Course/Programme:	MSc. Artificial Intelligence
Module Name and Code:	Advanced Software Development (SWE7302)
Student ID:	2011184
Tutor:	Muhammad Saeedi
Assessment Number:	2 of 2
Assessment Type:	Portfolio
Assessment Title:	A project that delivers a fully functioning and documented solution to a given problem.
Indicative Word Count:	3000 to 3500 words.
Weighting:	60% of overall module grade.
Submission Deadline:	12.05.2025 (no later than 23:59)
Submission Date:	12.05.2025
Learning Outcomes Assessed:	**LO1:** Critically apply object-oriented analysis, design and programming ideas and critically evaluate object-oriented solutions. **LO3:** Critically analyse and justify the application of design patterns in the development of software.

Designing and Crafting a Layered Vehicle Rental System Through Object-Oriented Programming and Design Patterns.

Abstract

This report documents the design, implementation and critical evaluation of a three-tier Vehicle Rental System. The solution applies object-oriented principles such as encapsulation, inheritance, and polymorphism. This is achieved within a layered architecture that separates a command-line interface, service façade and JDBC Data-Access Objects. Key design patters that are used are DAO, Service/Façade, GRASP Controller. The design is planned and represented through UML artefacts. Finally, the code is tested through a multi-level automated approach (unit testing and integration testing) and a manual exploration and testing of some of the main workflows. Results confirm functional aptness, maintainability scalability of the solution.

Keywords: Software Development, Design Patterns, UML, Vehicle Rental System

Contents

Table of Figures

Table of Tables

1. Introduction

The fast-paced transition to digital products and digitalisation of services has altered the way in which industries operate, including the transportation and mobility sectors. Companies that are dealing with vehicle rentals were primarily dealing with their day-to-day work through manual labour and documenting paperwork with heavy human supervision involved. Nowadays, such tasks are increasingly managed through software applications which deliver real-time data access, automation, and improved customer experiences. The current project delivers a working software solution and showcases the practical implementation of object-oriented design principles and patterns, software architectural modelling, and database integration in a real-world context.

1.1. Problem Statement

The current project revolves around the needs of a small vehicle renting business whose daily activities gather around dealing with the difficulties related to booking availability, overlapping schedules and bookings, payment tracking (as payments are taken from the customer upon vehicle pickup, separate from the booking), and managing the roles of users involved. Traditional “manual” systems of handling such tasks frequently stall or experience issues, thus their effectiveness is not very scalable and secure leading to reduced business productivity, errors and customer disappointment. Furthermore, ensuring data integrity and providing role-based access control to different actors are essential challenges in multi-user applications and company activities. The need for a flexible, secure, and extensible rental management system forms the basis of this project.

1.2. Project Aims and Objectives

The main goal of this project is to design and develop a fully functional **Vehicle Rental System** in Java that follows object-oriented programming (OOP) principles and underpins

persistent data management utilising a SQL-based database. The key objectives of the system comprise of:

- Clearly outlining different roles and responsibilities of users.
- Allowing them to register, log into the system, and manage vehicle rentals.
- Ensuring the ability of administrators to supervise vehicle inventories and user accounts.
- Guaranteeing safe, time-based booking and rental conflict prevention.
- Superintending rental history and payment status.
- Implementing notification features for customer and admin accounts for overdue or upcoming returns.
- Employing OOP principles (such as inheritance, encapsulation, and polymorphism) and design patterns for maintainability and scalability of the solution.

1.3. Scope and Features

The system incorporates core features like:

- Role-based user access (Admin/User).
- Full CRUD operations for users and vehicles.
- Date/time-based vehicle bookings.
- SQL database and persistence through Java Database Connectivity (JDBC).
- Notifications for expected returns of company assets.
- Payment tracking and confirmation.

The application's architecture has been drafted as a modular one, splitting data access logic (DAO pattern) from business logic, hence uplifting easier testing and maintainability.

1.4. Structure of the Report

This report is composed of several primary sections. Section 2 discusses the planned architecture of the application, its development methodology, object-oriented principles involved, and the employed design patterns. All of them are pictured and explained through different UML and ER diagrams. The third section of the work summarises the conclusions drawn and outlines potential future enhancements. Section 4 exemplifies the personal learning reflections expressed by the author of the current work.

2. Software/Product Design and Development

2.1. System Architecture Overview

The Vehicle Rental System has been outlined by employing a modular three-tier architecture. It encompasses a presentation layer, business logic layer, and persistence layer. The choice of such an approach is grounded in the concept of **segregation of issues**, which uplifts maintainability and testability of solutions. Through such separation the application achieves both flexibility and scalability (Laplante and Kassab, 2022).

The **presentation layer** is implemented in the MainMenu.java class, that is a straightforward command-line interface. This implementation delivers context-sensitive menus for both Admin and User roles and communicates with the system through service classes, not directly with the data layer. Such an approach segregates user interactions from application logic, thus clinging to the **Model-View-Controller (MVC)** paradigm. Although, the way it is presented in the current project is simplified (Fowler *et al.*, 2002).

The **business logic layer** encapsulates the core functionalities of the system. Classes like VehicleManager, NotificationService, and AuthService encapsulate domain logic like the guarantee that there would be no overlapping of rentals, user authentication, and tracking whether the booking is paid for upon pickup (the scope of the system excludes internal payment processing in favour of “On Pick Up” payments outside of the bookings as per problem statement). This layer is where OOP principles (such as **encapsulation and responsibility-driven design**) are utilised. For example, VehicleManager is the class that primarily organises the rental operations, hence complying with the Single Responsibility Principle (SRP) from the SOLID design principles (Gast, 2015; Noback, 2018).

The **persistence layer** comprises of DAO (Data Access Object) classes – VehicleDAO, UserDAO, and RentalDAO. They deal with the “correspondence” between the SQL database (MySQL used as DBMS) through JDBC. The employment of the DAO pattern is essential for the segregation of the application from logic that is intended for the database. Such an

implementation enhances the solution's portability and testability. For instance, if the DBMS option of choice is being swapped (MySQL to PosgreSQL), then only the latter classes should be modified accordingly (Gamma *et al.*, 2009).

A generalised depiction of this architecture is exemplified in **Figure 1**, where can be observed the interplay among layers and their dependencies.

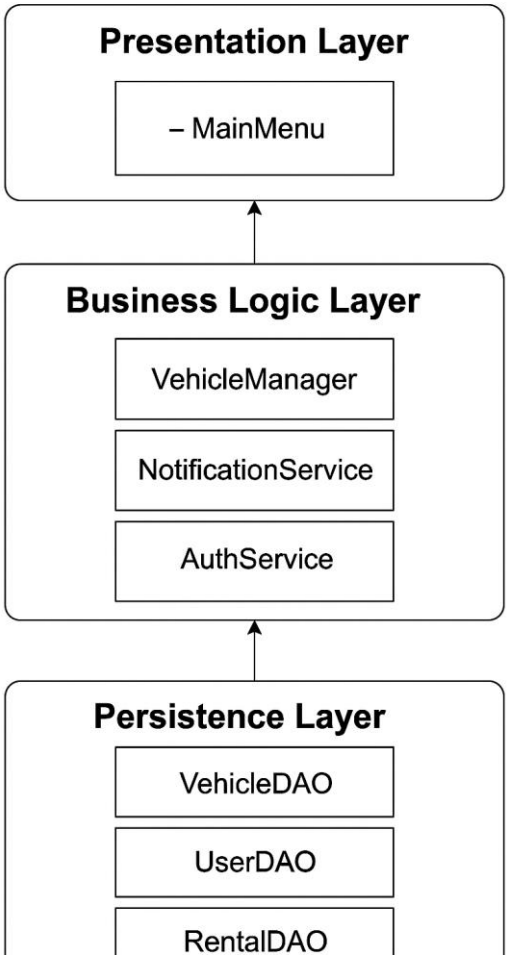

Figure 1 - Simplified View of the Interactions and Dependencies in the Application's Architecture.

The implementation of a storeyed design is assumed due to its proven effectiveness in the industry to resolve isolation, security, and scalability risks. On the other hand, if a unified design was followed, this could have created the opportunity of issues waterfalling through different modules (for example adjustments in UI may impact data management). Hence, the

latter approach minimises coupling. This is of utmost importance due to the fact that the app would superintend various user roles and interplay with persistent data (Bass *et al.*, 2015).

2.2. Key Functionalities Implemented

The Vehicle Rental System delivers a coherent set of capabilities that reflect the functional requirements that have been laid out in the problem domain. Each feature has bee carefully thought of and designed to satisfy a specific business need, which is presented in the problem statement or derived from it as a needed functionality considering the business needs and operations (Sommerville, 2016).

