TITLE: WESTMINSTER RENTAL VEHICLES(COURSE WORK)

NAME: JOSHUA ADDAI-MARNU

STUDENT NUMBER: 19548571

COURSE: OBJECT-ORIENTED PROGRAMMING

DATE: 04/01/2024

REFERENCE TO RECORDED VIDEO DEMONSTRATION OF THE IMPLEMENTED SYSTEM:

https://youtu.be/HnTYt6_iN0w?si=6yvEg96TgOJuWlPz

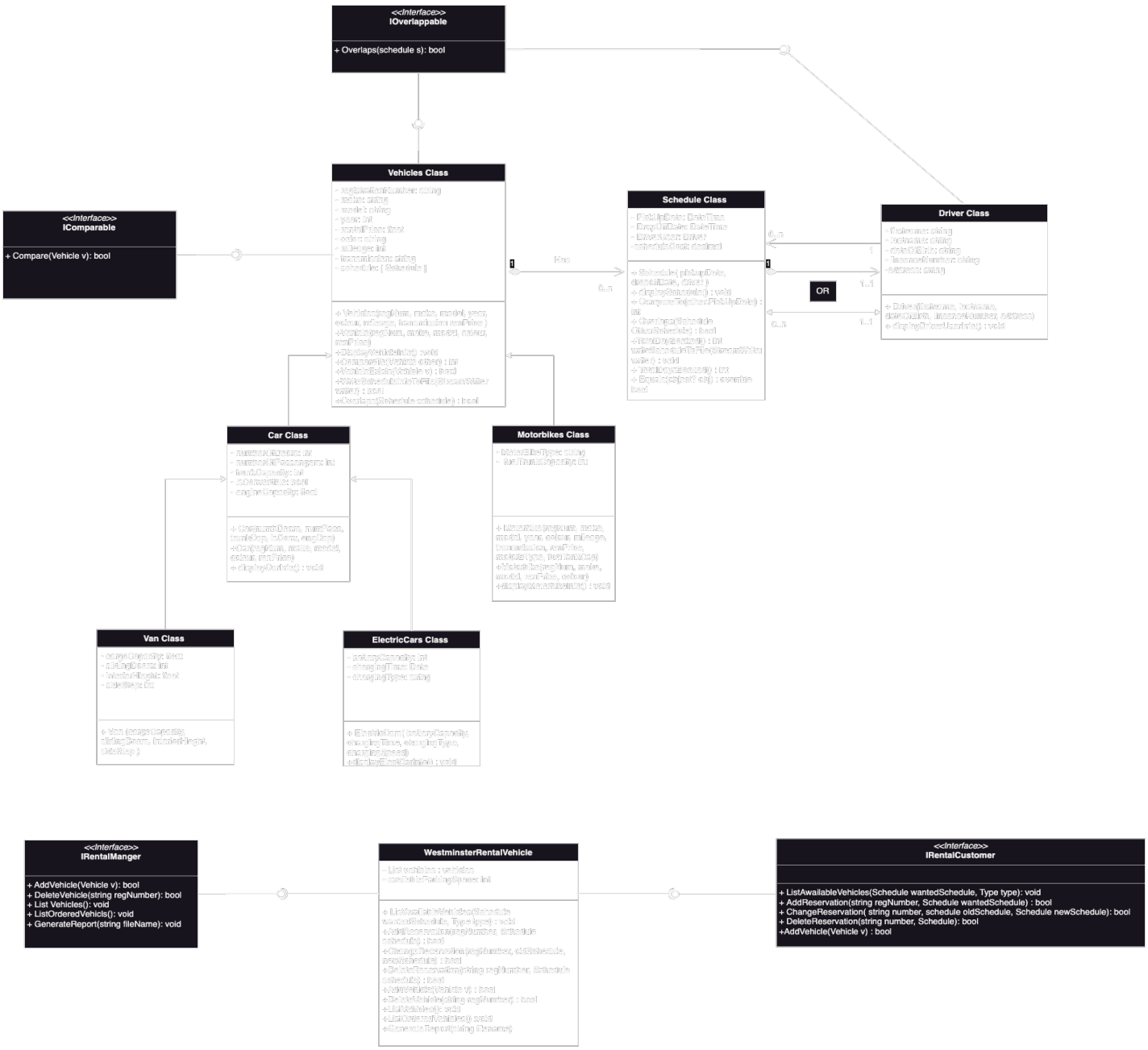
……

OVERVIEW:

The system, named WestminsterRentalVehicle, is designed for managing a vehicle rental service. The Class Diagram provides a visual representation of the system's structure and relationships. It includes classes such as Vehicles, Car, Motorbikes, Van, ElectricCars, Schedule, WestminsterRentalVehicle, Driver, and interfaces IOverlappable, IRentalCustomer, IRentalManager, and IComparable. These classes encapsulate entities like vehicle types, rental schedules, and drivers. The diagram illustrates associations, inheritance, and interfaces, offering a comprehensive view of the system's design.

