we used Amazon Web Services EC2 and RDS services. As our project took advantage of object relational mapping through the Java persistence API we used RDS running a SQL database for our project. Our EC2 instance was run on an Amazon Machine Image running linux, this allowed us to easily deploy our service. Thankfully, AWS allows us to easily connect our application with the database through the RDS settings and we just had to specify a datasource and credentials in our application.properties file. To run our app we SSH in the AWS console and upload our executable jar file through WinSCP using SSH file transfer protocol.



Not secure | ec2-3-253-55-69.eu-west-1.compute.amazonaws.com:8080/login

Login Page

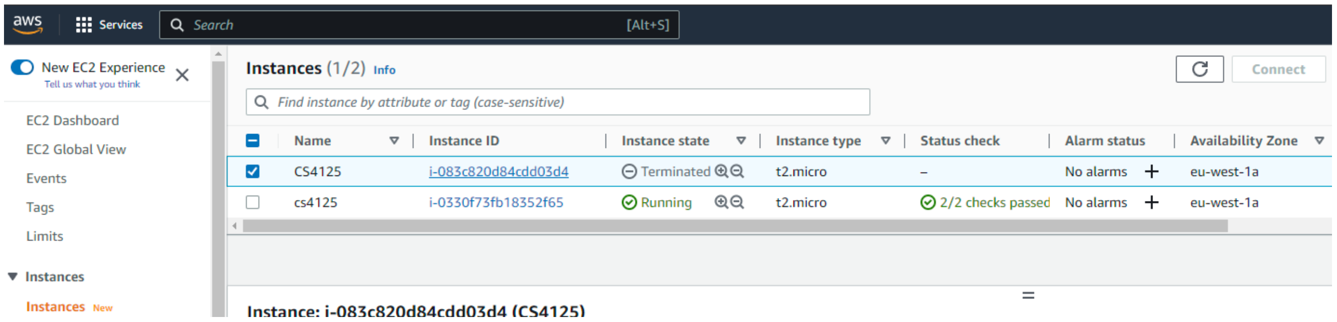
username

password

LOGIN

Not registered? Create an account

Figure 39. Application running on AWS, looking at Login Page



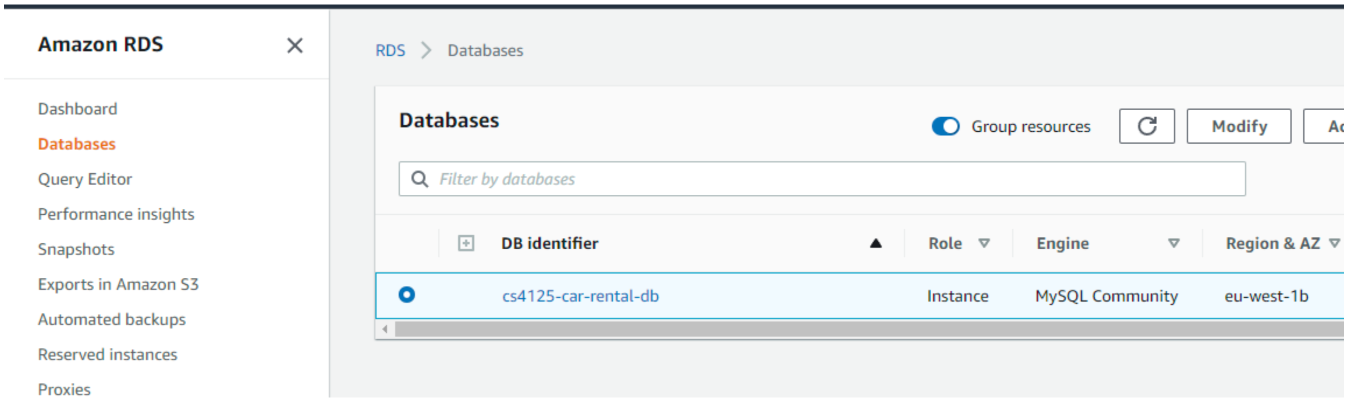
Instances (1/2) Info

Find instance by attribute or tag (case-sensitive)