There are several functional areas where the main capabilities of the system are focusing on. Starting with the **User authentication and registration,** the AuthService module is designed and implemented to has the user credentials in memory and then persist them through the UserDAO to the SQL database. The process encompasses the full contact details of each user (admin or customer), as it is assumed the company may need to contact users to resolve queries, if such would arise. The current project is attempting to integrate a role-based access control (RBAC) which is a well-known industry standard for systems and applications that need to be operated by multiple actors with different roles. The employment of RBAC warrantied that the menus of the presentation layer can be rendered during run time, thus minimising code repetition. Keeping the login details on the server side, rather than hardcoding them like plain text into the application adheres to industry standards regarding the system's confidentiality (Hu and Freidman, 2014; International Organisation for Standardisation, 2022).

The **Vehicle CRUD Management** supports the ability of admin operators to add, update and delete vehicles, where the duplication of plate numbers (which serve as IDs) is evaded by using the UNIQUE constraint at Database level. It is also ensured by a pre-input verification in he VehicleManager module. Utilising such a protective programming approach in the Java

code, combined with constrains in the database, creates a safe strategy, which diminishes the business layer complexity and error-checking in the DB (Larman, 2005).

The rentals have also been designed and implemented to **reflect date and time**. They utilise the LocalDateTime function to identify conflicts through revisioning of the active bookings for a specific vehicle and ensure that there would not be any overlapping booking events. The functionality also provides a total price for the rental which is calculated by multiplying the specific price per day rental for each vehicle by the amount of time which it will be rented for. The use of LocalDateTime has been implemented to ensure that there would be no collision, if the booking is made from a different time zone (Bloch, 2018; Gamma *et al.*, 2009).

A **payment tracking and confirmation** feature is also thought of and implemented, as it has been discussed earlier that the payment processing will happen out of the scope of the system and upon vehicle pick up from the company's rental location. The application is designed and implemented to default the payment status to "false", when the booking is being made. This is so to ensure that upon pickup the admin can confirm that the payment is processed. Hence, in such a manner the business' operations are properly mirrored into the system logic. Such an approach diminishes payment frauds and allows for future integration of third-party payment API's, if the business would require it (Laudon and Traver, 2022).

The **Notifications** functionality has also been introduced to reflect overdue returns (for Admins) and upcoming bookings and returns (for customers). The NotificationService runs on demand, querying active rentals and issuing relevant messages in the role-specific menus of each system actor. Hence, such real-time feedback uplifts usability.Pull-based generation was selected over push scheduling to keep the CLI stateless and simplify concurrency management (Nielsen, 1992; Hunt and Thomas, 1999).

The application also allows for users to **view past/current rentals**, which can be **cancelled** if the start time is not already past the current moment of checking. This is enforced through the VehicleManager.cancelRental() method. In such a manner, the functionality supports the

needs of the business and customers, by providing means of maintaining user authority, while guaranteeing the cancellation would not happen after the booking start time (International Standardisation Organisation, 2023).

The system also allows **Admin users to manage the current users** lists (consisting of both Admin and Customer level). Hence, operations such as view, update, delete or add users are permitted for the Admins through the MainMenu.

2.3. UML Diagrams and Design Artifacts.

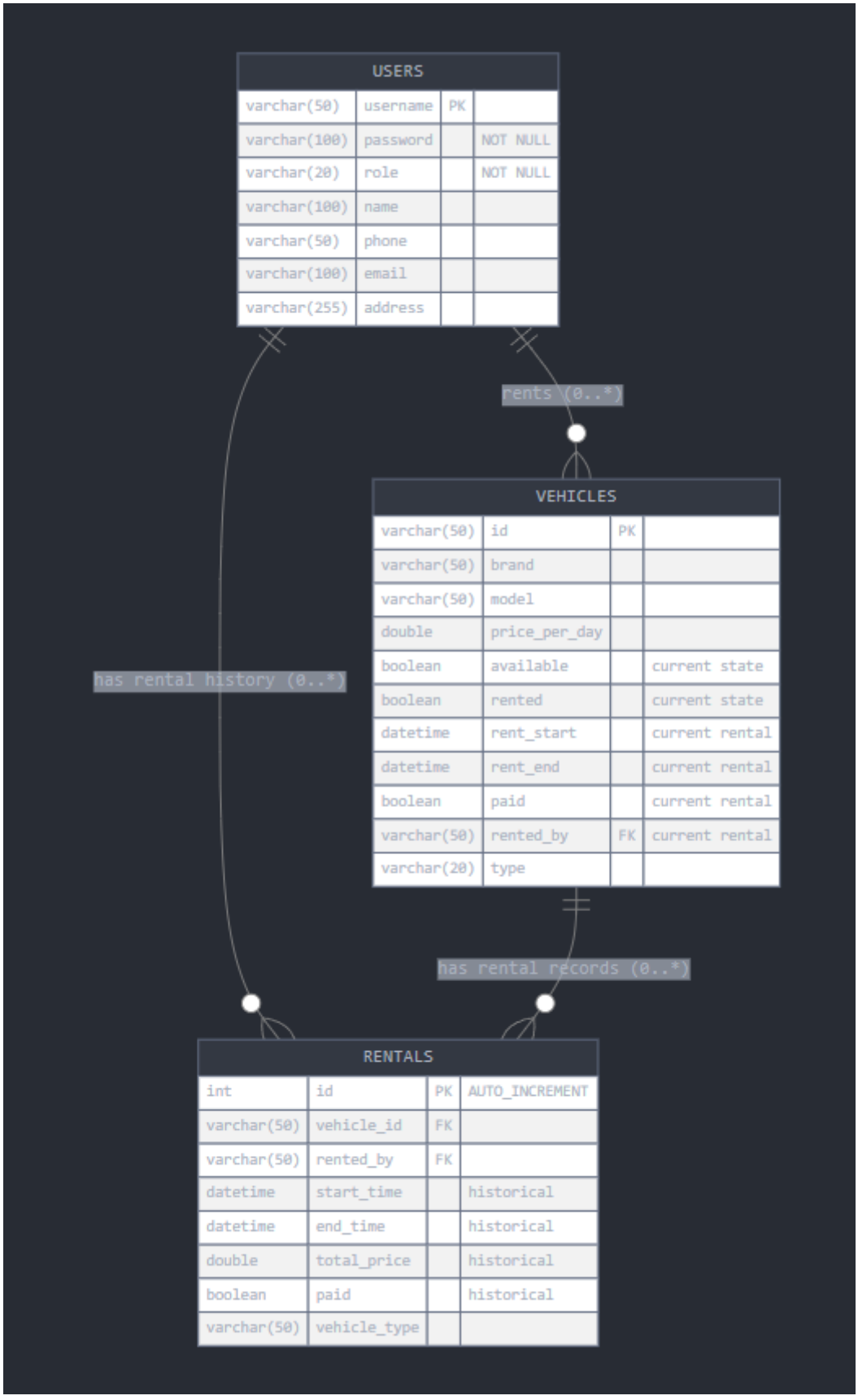

Figure 2 - ERD Diagram of the SQL Database.