As I delve into the document, I will explain the classes representing various vehicles, interfaces defining user interactions, and abstract classes providing a foundation for scheduling and system management. The Westminster Rental Vehicle System is designed for efficiency and adaptability to meet evolving rental demands in the automotive industry.

CLASS UML DIAGRAM



SYSTEM ENTITIES:

Classes:

1.Vehicle Class (Parent class):

The Vehicles class is an abstract class and it encapsulates general vehicle attributes such as Registration Number, Make and Model, Rental Price, Colour, Mileage, Transmission Type, etc.
Important methods that provide essential functionalities for vehicle management and interaction include:
-DisplayVehicleInfo() Method: Displays detailed information about the vehicle.
-CompareTo() Method: Implements the IComparable interface for sorting vehicles based on make and model.
- VehicleExists() Method: Checks if a vehicle with the same registration number already exists.
-Overlaps() Method: Checks if the vehicle's schedule overlaps with another schedule.
-writeInfoToFile() Method: Writes initial vehicle information to a file.
-writeScheduleInfoToFile() Method: Writes schedule information for the vehicle to a file.

These methods collectively provide functionality for initializing, displaying information, managing schedules, and file I/O for the Vehicle class. The class is designed to be a base class for different types of vehicles in a rental system, providing common

2.Car Class (subclass of Vehicle class)

Inherits properties from the Vehicle, the Car class adds specific attributes like the number of doors, number of passengers, trunk capacity, convertible status, and engine Capacity. It’s a specialized type of vehicle with attributes specific to cars, such as the number of doors, number of passengers, trunk capacity, etc.

3.Motorbikes Class(subclass of Vehicle class):
Inheriting attributes or properties from the Vehicle class, the motorbike class introduces attributes like Bike Type, Engine Capacity, and Fuel Trunk Capacity. Motorbike-Specific Operation method caters to functionalities exclusive to motorbikes. Represents a type of vehicle specializing in bikes, with attributes like bike type, engine capacity, and fuel tank capacity.

4. Van Class(subclass of Car class):

Inheriting properties from the Car class, the Van Class adds attributes such as Cargo Capacity, Sliding Doors, Interior Height, and Side Step. The van-specific Operation method manages operations tailored to vans.

5. Electric Car Class(subclass of Car class):

Inheriting from the Car class, the electric cars Class includes attributes like battery capacity, charging time, and charging type and it represents a type of vehicle that is electric,
Driver Class:
Driver Class attributes include First Name and Last Name, Date of Birth, License Number, and Schedule. This class helps manage driver information and associated reservation schedules.

6. Schedule Class:
Abstract Schedule Class introduces attributes Pickup Date, Drop-off Date, and Driver. Methods like overlaps, write schedules to file and total days booked provide a foundation for managing reservation schedules.

6. Westminster Rental Vehicle Class:
Representing the overall system, Westminster Rental Vehicle Class includes attributes like list of vehicles and available parking space. Methods cover core functionalities like:
-ListAvailableVehicles() Method: List available vehicles based on the requested schedule and vehicle type.

-AddReservation() Method: Add a reservation for a vehicle.
-ChangeReservation() Method: Change an existing reservation for a vehicle.
-DeleteReservation() Method: Purpose: Delete a reservation for a vehicle.
-AddVehicle() Method: Add a new vehicle to the rental system.
-DeleteVehicle() Method: Delete a vehicle from the rental system.
-ListVehicles() Method: List all vehicles in the system.
-ListOrderedVehicles() Method: List all vehicles in alphabetical order of Make.
-GenerateReport() Method: Generate a report and save it to a file.

INTERFACES

IRentalCustomer Interface:
IRentalCustomer Interface outlines methods like List Available Vehicles, Add Reservation, Change Reservation, and Delete Reservation. Classes like Car and Van implement this interface, facilitating seamless interaction with rental customers.

IRentalManager Interface:
IRentalManager Interface defines methods such as Add Vehicle, Delete Vehicle, List Vehicles, List Ordered Vehicles, and Generate Reports. WestminsterRentalVehicle Class implements this interface, allowing efficient management of the rental system.

IComparable Interface:
IComparable Interface includes methods like Compare, facilitating the comparison of vehicles based on predefined criteria. Classes like Vehicle may implement this interface for standardized comparisons.

