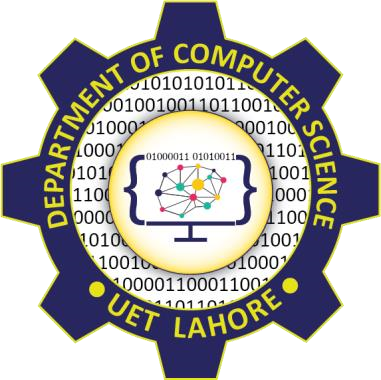
**Al Fajar Motors**



Session: 2022 – 2026

**Submitted by:**

Bisma Fajar 2022-CS-66

**Submitted To:**

Dr. Awais Hassan

Department of Computer Science

**University of Engineering and Technology** **Lahore Pakistan**

**Content**

[Al- Fajar Motors 6](#_Toc139812066)

[Short Description of project 6](#_Toc139812067)

[Users of the application 6](#_Toc139812068)

[Functional Requirements 6](#_Toc139812069)

[OOP Concepts in the Project 9](#_Toc139812070)

[1. Association: 9](#_Toc139812071)

[2. Inheritance: 9](#_Toc139812072)

[3. Polymorphism: 9](#_Toc139812073)

[Comparison with Traditional Procedural Programming 9](#_Toc139812074)

[1. Modularity and Reusability: 9](#_Toc139812075)

[2. Code Organization and Maintenance: 9](#_Toc139812076)

[3. Polymorphism and Flexibility: 10](#_Toc139812077)

[4. Data Abstraction and Security: 10](#_Toc139812078)

[Design Pattern Implementation: 10](#_Toc139812079)

[1. Business Logic (BL) Pattern: 10](#_Toc139812080)

[2. Data Access Layer (DL) Pattern: 10](#_Toc139812081)

[3. User Interface (UI) Pattern: 11](#_Toc139812082)

[Classes Details: 11](#_Toc139812083)

[1. `Person` (Base class): 11](#_Toc139812084)

[2. `Staff` (Derived from `Person`): 11](#_Toc139812085)

[3. `Customer` (Derived from `Person`): 11](#_Toc139812086)

[4. `Bus` (Base class): 11](#_Toc139812087)

[5. `Driver`(Derived from ‘Staff’): 12](#_Toc139812088)

[6. PersonDL: 12](#_Toc139812089)

[7.StaffDL: 12](#_Toc139812090)

[8.CustomerDL: 12](#_Toc139812091)

[9. BusDL: 12](#_Toc139812092)

[10. DriverDL: 12](#_Toc139812093)

[11. `PersonUI`: 12](#_Toc139812094)

[12. `CustomerUI`: 12](#_Toc139812095)

[13. AdminUI: 12](#_Toc139812096)

[Conclusion: 13](#_Toc139812097)

[Challenges: 13](#_Toc139812098)

[1. Designing a flexible and extensible architecture: 13](#_Toc139812099)

[2. Managing data persistence: 13](#_Toc139812100)

[3. User input validation: 13](#_Toc139812101)

[Achievements: 13](#_Toc139812102)