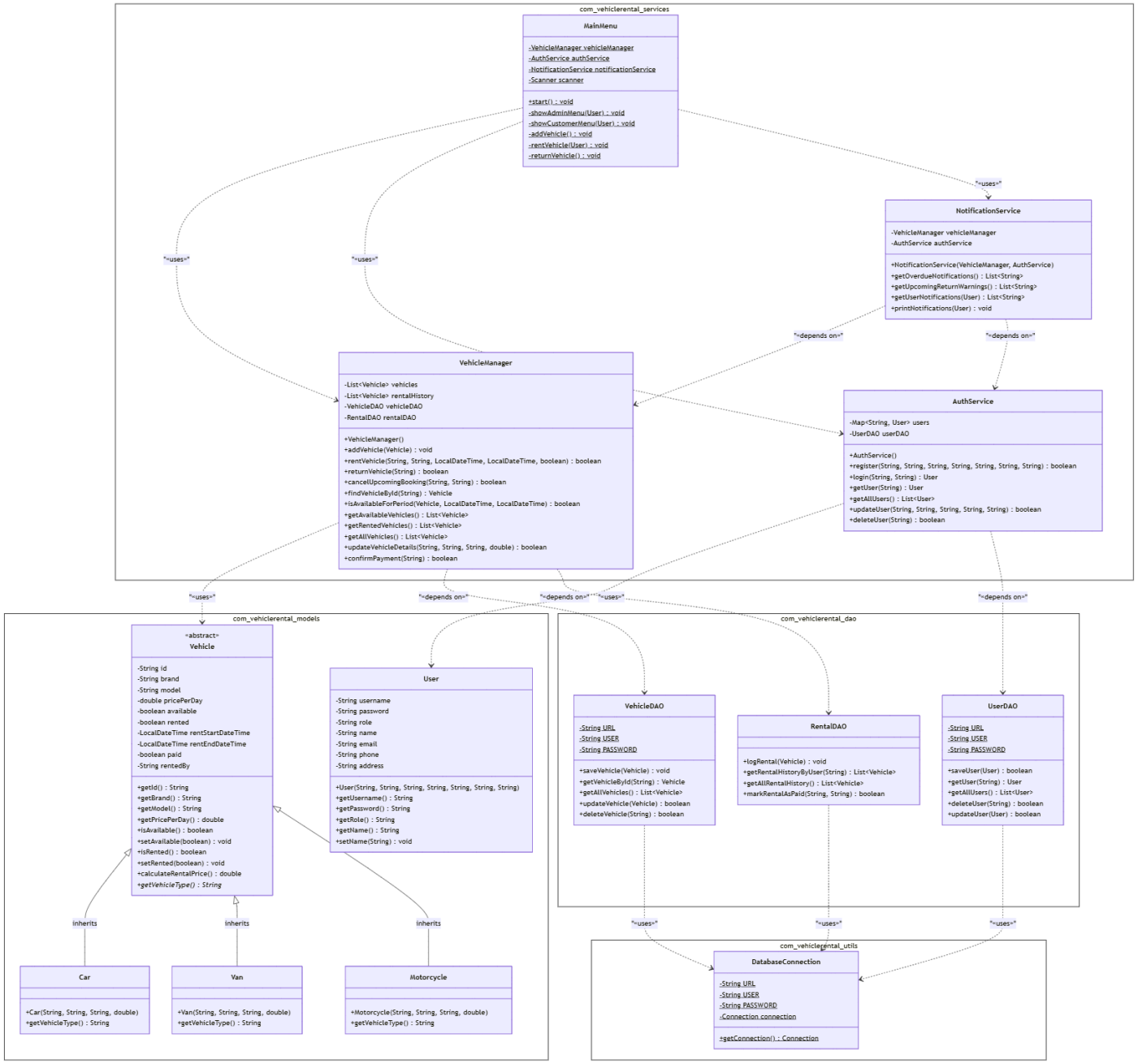

Figure 3 - Vehicle Rental App Class Diagram 1

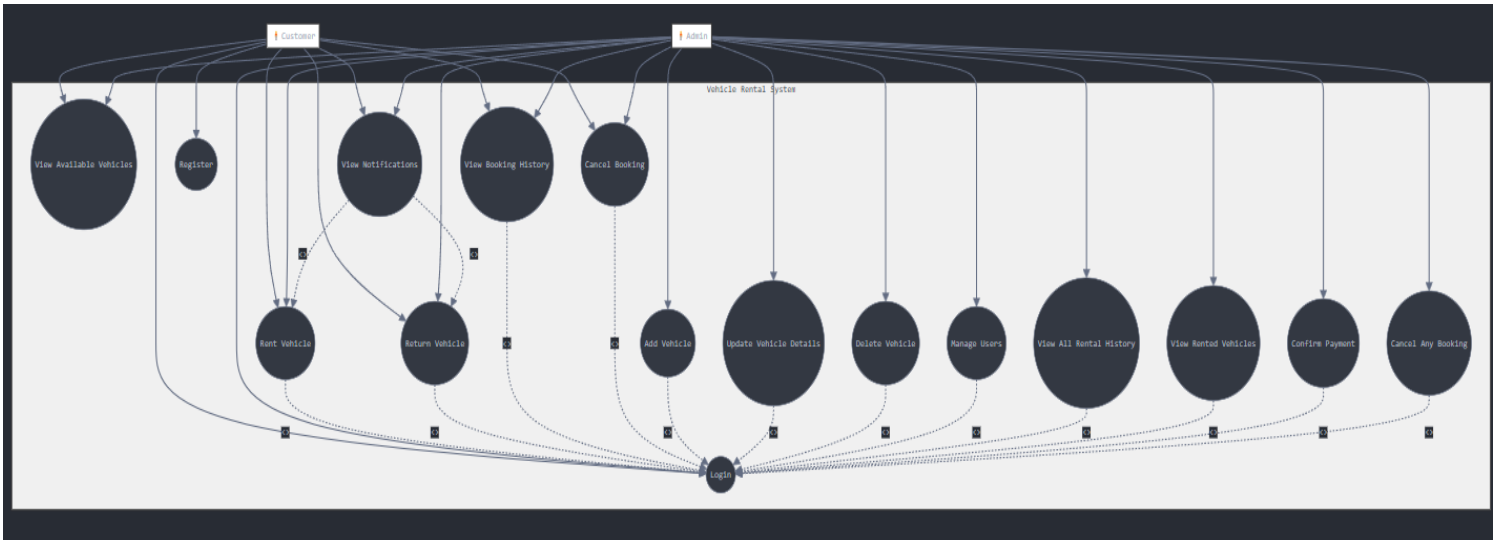

Figure 4 - Vehicle Rental System Use Case Diagram.

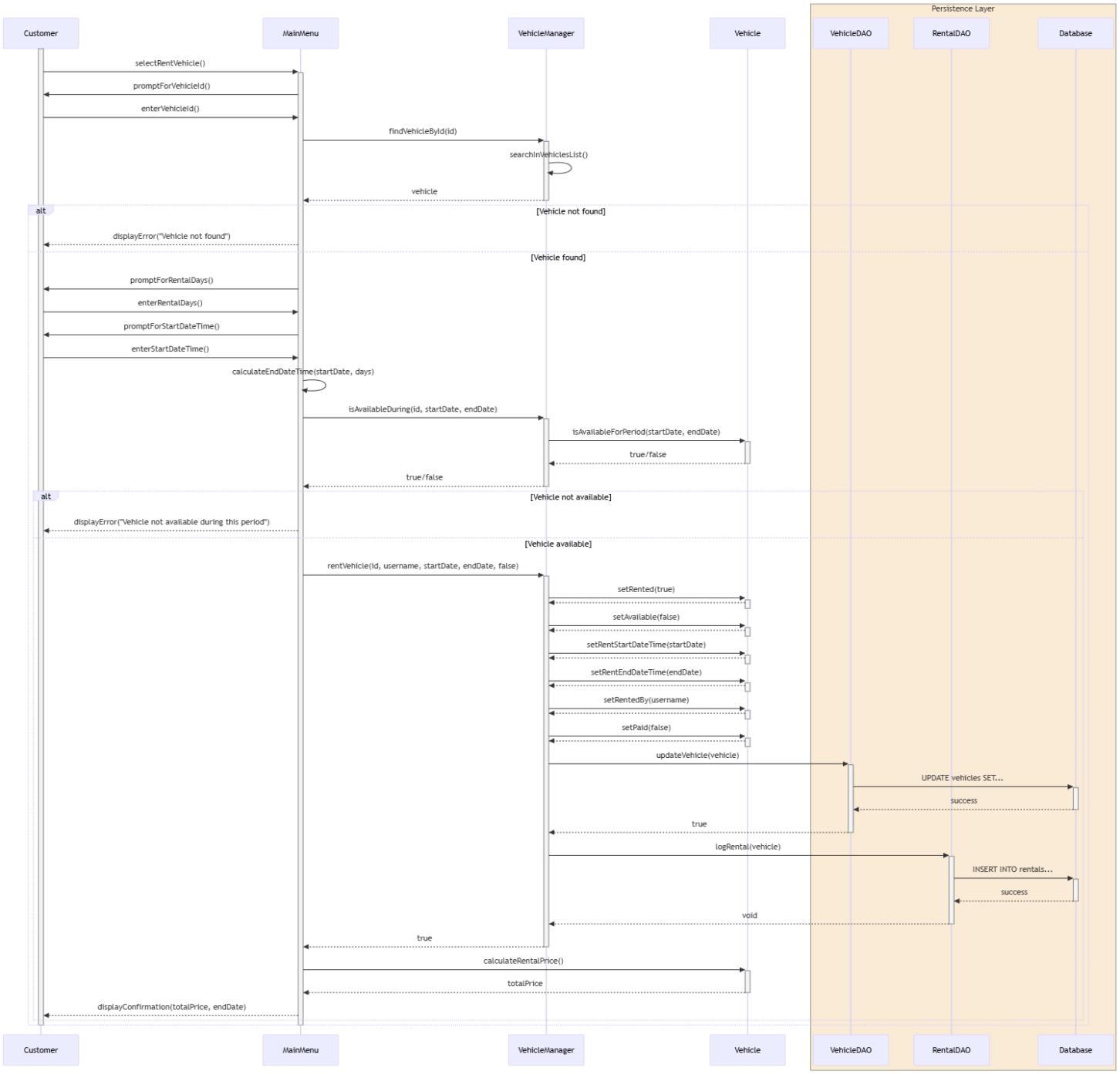

Figure 5 - Vehicle Rental System Sequence Diagram. Rent Vehicle Process.

