# **Transportation Fleet Management System - Project Documentation**

1. **OVERVIEW**

This program is a comprehensive simulation of a transportation fleet management system for a logistics company. It is built in Java and demonstrates core Object-Oriented Programming (OOP) principles to manage a diverse fleet of vehicles, including cars, trucks, buses, airplanes, and cargo ships.

The system supports creating and managing vehicles, simulating journeys, handling cargo and passengers, tracking maintenance schedules, and persisting the fleet's state to a file. User interaction is handled through a robust Command-Line Interface (CLI).

This Assignment is made by **Pramag Basantia (2024421)**.

1. **PROJECT STRUCTURE**

The project is organized into five distinct packages for maintainability and clarity, following the structure below :

src/  
|  
|--- exceptions/  
| |--- InsufficientFuelException.java  
| |--- InvalidOperationException.java  
| |--- OverloadException.java  
|  
|--- fleet/  
| |--- FleetManager.java  
| |--- VehicleFactory.java  
|  
|--- interfaces/  
| |--- CargoCarrier.java  
| |--- FuelConsumable.java  
| |--- Maintainable.java  
| |--- PassengerCarrier.java  
|  
|--- main/  
| |--- Main.java  
|  
|--- vehicles/  
 |--- Vehicle.java (Abstract)  
 |--- AirVehicle.java (Abstract)  
 |--- LandVehicle.java (Abstract)  
 |--- WaterVehicle.java (Abstract)  
 |--- Airplane.java  
 |--- Bus.java  
 |--- Car.java  
 |--- CargoShip.java  
 |--- Truck.java



1. **DEMONSTRATION OF OOP CONCEPTS**

This project is fundamentally designed around the four core pillars of OOP.

### **A. Inheritance**

Inheritance creates a logical class hierarchy, promoting code reuse and establishing an "is-a" relationship.

#### **Classes demonstrating Inheritance (extends keyword):**

* **LandVehicle**, **AirVehicle**, **WaterVehicle**: All extend the base Vehicle class.
* **Car**, **Truck**, **Bus**: All extend the LandVehicle class.
* **Airplane**: Extends the AirVehicle class.
* **CargoShip**: Extends the WaterVehicle class.

#### **Methods demonstrating Inheritance (reusing parent logic):**

* **Constructors (super() call)**: Every subclass constructor calls its parent's constructor using super() to initialize the inherited fields.
  + LandVehicle(id, model, ...) calls super(id, model, ...) in Vehicle.
  + Car(id, model, ...) calls super(id, model, ...) in LandVehicle.
  + Truck(id, model, ...) calls super(id, model, ...) in LandVehicle.
  + Bus(id, model, ...) calls super(id, model, ...) in LandVehicle.
  + Airplane(id, model, ...) calls super(id, model, ...) in AirVehicle.
  + CargoShip(id, model, ...) calls super(id, model, ...) in WaterVehicle.
* **Inherited Concrete Methods**: All concrete classes (Car, Truck, etc.) inherit and can use the following methods directly from Vehicle without re-implementing them:
  + displayInfo()
  + getCurrentMileage()
  + getId()
  + getMaxSpeed()
  + setMileage()
  + getModel()
  + compareTo()
* **Method Overriding**: Subclasses provide specialized implementations of methods defined in their superclass.
  + **estimateJourneyTime(double distance)**:
    - Defined as abstract in Vehicle.
    - Overridden in LandVehicle (adds 10% for traffic).
    - Overridden in AirVehicle (reduces 5% for direct paths).
    - Overridden in WaterVehicle (adds 15% for currents).

### **B. Polymorphism**

Polymorphism allows objects of different classes to be treated as objects of a common superclass (Vehicle). This is demonstrated extensively in the FleetManager class.

**Core Component demonstrating Polymorphism:**

* **FleetManager.fleet**: The private List<Vehicle> fleet; field can hold any object whose class is a subclass of Vehicle (e.g., Car, Airplane).