	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
✓	CS4125	i-083c820d84cdd03d4	Terminated	t2.micro	–	No alarms	eu-west-1a
	cs4125	i-0330f73fb18352f65	Running	t2.micro	2/2 checks passed	No alarms	eu-west-1a

Instance: i-083c820d84cdd03d4 (CS4125)

Figure 40. EC2 Instances

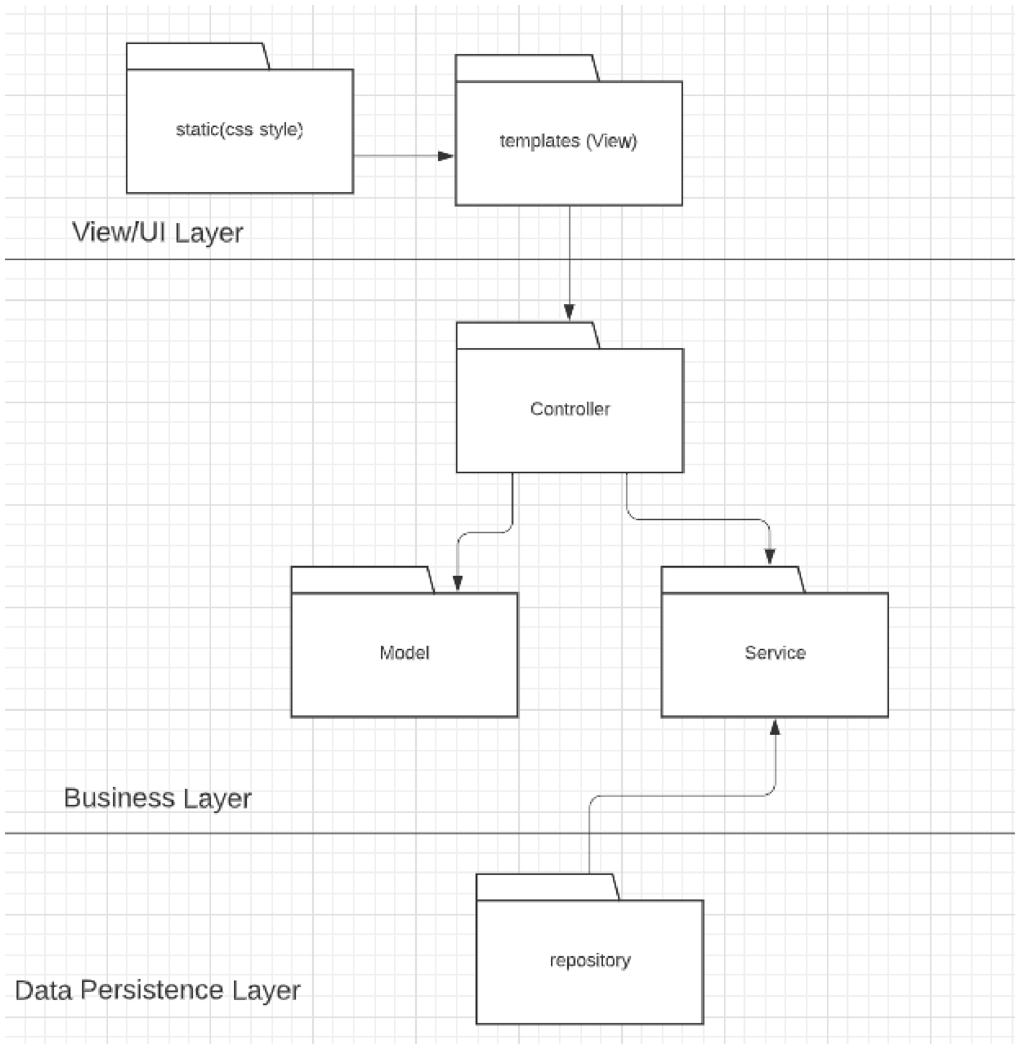


DB identifier	Role	Engine	Region & AZ
cs4125-car-rental-db	Instance	MySQL Community	eu-west-1b