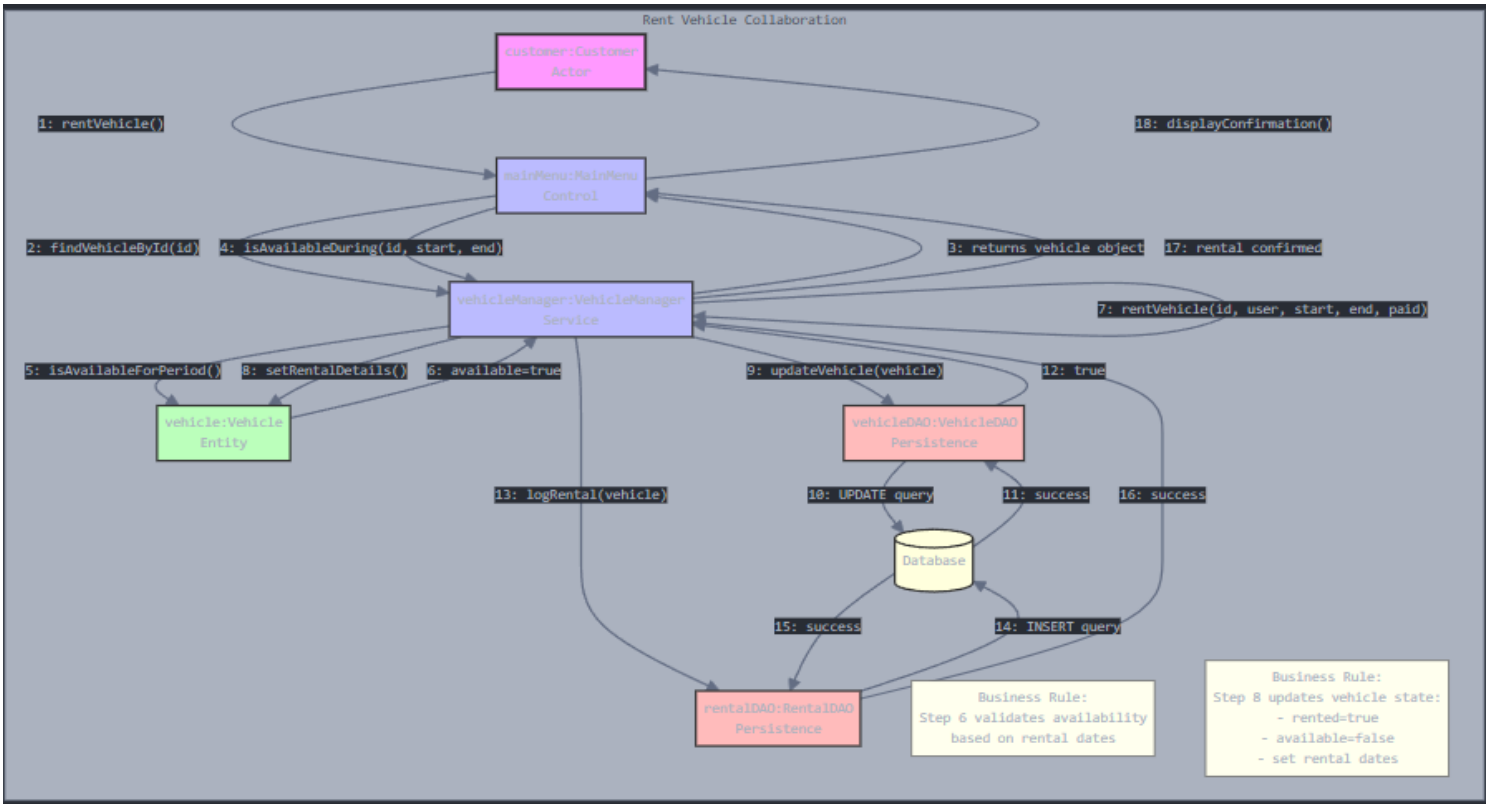

Figure 6 - Vehicle Rental System Collaboration Diagram. Rent Vehicle Process.

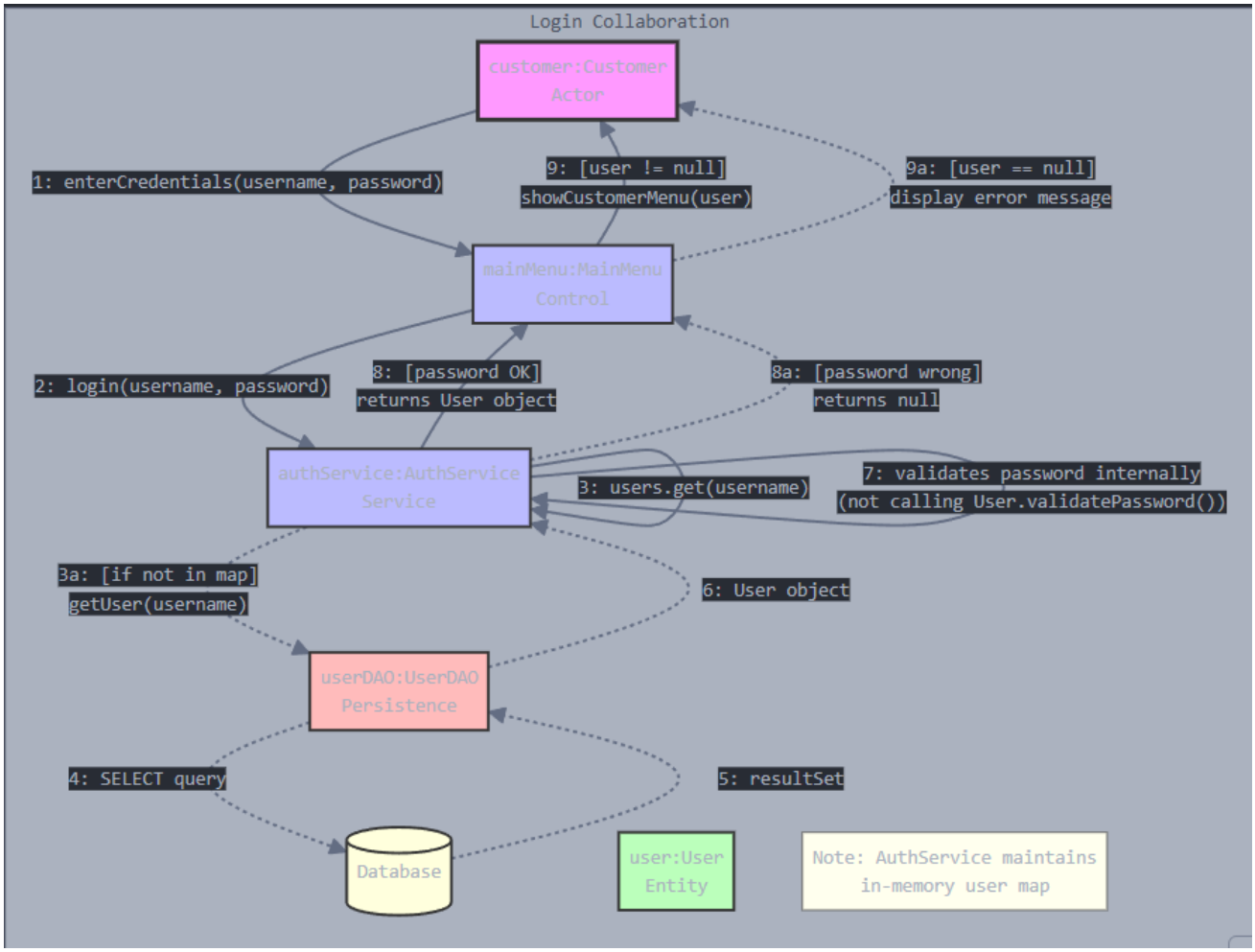

Figure 7 - Vehicle Rental System Collaboration Diagram. Login Process.

2.4. Implementation of OOP Principles.

There are four essential pillars of object-oriented design, namely **encapsulation, abstraction, inheritance and polymorphism**. In contemporary industrial environments these are also topped up by the **SOLID guidelines**, which also consist of five principles – open/closed principle, Liskov Substitution principle, Interface Segregation Principle, and Dependency Inversion Principle. Thus, this subsection critically evaluates how the Vehicle Rental System implements each principle and why these choices have been made to enhance maintainability, testability, and extensibility (Martin, 2014).