#### **Methods in FleetManager demonstrating Polymorphic Behavior:**

* **startAllJourneys(double distance)**: Iterates the fleet and calls vehicle.move(distance). The specific version of move() that gets executed depends on the actual runtime type of the vehicle object.
* **getTotalFuelConsumption(double distance)**: Iterates and calls vehicle.calculateFuelEfficiency() for each FuelConsumable vehicle.
* **generateReport()**: Iterates and calls vehicle.calculateFuelEfficiency(), vehicle.getCurrentMileage(), and vehicle.getClass().getSimpleName() on each object to gather statistics.
* **sortFleetByEfficiency()**: Uses Collections.sort(fleet), which relies on the polymorphic compareTo() method implemented in Vehicle and used by all subclasses. The comparison is based on the result of vehicle.calculateFuelEfficiency().
* **saveToFile(String filename)**: Iterates and calls vehicle.getClass().getSimpleName() , vehicle.getId(), etc, on each object to construct the CSV data.

#### **Overridden Methods Called Polymorphically:**

* **move(double distance)**: Implemented by Car, Truck, Bus, Airplane, CargoShip.
* **calculateFuelEfficiency()**: Implemented by Car, Truck, Bus, Airplane, CargoShip.

### **C. Abstraction (Abstract Classes)**

Abstract classes define a common template for a group of related subclasses but cannot be instantiated themselves.

#### **Abstract Classes:**

* Vehicle: Root of the hierarchy. Represents the general concept of a vehicle.
* LandVehicle: Represents land-based vehicles.
* AirVehicle: Represents air-based vehicles.
* WaterVehicle: Represents water-based vehicles.

#### **Abstract Methods (forcing implementation by subclasses):**

* **In Vehicle**:
  + move(double distance)
  + calculateFuelEfficiency()
  + estimateJourneyTime(double distance)

#### **Concrete Methods in Abstract Classes (providing shared functionality):**

* **In Vehicle**:
  + displayInfo()
  + getCurrentMileage()
  + getId()
  + compareTo(Vehicle other)
  + getMaxSpeed()
  + setMileage(double newMileage)
  + getModel()
* **In LandVehicle**:
  + estimateJourneyTime(double distance)
  + getNumWheels()
* **In AirVehicle**:
  + estimateJourneyTime(double distance)
  + getMaxAltitude()
* **In WaterVehicle**:
  + estimateJourneyTime(double distance)
  + getHasSail()

### 

### **D. Encapsulation (Interfaces)**

Interfaces define a contract of behaviors that can be implemented by any class, decoupling specific functionalities from the inheritance hierarchy.

#### **Interfaces:**

* **FuelConsumable**: Defines refuel(), getFuelLevel(), consumeFuel().
* **CargoCarrier**: Defines loadCargo(), unloadCargo(), getCargoCapacity(), getCurrentCargo().
* **PassengerCarrier**: Defines boardPassengers(), disembarkPassengers(), getPassengerCapacity(), getCurrentPassengers().
* **Maintainable**: Defines scheduleMaintenance(), needsMaintenance(), performMaintenance().

#### **Classes Implementing Interfaces (implements keyword):**

* **Car**: Implements FuelConsumable, PassengerCarrier, Maintainable.
* **Truck**: Implements FuelConsumable, CargoCarrier, Maintainable.
* **Bus**: Implements FuelConsumable, PassengerCarrier, CargoCarrier, Maintainable.
* **Airplane**: Implements FuelConsumable, PassengerCarrier, CargoCarrier, Maintainable.
* **CargoShip**: Implements CargoCarrier, Maintainable, FuelConsumable.