OOP Principles Applied:

Encapsulation:
Encapsulation is prominently employed in the design through the use of classes and their associated attributes and methods. Each class encapsulates a set of related

attributes (e.g., vehicle details, driver information) and methods (e.g., CompareTo() and Overlaps). Attributes are often declared as private, accessible only within the class, fostering information hiding and preventing unintended external access. Methods act as interfaces to manipulate the internal state, promoting data integrity and minimizing direct manipulation. For instance, the VehicleExists() method in the Vehicles class encapsulates the logic for checking if a vehicle with the same registration number already exists.
Encapsulation enhances modularity, allowing changes to the internal implementation without affecting other parts of the system.
Encapsulation is achieved through the use of access modifiers such as public (`+`) and private (`-`) in the class properties and methods. For instance, properties within the `Vehicles` class, such as regNumber and `make`, are marked as private (`-`), restricting direct access and modification. Public methods like displaySchedule() provide controlled access to internal functionalities, promoting information hiding and maintaining a clear interface for interaction.

Inheritance:
Inheritance is strategically utilized to establish a hierarchical relationship between classes. The Car, Motorbikes, Van, and ElectricCars classes inherit from the `Vehicles` class, inheriting its common attributes and methods. This promotes code reuse by inheriting the general vehicle properties, while still allowing for specialization through additional attributes and methods in the derived classes. Inheritance enhances maintainability, as changes to the common features can be made in the parent class, automatically reflecting in the derived classes.

Polymorphism:
Polymorphism is achieved through interfaces and method overriding. For example, the IComparable interface allows different vehicle classes to implement their comparison logic via the CompareTo method. This enables flexibility, as various classes can be compared based on their unique criteria. The IRentalCustomer interface demonstrates polymorphism by providing a common set of methods that can be implemented differently by various rental customer classes. This fosters adaptability and extensibility, allowing the system to accommodate new classes conforming to the specified interfaces without altering existing code.

Polymorphism enhances the robustness and flexibility of the design, supporting diverse interactions and behaviours within the system.

RELATIONSHIPS: ASSOCIATION, COMPOSITION, AND AGGREGATION:

The design exhibits various relationships, fostering connectivity and collaboration between entities.

Associations:
Associations are prevalent between entities, such as the association between the Vehicles class and the Schedule class. Each vehicle has a schedule, establishing a one-to-one relationship. Additionally, the Driver class is associated with the Schedule class, indicating that a driver is linked to a specific schedule. These associations are crucial for coordinating vehicle reservations and driver schedules, facilitating efficient management within the system. Also, there is an association between WestminsterRentalVehicle class and Driver class. Each WestminsterRentalVehicle has a driver, establishing a one-to-one relationship (Multiplicity: 1..1). The role is defined as "Has," indicating that a vehicle has a driver.

Composition:
Composition relationships are apparent in the design, especially in the `Schedule` class. The `Schedule` class encapsulates a `Driver` object, signifying a strong compositional bond. A schedule is composed of a driver, and when a schedule is deleted, it is likely that the associated driver is also removed. This close relationship implies a whole-part connection, reinforcing the interdependence between the `Schedule` and `Driver` entities.

Aggregation:
Aggregation relationships are observed in various contexts, such as the relationship between the WestminsterRentalVehicle class and the Vehicles class. This represents

a logical connection without a strict lifecycle dependency, allowing WestminsterRentalVehicle to be associated with multiple vehicles. The multiplicity in these aggregations ensures flexibility, enabling multiple entities to be linked without strong ownership. Aggregation is seen in relationships between IRentalCustomer and Vehicles, allowing customers to have multiple reservations without direct ownership or exclusive dependence.

In summary, the relationships in the design, comprising associations, compositions, and aggregations, are carefully defined to model the intricate connections between entities, supporting a comprehensive and adaptable vehicle rental management system.

REFLECTION ON DESIGN AND IMPLEMENTATION

The design demonstrates a commendable alignment with the specified system requirements, effectively capturing the essential entities, relationships, and behaviours outlined. The meticulous incorporation of classes, abstract classes, and interfaces ensures a comprehensive and extensible structure, adeptly addressing the complexity of a vehicle rental management system.

The implementation, ideally, should closely mirror the design to ensure a seamless translation of conceptualization into a functional system. Any deviations may lead to discrepancies between the intended functionality and the actual implementation. Therefore, a thorough examination of the codebase is necessary to verify adherence to the design.

Potential improvements or modifications could be suggested based on the ongoing reflection. It would be prudent to conduct a rigorous testing phase to ensure the robustness and reliability of the system. Additionally, performance optimization measures could be explored to enhance the system's efficiency, especially concerning large datasets or concurrent user interactions. Any discrepancies between the design and implementation should be meticulously addressed to prevent potential issues during deployment and usage.

Moreover, continuous feedback loops between design, implementation, and testing should be established to accommodate evolving requirements and ensure

the system remains adaptable. Regular code reviews and adherence to coding standards will contribute to a maintainable and scalable codebase, facilitating future enhancements and updates.

REFERENCE TO RECORDED VIDEO DEMONSTRATION OF THE IMPLEMENTED SYSTEM:

https://youtu.be/HnTYt6_iN0w?si=6yvEg96TgOJuWlPz