2.4.1. Encapsulation and Informal Hiding.

Each domain class (User, Vehicle, Rental) (Figures 8, 9 & 10) announces its state as private and exposes behaviour through coherent getter/setter methods. For instance, Vehicle.setRentalDetails() provides minimalistic updates to rentStart, tentEnd, and rentedBy, imperfect data from being written, which is a classical approach to implement the “tell-don’t-ask” advice. By localising invariants (like a paid rental must have a non-null totalPrice), the design satisfies the Design by Contract pre- and post-condition thinking, thus diminishing the danger of shared-state anomalies. Although the data from rentStart/rentEnd is saved in both the Vehicle and Rental, which can be identified as an encapsulation issue, it is important to note that in such a manner the availability checks are becoming faster, as no data joins on the database need to be queried (Meyer, 2022; Evans, 2003).

```java
public abstract class Vehicle {
    private String id;
    private String brand;
    private String model;
    private double pricePerDay;
    private boolean available = true;
    private boolean rented = false;
    private LocalDateTime rentStartDateTime;
    private LocalDateTime rentEndDateTime;
    private boolean paid = false;
    private String rentedBy;

    public Vehicle(String id, String brand, String model, double pricePerDay) {
        this.id = id;
        this.brand = brand;
        this.model = model;
        this.pricePerDay = pricePerDay;
    }

    //Gets the vehicle ID.
    public String getId() {
        return id;
    }

    //Gets the brand of the vehicle.
    public String getBrand() {
        return brand;
    }

    //Gets the model of the vehicle.
    public String getModel() {
        return model;
    }

    //Gets the rental price per day.
    public double getPricePerDay() {
        return pricePerDay;
    }

    //Checks if the vehicle is currently available.
    public boolean isAvailable() {
        return available;
    }
```

Figure 8 - Part of the Code of the Vehicle Model Class.

```java
public class User {
    private String username;
    private String password;
    private String role;
    private String name;
    private String email;
    private String phone;
    private String address;

    public User(String username, String password, String role, String name, String email, String phone, String address) {
        this.username = username;
        this.password = password;
        this.role = role;
        this.name = name;
        this.email = email;
        this.phone = phone;
        this.address = address;
    }

    // Getters methods for details
    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public String getRole() {
        return role;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }
```

Figure 9 - - Part of the Code of the User Model Class

```java
public boolean rentVehicle(String vehicleId, String username, LocalDateTime startDateTime, LocalDateTime endDateTime, boolean isPaid) {
    Vehicle vehicle = findVehicleById(vehicleId);
    if (vehicle != null && isAvailableForPeriod(vehicle, startDateTime, endDateTime)) {
        vehicle.setRented(rented:true);
        vehicle.setAvailable(available:false);
        vehicle.setRentStartDateTime(startDateTime);
        vehicle.setRentEndDateTime(endDateTime);
        vehicle.setRentedBy(username);
        vehicle.setPaid(isPaid);

        try {
            vehicleDAO.updateVehicle(vehicle);
            RentalDAO rentalDAO = new RentalDAO();
            rentalDAO.logRental(vehicle);
            return true;

        } catch (SQLException e) {
            System.out.println("Failed to update rental in database: " + e.getMessage());
        }
    }
    return false;
}
```

Figure 10 - Part of the Code of the VehicleManager Service where the rentVehicle method uses the Vehicle interface rather than internal fields - decoupled client code from object state.

2.4.2. Abstraction via Service and DAO Layers.

High-level policies have been insulated in service classes (VehicleManager, AuthService, NotificationService), while lo-level data operations are conducted in the DAO classes. In such a manner the application follows a policy-mechanism segregation (Figure 11 & 12). Likewise,

VehicleManger.rentVehicle() (Figure 10) abstracts “booking” into a single call that governs availability checks, entity updates, and persistence. Hence, the client code is unaware of the JDBC, which uplifts loose coupling in the system. As for the SOLID principles, the application complies with the Dependency-Inversion principle, where business logic depends on abstraction (DAO interfaces) rather than concrete SQL statements (Bonocore, 2022).

```java
//Updates the contact details of an existing user. Applies changes to both the in-memory user map and the database.
public boolean updateUser(String username, String newName, String newPhone, String newEmail, String newAddress) {
    User user = users.get(username);
    if (user != null) {
        user.setName(newName);
        user.setPhone(newPhone);
        user.setEmail(newEmail);
        user.setAddress(newAddress);
        try {
            boolean updated = userDAO.updateUser(user);
            if (updated) {
                users.put(username, user);
            }
            return updated;
        } catch (SQLException e) {
            System.out.println("Error updating user: " + e.getMessage());
            return false;
        }
    }
    return false;
}
```

Figure 11 - The updateUser method exemplifies the policy-mechanism separation in the AuthService class, where data is abstracted via access through the DAO.

```java
public class VehicleDAO {
    private static final String URL = "jdbc:mysql://localhost:3306/vehiclerental";
    private static final String USER = "root";
    private static final String PASSWORD = "root";


    //Saves a new vehicle to the database.
     public void saveVehicle(Vehicle vehicle) throws SQLException {
        String sql = "INSERT INTO vehicles (id, brand, model, price_per_day, available, rented, rent_start, rent_end, paid, rented_by, type) " +
                     "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";
        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
             PreparedStatement stmt = conn.prepareStatement(sql)) {

            stmt.setString(parameterIndex:1, vehicle.getId());
            stmt.setString(parameterIndex:2, vehicle.getBrand());
            stmt.setString(parameterIndex:3, vehicle.getModel());
            stmt.setDouble(parameterIndex:4, vehicle.getPricePerDay());
            stmt.setBoolean(parameterIndex:5, vehicle.isAvailable());
            stmt.setBoolean(parameterIndex:6, vehicle.isRented());
            stmt.setObject(parameterIndex:7, vehicle.getRentStartDateTime());
            stmt.setObject(parameterIndex:8, vehicle.getRentEndDateTime());
            stmt.setBoolean(parameterIndex:9, vehicle.isPaid());
            stmt.setString(parameterIndex:10, vehicle.getRentedBy());
            stmt.setString(parameterIndex:11, vehicle.getVehicleType());
            stmt.executeUpdate();
        }
    }

    //Retrieves vehicles from the database by ID.
    public Vehicle getVehicleById(String id) throws SQLException {
        String sql = "SELECT * FROM vehicles WHERE id = ?";
        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
             PreparedStatement stmt = conn.prepareStatement(sql)) {

            stmt.setString(parameterIndex:1, id);
            ResultSet rs = stmt.executeQuery();

            if (rs.next()) {
                return buildVehicleFromResultSet(rs);
            }
        }
        return null;
    }
```

Figure 12 - Part of the Code of the VehicleDao Class.

2.4.3. Inheritance and the Open-Closed Principle.

Vehicle is an abstract concept (Figure 13) extended by Car, Van, and Motorcycle. The child classes provide with specialised constructors but inherit common rental-state logic. If a new vehicle type would be required, the logic could simply be extended by a new class which will inherit common traits from Vehicle, reducing the need of changing current code, thus complying with the Open-Closed Principle (Figure 14). The subclasses do not deliver any new behaviour beyond constructor wiring (Bloch, 2018; Martin, 2014).

```java
public abstract class Vehicle {
    private String id;
    private String brand;
    private String model;
    private double pricePerDay;
    private boolean available = true;
    private boolean rented = false;
    private LocalDateTime rentStartDateTime;
    private LocalDateTime rentEndDateTime;
    private boolean paid = false;
    private String rentedBy;

    public Vehicle(String id, String brand, String model, double pricePerDay) {
        this.id = id;
        this.brand = brand;
        this.model = model;
        this.pricePerDay = pricePerDay;
    }

    //Gets the vehicle ID.
    public String getId() {
        return id;
    }

    //Gets the brand of the vehicle.
    public String getBrand() {
        return brand;
    }

    //Gets the model of the vehicle.
    public String getModel() {
        return model;
    }

    //Gets the rental price per day.
    public double getPricePerDay() {
        return pricePerDay;
    }

    //Checks if the vehicle is currently available.
    public boolean isAvailable() {
        return available;
    }
```

Figure 13 - Part of the Vehicle Class Code.

```java
//It constructs a Car object with the specified ID, brand, model, and daily rental price.
public class Car extends Vehicle {
    public Car(String registrationNumber, String make, String model, double rentalPrice) {
        super(registrationNumber, make, model, rentalPrice);
    }

    @Override
    public String getVehicleType() {
        return "Car";
    }
}
```

Figure 14 - The Car Child Class.

2.4.4. Polymorphism and Late Binding.

Polymorphism is utilised in the application in two ways:

- Role-based dispatch – within MainMenu.start() the system calls showAdminMenu(currentUser) or showCustomerMenu(currentUser) (Figure 15) at run-time depending on the user role of the current system operator. This portrays a polymorphic menu mechanism and adheres to the Liskov Substitution Principle. Nonetheless, the resolution path is initiated by an “if” statement, rather than virtual method overriding, the workflow is still operated by role-specific methods that have a mutual conceptual contract, so the effect still envisions late binding of behaviour to the abstraction role (Liskov and Wing, 1994).
- Interface polymorphism – each DAO implements a shared CRUD contract, that allows future swapping of database technologies without client-code alterations.

```java
//Launches the main menu loop for the system. Handles login, registration, and routes users to role-specific menus.
public static void start() {
    System.out.println(x:"=== Welcome to the Vehicle Rental System ===");
    User currentUser = null;

    while (currentUser == null) {
        System.out.println(x:"\n1. Login\n2. Register\n3. Exit");
        int choice = getIntInput(prompt:"Choose an option: ");
        switch (choice) {
            case 1 -> {
                String username = getStringInput(prompt:"Username: ");
                String password = getStringInput(prompt:"Password: ");
                currentUser = authService.login(username, password);
                if (currentUser == null) System.out.println(x:"Invalid credentials. Try again.");
            }
            case 2 -> {
                String username = getStringInput(prompt:"Choose a username: ");
                if (authService.getUser(username) != null) {
                    System.out.println(x:"Username already exists.");
                    break;
                }
                String password = getStringInput(prompt:"Choose a password: ");
                String name = getStringInput(prompt:"Full name: ");
                String phone = getStringInput(prompt:"Phone number: ");
                String email = getStringInput(prompt:"Email address: ");
                String address = getStringInput(prompt:"Home address: ");
                authService.register(username, password, role:"CUSTOMER", name, phone, email, address);
                currentUser = authService.login(username, password);
            }
            case 3 -> {
                System.out.println(x:"Exiting application.");
                return;
            }
            default -> System.out.println(x:"Invalid option. Try again.");
        }
    }

    if ("ADMIN".equalsIgnoreCase(currentUser.getRole())) showAdminMenu(currentUser);
    else showCustomerMenu(currentUser);
}
```

Figure 15 - The start() Method Code from the MainMenu Service Class.