#### **Methods demonstrating Interface usage (instanceof keyword in FleetManager):**

* **getTotalFuelConsumption(double distance)**: Uses instanceof FuelConsumable to check if a vehicle can consume fuel.
* **getVehiclesNeedingMaintenance()**: Uses instanceof Maintainable to check for maintenance needs.
* **maintainAll()**: Uses instanceof Maintainable to check before performing maintenance.
* **refuelAll(double refuelAmount)**: Uses instanceof FuelConsumable to check before refueling.
* **searchByType(Class<?> type)**: The type::isInstance method check works for both classes and interfaces, allowing searches for FuelConsumable.class, CargoCarrier.class, etc.

1. **PACKAGE AND CLASSES**

### **Package: exceptions**

This package contains all custom exception classes used for specific error conditions, making the error handling more robust and readable. All exceptions extend java.lang.Exception.

* **InsufficientFuelException.java**: Thrown by the consumeFuel method in FuelConsumable vehicles when a journey requires more fuel than is available.
* **InvalidOperationException.java**: A general-purpose exception for logically invalid actions. Thrown by:
  + move methods if distance is negative.
  + refuel if the amount is non-positive.
  + unloadCargo and disembarkPassengers if attempting to remove more than is present.
  + addVehicle in FleetManager if a vehicle ID is duplicated.
  + removeVehicle in FleetManager if a vehicle ID is not found.
* **OverloadException.java**: Thrown by loadCargo and boardPassengers methods when an attempt is made to exceed a vehicle's capacity.

### **Package: interfaces**

This package defines the contracts for modular behaviors that can be applied to any vehicle, regardless of its type.

* **CargoCarrier.java**: For vehicles that transport cargo.
* **FuelConsumable.java**: For vehicles that use fuel.
* **Maintainable.java**: For vehicles that require maintenance.
* **PassengerCarrier.java**: For vehicles that transport passengers.

### **Package: vehicles**

This package contains the entire class hierarchy for all vehicle types.

* **Vehicle.java (Abstract Class)**: The root of the hierarchy. Contains properties and methods common to all vehicles.
* **LandVehicle.java, AirVehicle.java, WaterVehicle.java (Abstract Classes)**: Intermediate classes that define category-specific logic.
* **Concrete Vehicle Classes (Car, Truck, Bus, Airplane, CargoShip)**: These are the instantiable classes that implement all abstract methods from their superclasses and all methods from their chosen interfaces.

### **Package: fleet**

This package contains the high-level logic for managing the collection of vehicles.

* **FleetManager.java**: The main controller class. It holds the List<Vehicle> and contains all the core logic for fleet operations.
* **VehicleFactory.java**: A helper class that decouples the FleetManager from the process of creating vehicles, especially when loading from a file. Its createVehicle method is a single point of responsibility for object creation from raw data.

### **Package: main**

This package contains the entry point for the application.

* **Main.java**: Contains the main method. Its sole responsibilities are to create instances of FleetManager and Scanner, run the initial demo, and manage the user interaction loop for the CLI menu.

1. HOW TO COMPILE AND RUN

Open a terminal/command prompt and navigate to the project's root directory (the one containing the src folder).

**Step 1: Compile All Java Files**

*For Windows:*

**javac -d out src\main\\*.java src\fleet\\*.java src\vehicles\\*.java src\interfaces\\*.java src\exceptions\\*.java

*For macOS / Linux:*

**javac -d out src/main/\*.java src/fleet/\*.java src/vehicles/\*.java src/interfaces/\*.java src/exceptions/\*.java

**Step 2: Run the Application**

****java -cp out main.Main

1. **HOW TO USE THE APPLICATION (CLI AND DEMO)**

### **A. Initial Demo Walkthrough**

When the application starts, it automatically performs a demonstration:

1. **Adds Vehicles**: A Car, a Truck, and an Airplane are created and added to the fleet.
2. **Simulates Journey**: A 100 km journey is simulated for all vehicles.
3. **Generates Report**: A full fleet status report is printed to the console, showing the state of the fleet after the journey.
4. **Saves to File**: The state of the demo fleet is saved to a file named demo\_fleet.csv in the project's root directory.