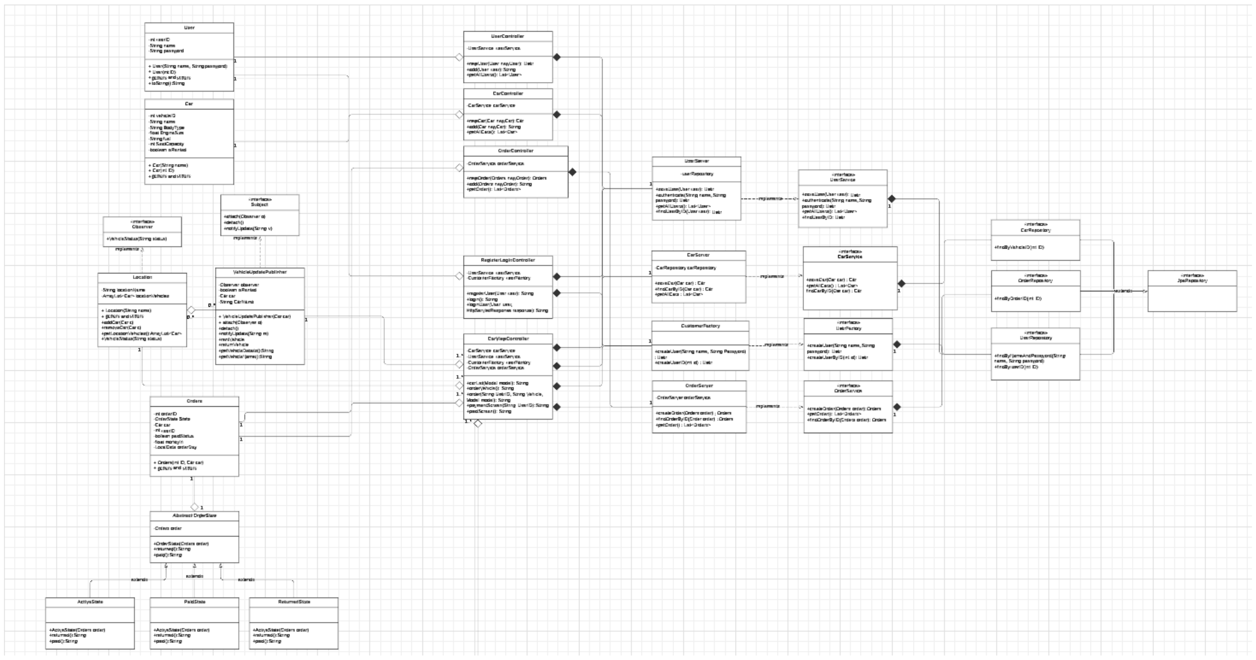
Figure 41. AWS SQL Relational Database

Recovered architecture and design blueprints

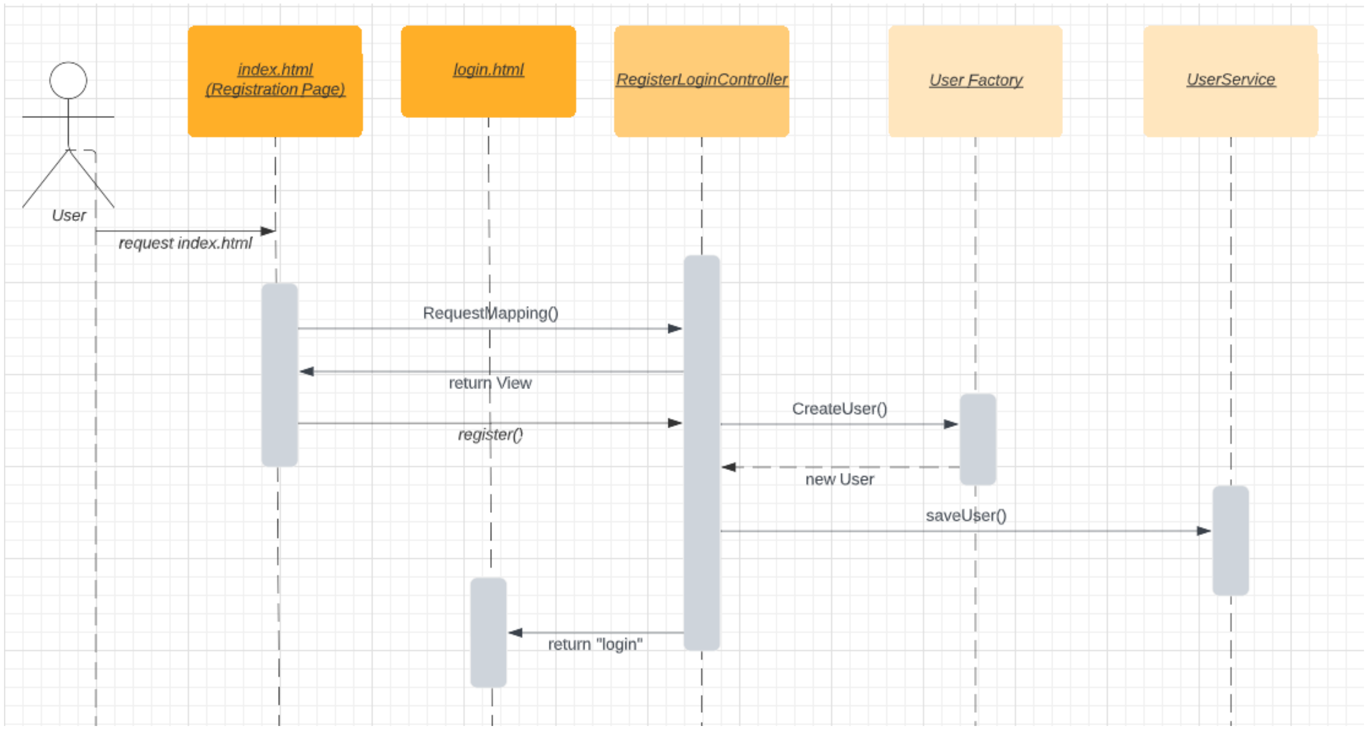
Design Time Package Diagram



Design Time Class Diagram



Design Time Sequence Diagram



Evaluation

Looking back at the initial design of the project, and also our current implementation, there was some merit to our design but it should have been fleshed out further. In regards to the utility of diagrams and UML for an initial design, it was difficult to follow up with an implementation as the diagrams don’t reflect the complexities or requirements of adapted tools/frameworks and technologies (i.e. Spring Boot, AWS etc).

During the early stages of the project we found some benefit of the UML language to communicate abstract ideas to each other while the project was still quite small in scale. As the scale of the design increased the efficacy of UML models and diagrams deteriorated significantly. This led to a significant gap between our diagrams pre-implementation versus post implementation.

We unfortunately encountered some difficulties navigating the Spring Boot framework, which overall affected the quality of our implementation, and hampered our development speed. Due to time constraints we were unable to realize our design fully and some goals for this assignment were missed.

References

Wrike.com. (2022). *The Agile Software Development Life Cycle | Wrike Agile Guide*. [online] Available at: https://www.wrike.com/agile-guide/agile-development-life-cycle/ [Accessed 07 Nov. 2022].

Refactoring.guru. (2014). *Refactoring and Design Patterns*. [online] Available at: https://refactoring.guru/ [Accessed 15 Nov. 2022].

Gamma, E., Helm, R., Johnson, R., Johnson, R.E. and Vlissides, J., 1995. Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH.

Stripe.com. (2022). *Documentation*. [online] Available at: https://stripe.com/docs [Accessed 1 Dec. 2022].