2.4.5. Single Responsibility (SRP) and Cohesion.

Employing SRFP and also supports Cohesion in the modules (Fowler *et al.*, 2002):

- AuthService -used only for authentication and registration.
- NotificationService – used only for temporal queries for upcoming bookings (customer side) and overdue(admin) or upcoming returns(customer).
- VehicleManager – Rental life cycle orchestration.

2.4.6. Important Reflective Notes on Implemented OOP Principles.

From a broad perspective, the application encompasses textbook utilisation of the above principles, however there are two things to be mindful of:

- Encapsulation vs. Performance – as discussed earlier the duplication of saving of the rental state in two classes enhances the query speed but lowers information hiding (Fowler *et al.*, 2002).
- Inheritance vs. Composition – at the current state of the system, the hierarchy is a good fit, however if more features are to be included this might bloat the base class. Thus, the introduction of a strategy object could provide a cleaner extension path (Gamma *et al.*, 2009).

2.5. Design Patterns Applied.

The Vehicle Rental Systems deliberately utilises a small set of proven design patterns rather than all available ones, to suit the current requirements towards the system. This section assesses each one that have been used and justifies its employment. They are summarised in Table 1, where it is also provided the rationale for their usage and a critical reflection on each pattern.

Table 1 - Design Patterns Applied in the Vehicle Rental System Project (Gamma et al., 2009; Fowler et al., 2002; Larman, 2005).

Pattern	Concrete Implementation in Code	Rationale / Benefits	Critical Reflection
Data Acess Object (DAO)	VehicleDAO, UserDAO, RentalDAO encapsulate JDBC operations (addVehicle(), getUserByUsername(), etc.). The service layer (VehicleManager, AuthService) never issues SQL directly.	Separation of concerns—business logic stays database-agnostic. Allows unit testing via stub DAOs. Facilitates future migration to JPA or a REST micro-service.	Adds boilerplate; could be further abstracted behind a Repository interface (Evans, 2004) to remove duplicate CRUD signatures.
Singleton	utils/DatabaseConnection.java – static field connection + public getConnection() that instantiates once.	Guarantees a single shared JDBC connection, avoiding repeated driver loads. Simplifies DAO construction (no explicit pooling required).	Not thread-safe; concurrent CLI threads could race.
Service Layer / Facade	services/ VehicleManager.java (core booking, availability, admin ops) AuthService.java (login/registration/user CRUD) NotificationService.java (overdue & reminder logic)	Presents coarse-grained operations to UI (MainMenu). Aggregates multiple DAO calls, shielding the controller from business rules. Central place to add transactions/validation.	Currently lacks explicit transaction rollback; wrapping DAO calls in a Unit-of-Work or Spring @Transactional annotation would harden consistency.
Layered Architecture	Clear package tiers: ui (CLI in MainMenu.java), services, dao, models, utils. All dependencies flow downward only (UI → Service → DAO).	Enhances modifiability and parallel team work.	Coupling between CLI and services is still compile-time static; introducing interfaces or a REST layer would further decouple presentation.
GRASP Controller	VehicleManager.java is the single "system controller" for rental workflows; MainMenu.java acts as the UI controller (handling user events then delegating).	Centralises responsibility for critical policies such as rental-overlap checks and payments. Prevents "God objects" by dividing UI flow and domain control.	As features grow, sub-controllers or Command objects per use-case can be considered to keep VehicleManager from becoming monolithic.

2.6. Testing and Verification

Thorough testing and verification are pivotal to demonstrate that the Vehicle Rental System adheres to the functional and quality requirements. The project has followed a multi-layer test strategy – unit and integration. This is also continued with some manual exploratory tests on the command-line UI. Thus, the selected way of testing analyses the riskiest areas like persistence, date-range logic. Although, the testing process is not examining the complete functionality of the app due to time constraints, the most important ones are tested.

2.6.1. Test Strategy

In Table 2 are summarised the automated and manual tests that have been performed on the product, their objectives and key assertions.

Table 2 - Test Strategy for the Vehicle Rental System.

Level	Test Class	Objective	Key Assertions
Unit	VehicleManagerTest (UT-01)	Booking conflict logic	rentVehicle() returns false and RentalDAO.logRental() is never invoked when the requested slot overlaps an existing rental.
	AuthServiceTest (UT-02, UT-03)	Reject bad credentials Prevent duplicate usernames	login() with wrong password returns null. Second call to register() with same username returns false.
	NotificationServiceTest (UT-04, UT-05)	Identify overdue rentals Flag returns due < 24 h	Polymorphic vehicle list produces 1 overdue notice and 1 upcoming-return warning respectively.
Integration	RentalDAOIntegrationTest (IT-01)	Verify DAO - live MySQL round-trip using Testcontainers	After logRental(), getAllRentalHistory() returns the persisted row.
Exploratory Manual Testing	Register and Login Workflow (MT-01)	Identify possible issues with the Register and Login Process	"Registration successful" banner shown. New username now appears in View Users (admin menu) or is returned by AuthService.getAllUsers(). Immediate login.
	Customer Booking and Overlap Period for the same Vehicle (MT-2 & 3)	Identify issues with booking a vehicle and receive an error for overlapping one.	Happy-path - System prints total price and some booking details.. Overlap attempt - CLI displays that the vehicle cannot be rented and does not create a new rental row.
	Admin confirm payment upon Pick up (MT-04)	Assess proper work state of the feature.	Selecting a booking ID changes its paid column from false to true in the live rental list.
	Overdue & upcoming notifications (MT-05)	Assess proper work state of the feature.	Such notifications are actually displayed to the users.

2.6.2. Toolkit Used for the Unit and Integration Testing.

The tools utilised for the automated testing are listed as follows:

- JUnit 5.10.2 – contemporary test framework.
- Mockito 5.11.0 – inline mocking for VehicleDAO, RentalDAO, and constructor-intercepted UserDAO.

- Testcontainers 1.19.6 – spins an ephemeral MySQL 8 container; the test overrides DatabaseConnection via overrideJdbcUrl() to isolate the real database.
- Maven Surefire 3.2.5 – executes tests; Byte Buddy 1.14.14 pinned for JDK 23 compatibility
- Docker – for the Integration test. **Note, that if docker is not running the tests will fail**.

2.6.3. Results from the Automated Unit and Integration Testing.

From Figure 16 is clearly visible that the above planned tests have been conducted, and the system is performing as expected with no apparent bugs or other problems. Hence, the automated tests are passed with success.

```
[INFO] Results:
[INFO]
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  35.317 s
[INFO] Finished at: 2025-05-12T18:12:01+01:00
[INFO] ------------------------------------------------------------------------
```

Figure 16 - Results from the Automated Unit and Integration Tests.

2.6.4. Results from the Exploratory Manual Testing.

2.6.4.1. Results from the MT – 01 Test. Register and Login Workflow.

In Figures 17,18 & 19 can be found that the registration of the user John22 is successful and when login in again as an admin user and checking from the user list, the new username is appearing in it. Thus, the test is deemed successfully passed.

```
=== Welcome to the Vehicle Rental System ===

1. Login
2. Register
3. Exit
Choose an option:
```

Figure 17 - Welcome Login Screen.

```
=== Welcome to the Vehicle Rental System ===

1. Login
2. Register
3. Exit
Choose an option: 2
Choose a username: John22
Choose a password: john22
Full name: John Johnson
Phone number:
Email address:
Home address:

=== Customer Menu ===
    1. Rent Vehicle
    2. Return Vehicle
    3. View Available Vehicles
    4. View Notifications
    5. View Booking History
    6. Cancel Upcoming Booking
    7. Logout

Choose an option:
```

Figure 18 - New Customer Successfully Registering and Logging in.

```
1. Login
2. Register
3. Exit
Choose an option: 1
Username: admin1
Password: adminpass

=== Admin Menu ===
    1. Add Vehicle
    2. Rent Vehicle
    3. Return Vehicle
    4. View Available Vehicles
    5. Add Regular User
    6. Add Admin User
    7. View Rented Vehicles
    8. View Notifications
    9. Update Vehicle Details
    10. Manage Users
    11. View Rental History
    12. Cancel Booking
    13. Confirm Payment on Pickup
    14. Logout

Choose an option: 10

=== User Management ===
1. View All
2. Update
3. Delete
4. Back
Choose: 1
Username: user1 | Role: CUSTOMER | Name: Bob User | Email: 444-555-6666
Username: bojo1 | Role: CUSTOMER | Name: Bojo Jiji | Email: bojo1@abv.bg
Username: tosho1 | Role: CUSTOMER | Name: Todor Todorov | Email: tosho1@abv.bg
Username: admin1 | Role: ADMIN | Name: Alice Admin | Email: 111-222-3333
Username: John22 | Role: CUSTOMER | Name: John Johnson | Email:
Username: Jorji1 | Role: CUSTOMER | Name: Jorji Jorji | Email: jorji1@gmail.com
```

Figure 19 - Verification of the Newly Registered User Through the Admin Menu.