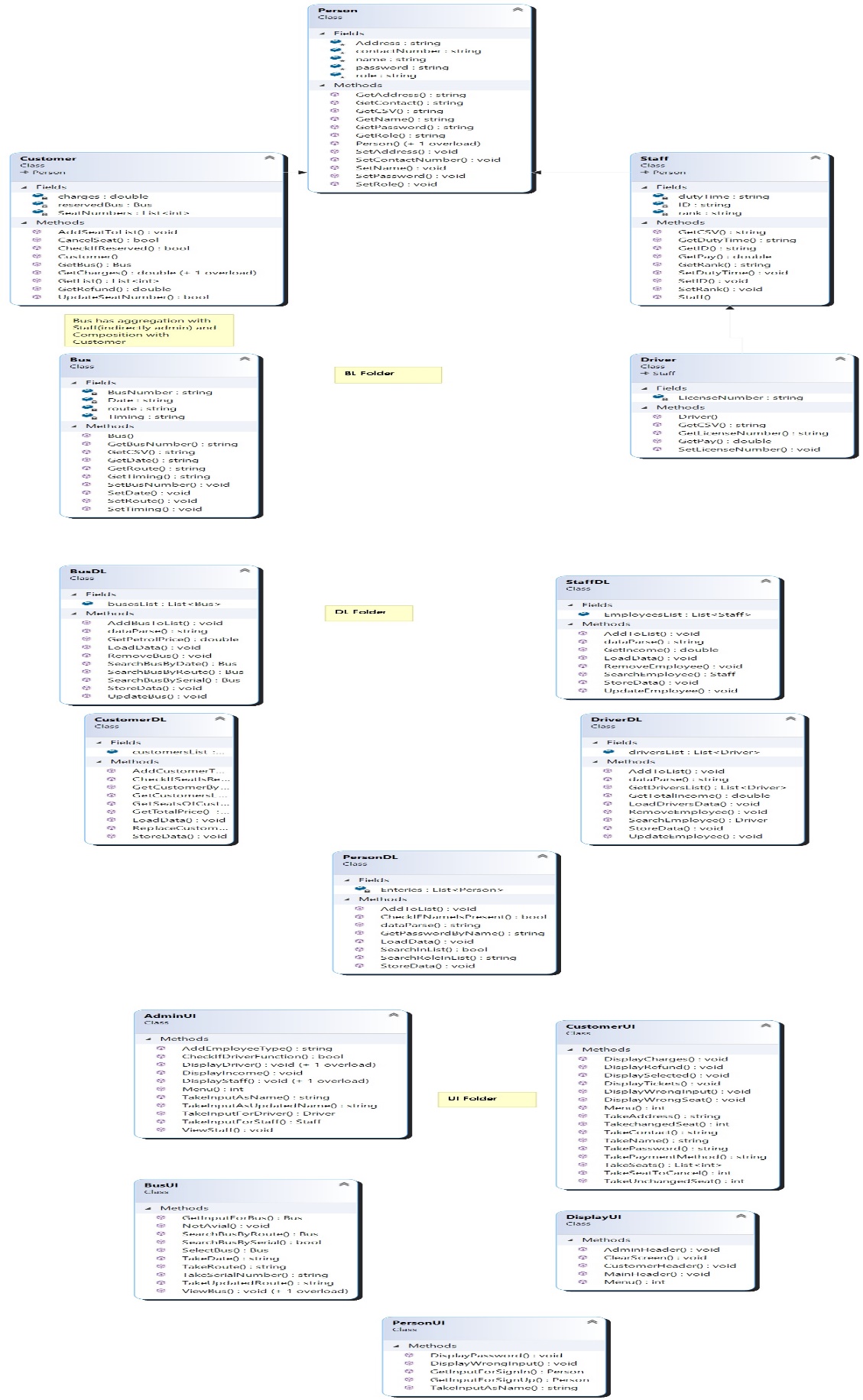
[1. Modularity and separation of concerns: 13](#_Toc139812103)

[2. Code reusability: 13](#_Toc139812104)

[3. Improved maintainability: 13](#_Toc139812105)

[Lesson Learned: 13](#_Toc139812106)

# CRC



# Al- Fajar Motors

# Short Description of project

* + Bus reservation system will help a user to reserve seats online. This project covers two types of persons to access the application, first will be the admin and second will be the customer. At first a user will come and enter its user type if he is admin then he will be given access to money earned by selling tickets and loss if any, he can add, delete, update, search and view employees and buses. He will also be able to allot numbers to buses and he will set the timings of buses. And if the user type is customer then he can search the timings of buses, view all buses he will be able to reserve the seat in the bus, he can delete his ticket and can get refund according to policy of company, he can update ticket and view all tickets. This application will serve the computer user in a way that he/she has not to go bus station to check whether a bus is available or not and can reserve seats easily.

# Users of the application

There are two users of the application:

* + Admin
  + Customer

# Functional Requirements

***User***

***Story ID As a I want to perform So that I can***

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Admin | * Add employee. | First he/she will tell which type of employee he want to enter Driver or Other Staff . |
| 1 | Admin | * Remove employee. | He will check if the employee is present in the list then he will remove it from list and file. |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Admin | * Update employee. | He will check if the employee is present in the list then he will update it from list and file. |
| 1 | Admin | * Search employee. | He will check if the employee is present in the list. |
| 1 | Admin | * View all employees. | Can see the whole list. |
| 1 | Admin | * Add bus. | He can add bus. |
| 1 | Admin | * Remove bus. | He will check if the bus is present in the list then he will remove it from list and file. |
| 1 | Admin | * Update bus. | He will check if the bus is present in the list then he will update it from list and file. |
| 1 | Admin | * Search bus. | He will check if the bus is present in the list then he will return its object. |
| 1 | Admin | * View all buses. | Can see the whole list. |
| 1 | Admin | * View Income. | Can view profit and loss. |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | Customer | * Search bus by route. | Customer can search the buses entered by admin. |
| 2 | Customer | * View all buses. | Can see the whole list of buses. |
| 2 | Customer | * Select bus. | Can select bus by entering serial number. |
| 2 | Customer | * Add personal details. | Add the personal details of passenger because sometimes the account name and passenger name are different. |
| 2 | Customer | * Reserve seat. | Enter seat number or numbers to reserve. |
| 2 | Customer | * View tickets. | Can view the reserved seats. |
| 2 | Customer | * Update ticket. | He will check if the seat is present in the list then he will update it. |
| 2 | Customer | * Pay for ticket. | Payment by cash or card. |
| 2 | Customer | * Cancel Seat. | Cancel the seat if he wants. |
| 2 | Customer | * Get refund. | Get Refund for cancelled seat. |