2.6.4.2. *Results from the MT – 02 & 03 Test. Customer Booking and Overlap Period for the Same Vehicle.*

In Figures 20 and 21 is visualised that the new user managed to book a vehicle successfully and when trying to book the same vehicle for a second time in an overlapping period, the application throws a message that it cannot be done. Thus, the tests are passed.

```
=== Customer Menu ===
    1. Rent Vehicle
    2. Return Vehicle
    3. View Available Vehicles
    4. View Notifications
    5. View Booking History
    6. Cancel Upcoming Booking
    7. Logout

Choose an option: 3
Motorcycle - Honda CBR500R (ID: YR99JEP), €40.0/day

=== Customer Menu ===
    1. Rent Vehicle
    2. Return Vehicle
    3. View Available Vehicles
    4. View Notifications
    5. View Booking History
    6. Cancel Upcoming Booking
    7. Logout

Choose an option: 1
Enter vehicle ID to rent: YR99JEP
Enter number of rental days: 3
Enter rental start date and time (yyyy-MM-dd HH:mm): 2025-05-13 13:00
Rental confirmed. Total: £120.00
Return due: 2025-05-16 13:00
```

Figure 20 - New User Booking Vehicle Successfully.

```
=== Customer Menu ===
    1. Rent Vehicle
    2. Return Vehicle
    3. View Available Vehicles
    4. View Notifications
    5. View Booking History
    6. Cancel Upcoming Booking
    7. Logout

Choose an option: 1
Enter vehicle ID to rent: yr99jep
Enter number of rental days: 5
Enter rental start date and time (yyyy-MM-dd HH:mm): 2025-05-14 13
Invalid format. Use yyyy-MM-dd HH:mm.
Enter rental start date and time (yyyy-MM-dd HH:mm): 2025-05-14 13:00
Vehicle is not available during this period.

=== Customer Menu ===
    1. Rent Vehicle
    2. Return Vehicle
    3. View Available Vehicles
    4. View Notifications
    5. View Booking History
    6. Cancel Upcoming Booking
    7. Logout
```

Figure 21 - Unsuccessful Overlapping Booking Attempt.

2.6.4.3. *Results from the MT – 04. Admin Confirming Payment upon Pick up.*

Figures 22 & 23 depict the process of the admin confirming the payment upon vehicle pick up. It is clear that the process is successful, and that the payment status is changed from “false” (which is default when the booking is made by the customer) to “true”. Hence, the test is passed.

```
=== Admin Menu ===
    1. Add Vehicle
    2. Rent Vehicle
    3. Return Vehicle
    4. View Available Vehicles
    5. Add Regular User
    6. Add Admin User
    7. View Rented Vehicles
    8. View Notifications
    9. Update Vehicle Details
    10. Manage Users
    11. View Rental History
    12. Cancel Booking
    13. Confirm Payment on Pickup
    14. Logout

Choose an option: 13
Unpaid Rentals:
1. LL92JJK | Rented by: tosho1 | Due: 2025-05-14 19:40
2. YR99JEP | Rented by: John22 | Due: 2025-05-16 13:00
Select number to confirm payment: 1
Payment confirmed.
```

Figure 22 - Successful Payment Confirmation by Admin.

id	vehicle_id	rented_by	start_time	end_time	total_price	paid	vehicle_type
1	KM62KKL	user1	2025-05-06 11:30:00	2025-05-13 11:30:00	420	0	Car
2	KM62STU	tosho1	2025-05-07 13:00:00	2025-05-10 13:00:00	150	0	Car
3	KM62STU	tosho1	2025-05-07 13:00:00	2025-05-10 13:00:00	150	0	Car
4	KM62STU	tosho1	2025-06-05 19:00:00	2025-06-10 19:00:00	250	0	Car
5	KM62STU	tosho1	2025-05-06 19:00:00	2025-05-11 19:00:00	250	0	Car
6	KM62STU	bojo1	2025-05-06 20:30:00	2025-05-08 20:30:00	100	0	Car
7	KM62STU	bojo1	2025-05-06 20:30:00	2025-05-08 20:30:00	100	1	Car
8	RV59RRC	Jorji1	2025-05-06 20:00:00	2025-05-07 20:00:00	80	1	Van
9	YR99JEP	John22	2025-05-13 13:00:00	2025-05-16 13:00:00	120	0	Motorcycle
10	LL92JJK	tosho1	2025-05-12 19:40:00	2025-05-14 19:40:00	40	1	Car
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 23 - The Changed Payment Status Rectified in the Database.

2.6.4.4. Results from the MT – 05. Overdue & Upcoming Notifications.

Figure 24 exemplifies the notifications available to the user that has already made a booking.

Thus, the notification test is deemed as passed.

```
=== Customer Menu ===
    1. Rent Vehicle
    2. Return Vehicle
    3. View Available Vehicles
    4. View Notifications
    5. View Booking History
    6. Cancel Upcoming Booking
    7. Logout

Choose an option: 4
Notifications:
Upcoming rental: Motorcycle - Honda CBR500R (ID: YR99JEP), €40.0/day starts on 2025-05-13T13:00
```

Figure 24 - User Notification for Upcoming Rental.

2.7. Security and Authentication Implementation.

In Table 3 are summarised the current security and Authentication Implementation, what risks have been mitigated by the chosen approach and what is the remaining gap that may need to be addressed depending on future application development.

Table 3 - Security and Authentication Implementation, Risks Mitigated and Gaps Unresolved (Anderson, 2020; Khan et al.*, 2023).*

Concern	Current Implementation	Risk Mitigated / Remaining Gap
Credential persistence & leakage	AuthService.register() stores passwords in the users table through UserDAO.saveUser() using a parameterised INSERT prepared-statement, eliminating SQL-injection vectors .	Prevents injection, but passwords are kept in clear text (User.getPassword() simply returns the raw string) .
Authentication logic	AuthService.login() retrieves the User from a HashMap cache loaded at start-up (loadUsersFromDatabase()) and checks password.equals() .	Cache provides O(1) lookup, reducing DB exposure. Still vulnerable to timing-attack differentials due to String.equals.
Transport security	CLI is local; DB traffic uses default TCP 3306 without TLS.	Low risk for localhost development but unsafe once deployed on a network.
SQL injection	All DAOs (UserDAO, VehicleDAO, RentalDAO) utilise PreparedStatement placeholders for every variable input—e.g., getUser(String) and logRental(Vehicle) User.	Input is never concatenated into SQL, blocking classic injection.
Authorisation (RBAC)	Role is persisted in users.role and checked centrally: MainMenu.start() routes to showAdminMenu vs. showCustomerMenu after login AuthServiceDatabaseConnection. Each admin-only operation is reachable only from the admin menu (e.g., manageUsers(), confirmPaymentOnPickup()).	Coarse-grained but effective; accidental privilege escalation impossible via CLI navigation.
Payment state integrity	VehicleManager.confirmPayment() flips paid=true and RentalDAO.markRentalAsPaid() executes a conditional UPDATE … WHERE paid=FALSE preventing duplicate confirmations (idempotent) MainMenuRentalDAO.	Guards against double-spend.
Sensitive logging	Errors are printed to stdout (e.g., "Error saving user: …").	No stack-trace disclosure, but SQL error messages may leak schema.

3. Conclusion and Future Scope

The Vehicle Rental System exemplifies that robust object-oriented design, layered architecture, and incremental testing can yield a maintainable solution. Among key deliverables that were achieved in the current work are a fully functional CLI application satisfying core use-cases (registration, booking, payment on pickup, and notifications), also a clean Object-oriented model with a polymorphic Vehicle hierarchy, cohesive service layer, and JDBC DAOs that decouple persistence. Another, thing that this project is delivering and effective solution to the business requirements that is proof checked through quality

assurance, that validates functional correctness, security controls and role-based access. Software design patterns were also effectively applied to solve common software development issues – DAO, Service/Façade, GRASP Controller, plus layered isolation. To add up, the solution mitigates SQL-injection and enforces coarse-grained RBAC, although clear-text password storage, single-threaded execution, and minimal logging indication room for maturation.

3.1. Scope for Further Development

Table 4 outlines some possible areas for further development.

Table 4 - Areas for Further Development.

Road-map Item	Rationale & Expected Benefit
BCrypt password hashing + TLS MySQL channel	Closes OWASP A02 Cryptographic Failures gap; elevates confidentiality for production deployment.
RESTful API + React/Flutter front-end	Decouples presentation from business logic; enables mobile clients and expands test surface (Postman + Selenium).
Connection-pool & multi-thread booking	Removes singleton connection bottleneck; supports simultaneous customers; requires race-condition tests with Awaitility.
Payment gateway integration (Stripe sandbox)	Automates paid flag and receipt, reducing manual admin steps.
Scheduler for nightly notification sweep (Quartz)	Moves cron-like job out of CLI, enabling unattended operation and richer reminder channels (email/SMS).
Mutation testing & SonarQube pipeline	Raises defect detection effectiveness beyond 90 %, supporting continuous quality gates.
Containerised deployment (Docker Compose)	Matches Testcontainers environment, easing Dev→Prod parity and fulfilling DevOps best practice.

Integrating the above enhancements would not only harden security and scalability but also offer strong base for further additional exploration.

4. Personal Learning Reflections

Developing this project has been my most hands-on software development experience so far. It practically thought me how to connect object-oriented principles to real application development.

The assignment allowed me to further deepen my knowledge and understanding of the object-oriented principles like polymorphism, inheritance, and practically apply them. It has also allowed me to go a step beyond into research more adverse design patterns outside of the ones that we studied in class (produced by the GoF). Examples for such patterns are the DAO pattern, the Layered Architecture, and GRASP Controller that I have successfully applied through my code. I do recognise the fact that sticking to the GoF patterns and applying them in a minimalistic environment would have been enough, but I did want to achieve the latter knowledge and went a step further to try something unfamiliar. In the process of researching these patterns I have also stumbled upon a number of ISO standards regarding software security, design standards and more that are growing my knowledge outside of the main ones that we studied in class. Testing the code has also been an interesting part of my learning curve. Although, I followed the testing session and used Junit 5, I did use other tools like Mockito and Docker following different tutorials, which enriched my knowledge regarding the testing process. And last, but not least, I have used VS Code to develop the application, which I do consider a mistake. The IDE created quite a few issues – mainly with versions of different tools like Maven that did not comply with JDK 23, for some reason, so I lost quite a lot of time trying to sort out the dependencies.

5. Bibliography

Anderson, R. (2020) *Security Engineering, 3rd Edition*. Third Edition. [Online]. Wiley. Available at: https://learning.oreilly.com/library/view/security-engineering-3rd/9781119642787/ [Accessed: 12 May 2025].

Hu, V.C. & Freidman, A. (2014) Attribute based access control definition and considerations (NIST special publication 800-162) and attribute assurance. *IT Professional Conference: Challenges in Information Systems Governance, IT Pro 2014*. Institute of Electrical and Electronics Engineers Inc. [Accessed: 11 May 2025].

Bass, Len. et al. (2015) *Software architecture in practice*. Addison-Wesley.

Bloch, J. (2018) *Effective Java: Best Practices für die Java-Plattform by Bloch, Joshua, author, Bloch, Joshua. German*. [Online]. Wesley/Pearson Education. Available at: https://prism.librarymanagementcloud.co.uk/bolton-ac/items/335457 [Accessed: 11 May 2025].

Bonocore, G. (2022) *Hands-On Software Architecture with Java*. [Online]. Packt Publishing. Available at: https://learning-oreilly-com.ezproxy.bolton.ac.uk/library/view/hands-on-software-architecture/9781800207301/?ar=&email=%2Bhp9P49CamjLDmv5eqqqg8NEqj5sZju0&tstamp=1746994241&id=A7B578DCA4C40111E68A7875D9195465FA1D0F6E [Accessed: 11 May 2025].

Evans, E. (2003) *Domain-Driven Design: Tackling Complexity in the Heart of Software*. [Online]. Addison-Wesley Professional. Available at: https://learning-oreilly-com.ezproxy.bolton.ac.uk/library/view/domain-driven-design-tackling/0321125215/?ar=&email=%2Bhp9P49CamjLDmv5eqqqg8NEqj5sZju0&tstamp=1746993598&id=970FF4B9918EDE57E9159CA439A16E5BDAC2F1BF [Accessed: 11 May 2025].

Fowler, M. et al. (2002) Patterns of Enterprise Application. In: *Wesley, Addison*. [Online]. Available at: https://prism.librarymanagementcloud.co.uk/bolton-ac/items/198367 [Accessed: 7 May 2025].

Gamma, Erich. et al. (2009) *Design patterns : elements of reusable object-oriented software*. 395. Addison-Wesley.

Gast, H. (2015) Part IV: Responsibility-Driven Design. . In: Addison-Wesley *How to Use Objects*. [Online]. Addison-Wesley. Available at: https://learning-oreilly-

com.ezproxy.bolton.ac.uk/library/view/how-to-use/9780133840100/ch11.html#ch11 [Accessed: 7 May 2025].

Hunt, A. & Thomas, D. (1999) *The Pragmatic Programmer,*. First Edition. [Online]. Addison-Wesley Professional. Available at: https://learning-oreilly-com.ezproxy.bolton.ac.uk/library/view/hunt-the-pragmatic-programmer/020161622X/?ar=&email=%2Bhp9P49CamjLDmv5eqqqg8NEqj5sZju0&tstamp=1746979358&id=31773551EAE86A5265271EDBAA9A4FF0D08C2264 [Accessed: 11 May 2025].

International Organisation for Standardisation (2022) ISO/IEC 27002:2022(en), Information security, cybersecurity and privacy protection — Information security controls. *iso.org*. ISO. Available at: https://www.iso.org/obp/ui/#iso:std:iso-iec:27002:ed-3:v2:en [Accessed: 11 May 2025].

International Standardisation Organisation (2023) *ISO/IEC 25010:2023 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. Available at: https://www.iso.org/standard/78176.html [Accessed: 11 May 2025].

Khan, J.R. et al. (2023) A Survey on SQL Injection Attacks Types & their Prevention Techniques. *JISR on Computing*, 21(2). Shaheed Zulfikar Ali Bhutto Institute of Science and Technology. [Accessed: 12 May 2025].

Laplante, P.A. & Kassab, M. (2022) What Every Engineer Should Know about Software Engineering: Second Edition. In: CRC Press *What Every Engineer Should Know about Software Engineering: Second Edition*. [Online]. CRC Press. Available at: doi:10.1201/9781003218647/EVERY-ENGINEER-KNOW-SOFTWARE-ENGINEERING-PHILLIP-LAPLANTE-MOHAMAD-KASSAB/RIGHTS-AND-PERMISSIONS [Accessed: 7 May 2025].

Larman, C. (2005) *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition*. [Online]. Pearsons. Available at: https://learning-oreilly-com.ezproxy.bolton.ac.uk/library/view/applying-uml-and/0131489062/ [Accessed: 11 May 2025].

Laudon, K.C.. & Traver, C.Guercio. (2022) *E-commerce 2023 : business, technology, society*. [Online]. Pearson. Available at: https://dokumen.pub/e-commerce-business-technology-society-17th-edition-17nbsped-0137922205-9780137922208.html [Accessed: 11 May 2025].

Liskov, B.H. & Wing, J.M. (1994) A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6), 1811–1841. ACMPUB27New York, NY, USA. [Accessed: 11 May 2025].

Martin, R.C.. (2014) *Agile software development, principles, patterns, and practices*. Pearson.

Meyer, B. (2022) *Object-oriented Software Construction* . Second Edition. [Online]. Available at: https://bertrandmeyer.com/wp-content/upLoads/OOSC2.pdf [Accessed: 11 May 2025].

Nielsen, J. (1992) The Usability Engineering Life Cycle. *Computer*, 25(3), 12–22. [Accessed: 11 May 2025].

Noback, M. (2018) Principles of Package Design. In: Apress *Principles of Package Design*. [Online]. Apress. Available at: doi:10.1007/978-1-4842-4119-6_1 [Accessed: 7 May 2025].

Sommerville, Ian. (2016) *Software Engineering, Global Edition*. [Online]. Pearson Education Limited. Available at: https://prism.librarymanagementcloud.co.uk/bolton-ac/items/198140 [Accessed: 11 May 2025].

6. Word Count

3098 words

7. GAI Declaration

I declare that no GAI was used for the development of this work.

The software used is:

- Microsoft Word and its grammar/spelling corrector
- Microsoft Excel – for the development of comparison tables and graphs
- Microsoft Paint
- Drawio
- Mendeley Reference Manager
- Google, Google Scholar and Discover@Bolton to narrow down and uplift research

8. Appendices

8.1. List of Abbreviations Used

1. **BDP** – Behavioural Design Pattern
2. **CDP** – Creational Design Pattern
3. **CRUD** – Create, Read, Update, and Delete
4. **DAO** – Data Access Object
5. **DB** - Database
6. **DBMS** – Database Management System
7. **DP** – Design Pattern
8. **JDBC** – Java Database Connectivity
9. **MVC** – Model-View-Controller
10. **OOP** – Object Oriented Programming
11. **RBAC** - Role-based Access Control
12. **SD** – Software Development
13. **SDP** – Structural Design Pattern
14. **SE** – Software Engineering
15. **SDLC** – Software Development Life Cycle
16. **SRP** – Single Responsibility Principle