# OOP Concepts in the Project

The project incorporates the concepts of association, inheritance, and polymorphism to facilitate a well-structured and modular approach to software development. These concepts provide several advantages over traditional procedural programming, such as code reusability, modularity, flexibility, and enhanced code organization and maintenance.

## 1. Association:

The project demonstrates association by establishing relationships between classes and objects. For example, the `Customer` class is associated with the `Bus` class through the concept of a serial number of bus in which the customer is travelling.

## 2. Inheritance:

The project utilizes inheritance to create class hierarchies, where child classes inherit properties and behaviors from parent classes. The `Staff` and `Customer` classes inherit from the `Person` class, while ‘Driver’ class is associated with ‘Staff’ which provides common attributes and methods such as name and password. This inheritance relationship facilitates code reuse, encapsulation, and the organization of classes based on their similarities and differences.

## 3. Polymorphism:

Polymorphism is employed in the project through method overriding and method overloading. Method overriding allows child classes (`Admin` and `Customer`) to provide their own implementation of methods inherited from the parent class (`Person`). This enables the program to use the same method name while executing different behaviors based on the specific class instance.

For example there are functions for calculating pay and income in staff.

# Comparison with Traditional Procedural Programming

In comparison to traditional procedural programming approaches, the adoption of object-oriented programming (OOP) brings several advantages:

## 1. Modularity and Reusability:

OOP promotes modular design by encapsulating data and behaviors within classes, facilitating code reuse. Classes can be easily reused in different parts of the program or in future projects. In procedural programming, code tends to be fragmented, making it challenging to reuse and maintain.

## 2. Code Organization and Maintenance:

OOP offers a structured and organized approach to code structure. Classes, objects, and their relationships mirror real-world entities and interactions, resulting in code that is easier to comprehend and maintain. In procedural programming, functions and data are often scattered throughout the code, making it more difficult to grasp the overall structure and flow.

## 3. Polymorphism and Flexibility:

Polymorphism in OOP allows for flexible and extensible code. Objects can be treated uniformly through their common interfaces or base classes, leveraging inheritance and polymorphic behavior. This simplifies code development, maintenance, and enhances scalability. Achieving similar flexibility and extensibility in procedural programming requires additional code and manual handling of different cases.

## 4. Data Abstraction and Security:

OOP enables data abstraction, where complex data structures and operations are hidden behind classes and interfaces. This abstraction enhances security by controlling data access and preventing unauthorized modifications. In procedural programming, data is often more exposed, increasing the risk of data manipulation or unauthorized access.

Overall, OOP provides a well-structured, modular, and flexible approach to software development compared to traditional procedural programming. It promotes code reuse, simplifies maintenance, and enhances scalability, making it the preferred choice for complex and large-scale projects.

# Design Pattern Implementation:

The project incorporates the Business Logic (BL), Data Access Layer (DL), and User Interface (UI) design patterns to ensure modularity and separation of concerns.

## 1. Business Logic (BL) Pattern:

The BL pattern is implemented to encapsulate the business rules and operations of the application. It separates the business logic from the UI and DL layers, promoting modularity and code reusability. The BL layer acts as an intermediary between the UI and DL layers, handling data validation, processing, and coordinating interactions between different components. By isolating the business logic, changes or updates to the UI or DL layers do not affect the core business rules, making the codebase more maintainable.

## 2. Data Access Layer (DL) Pattern:

The DL pattern is used to abstract the data storage and retrieval operations from the BL and UI layers. It provides a consistent interface for accessing and manipulating data, regardless of the underlying data storage technology or implementation details. The DL layer encapsulates database queries, file operations, or any other data access mechanisms. By separating data access logic from the BL layer, the codebase becomes more flexible and adaptable to changes in data sources or storage technologies. It also allows for better testability, as the BL layer can be tested independently without requiring a specific data source.

## 3. User Interface (UI) Pattern:

The UI pattern is employed to provide a structured and user-friendly interface for interacting with the application. It separates the presentation logic from the underlying business logic, enabling easier maintenance and updates to the UI without impacting the core functionality. The UI layer handles user input, displays relevant information, and interacts with the BL layer to perform actions and retrieve data. By decoupling the UI from the business logic, the application becomes more modular, allowing for different UI implementations or adaptations for various platforms or user experiences.

These design patterns ensure modularity and separation of concerns by dividing the application into distinct layers, each with its own specific responsibilities. The BL pattern isolates and manages the business logic, the DL pattern handles data access operations, and the UI pattern provides an intuitive interface for user interaction. This approach promotes code reusability, maintainability, and scalability, as changes or updates in one layer have minimal impact on the others. It also enhances testability, as each layer can be tested independently, leading to more robust and reliable software.

Overall, the implementation of these design patterns in the project ensures a well-structured and maintainable codebase, allowing for future enhancements and modifications with ease.

# Classes Details:

Here are the key classes in the project along with their responsibilities:

## 1. `Person` (Base class):

- Responsible for representing a person entity with common attributes like name, password, contact, address and role.

- Provides methods for setting and getting the person's name and password.

## 2. `Staff` (Derived from `Person`):

- Represents an admin user in the system.

- It has attributes of duty time, rank and Id.

- Inherits the properties and methods from the `Person` class and adds specific behavior for an admin user.

## 3. `Customer` (Derived from `Person`):

- Represents a customer user in the system.

- Inherits the properties and methods from the `Person` class and adds specific behavior for a customer user.

- Stores additional information such as the customer's address.

## 4. `Bus` (Base class):

- Represents the product it has attributes of bus serial, timing, route, date.

- Provides methods for setting and getting the product's attributes.

## 5. `Driver`(Derived from ‘Staff’):

- It has common attributes with staff and also license number of driver.

## 6. PersonDL:

Contains list of users of application and handles CURD operation on users.

## 7.StaffDL:

Contains method for loading and storing admin from file. Moreover, have function to load transaction history from file or store transactions in file.

## 8.CustomerDL:

Contains methods for loading and storing customers from file. Moreover, have function related to the cart of customers. To add or remove products from cart.

## 9. BusDL:

- Contains list of buses to perform CURD for admin and customer.

## 10. DriverDL:

- Contains list of buses to perform CURD for admin.

## 11. `PersonUI`:

- Contains static methods for handling user interface interactions related to persons (users).

- Provides menus for admin, customer, and sign-in/sign-up functionality.

- Takes user input for creating persons, signing in, and signing up.

## 12. `CustomerUI`:

- Contains static methods for handling user interface interactions related to customers.

- Displays and interacts with product information, shopping cart, and customer details.

## 13. AdminUI:

Contains static methods for handling user interface interaction related to buses, drivers and other employees.

These classes represent the core entities and functionality of the application. They encapsulate the properties and behaviors related to persons, products, and user interactions. By dividing responsibilities among classes, the project achieves modularity and separation of concerns, making it easier to understand, maintain, and extend the codebase.

# Conclusion:

The project aimed to develop a business application with a focus on modularity, separation of concerns, and utilizing object-oriented design principles and design patterns. The project successfully implemented the BL (Business Logic), DL (Data Access Layer), and UI (User Interface) layers to achieve these goals.

By utilizing the BL, DL, and UI design patterns, the project ensured modularity and separation of concerns. The BL layer encapsulated the business logic and rules, the DL layer handled data access and manipulation, and the UI layer provided the user interface for interaction. This design approach allowed for easy maintenance, scalability, and code reusability.

# Challenges:

Throughout the development process, some challenges were faced. These included:

1. Designing a flexible and extensible architecture: Ensuring that the application's architecture could accommodate future changes and enhancements required thoughtful design decisions and planning.

2. Managing data persistence: Implementing an efficient and reliable data storage mechanism and integrating it with the DL layer posed challenges. Selecting the appropriate database technology and ensuring proper data retrieval and manipulation were key considerations.

3. User input validation: Validating user input to maintain data integrity and security required careful handling and validation mechanisms in the BL and DL layers.

# Achievements:

Despite these challenges, the project achieved the following:

1. Modularity and separation of concerns: The project successfully implemented a layered architecture with clear separation between the BL, DL, and UI layers. This ensured maintainability, scalability, and ease of testing.

2. Code reusability: By utilizing object-oriented design principles, design patterns, and modular architecture, the project achieved code reusability. This allowed for efficient development and reduced duplication of code.

3. Improved maintainability: The separation of concerns and modular design made it easier to identify and modify specific components of the application without impacting other parts. This facilitated maintenance and future enhancements.

# Lesson Learned:

Lessons learned from the project include the importance of proper architectural planning, adherence to design principles, and the effective use of design patterns to achieve modular and maintainable code. Additionally, careful consideration should be given to data persistence and user input validation to ensure data integrity and security.