### **B. Command-Line Interface (CLI)**

After the demo, a menu with 11 options is displayed.

1. **Add Vehicle**: Prompts for vehicle type and all required properties, then creates and adds the new vehicle to the fleet.
2. **Remove Vehicle**: Prompts for a vehicle ID and removes the matching vehicle from the fleet.
3. **Start Journey**: Prompts for a distance (in km) and calls the move method on every vehicle in the fleet.
4. **Refuel All**: Prompts for a fuel amount and adds it to all vehicles that implement the FuelConsumable interface.
5. **Perform Maintenance**: Checks all vehicles that implement Maintainable. If a vehicle needsMaintenance(), it performs the maintenance.
6. **Generate Report**: Displays a detailed, formatted report of the current fleet status.
7. **Save Fleet**: Prompts for a filename and saves the current fleet's full state to that CSV file.
8. **Load Fleet**: Prompts for a filename, clears the current fleet, and loads a new fleet from the specified CSV file.
9. **Search by Type**: Prompts for a type (e.g., Car, LandVehicle, FuelConsumable) and displays all vehicles that match that class or interface.
10. **List Vehicles Needing Maintenance**: Displays a list of all vehicles that currently require maintenance.
11. **Exit**: Terminates the application.
12. **PERSISTENCE (SAVE/LOAD FEATURE)** The application can save and load the fleet's state to/from a CSV file. The format is designed to capture the full state of each vehicle.

* **Car Format**: Type,ID,Model,MaxSpeed,Mileage,NumWheels,FuelLevel,CurrentPassengers
* **Truck Format**: Type,ID,Model,MaxSpeed,Mileage,NumWheels,FuelLevel,CurrentCargo
* **Bus Format**: Type,ID,Model,MaxSpeed,Mileage,NumWheels,FuelLevel,CurrentPassengers, CurrentCargo
* **Airplane Format**: Type,ID,Model,MaxSpeed,Mileage,MaxAltitude,FuelLevel, CurrentPassengers,CurrentCargo
* **CargoShip Format**: Type,ID,Model,MaxSpeed,Mileage,HasSail,FuelLevel,CurrentCargo

System.out.println("\n================================ DEMO ================================");

FleetManager managerDemo = new FleetManager();

//CREATED EVERY TYPE OF SAMPLE VEHICLES

Car car= new Car("101","Toyota",50.0,40.0,4);

(car).refuel(30.0);

car.boardPassengers(3);

managerDemo.addVehicle(car);

Truck truck = new Truck("102","Volvo",100.0,30.0,6);

truck.refuel(100.0);

truck.loadCargo(200.0);

managerDemo.addVehicle(truck);

Bus bus = new Bus("103","AshokLeLand",120.0,35.0,8);

bus.refuel(100.0);

bus.boardPassengers(20);

bus.loadCargo(200.0);

managerDemo.addVehicle(bus);

Airplane airplane= new Airplane("104","Indigo",300.0,400.0,10);

airplane.refuel(300.0);

airplane.loadCargo(400.0);

airplane.boardPassengers(200);

managerDemo.addVehicle(airplane);

CargoShip cargoShip= new CargoShip("105","Titanic",200.0,300.0,true);

// cargoShip.refuel(300.0);

cargoShip.loadCargo(400.0);

managerDemo.addVehicle(cargoShip);

// REFUELLED ALL THE VEHICLES AND STARTED THE JOURNEY

managerDemo.refuelAll(10001);

managerDemo.startAllJourneys(10001);

System.out.println(managerDemo.generateReport());

// PERSISTENCE(SAVING FILE) IN DEMO.CSV

managerDemo.saveToFile("Demo.csv");

// REMOVE ALL THE CONTENT IN MANAGERDEMO FLEET

managerDemo.removeVehicle("101");

managerDemo.removeVehicle("102");

managerDemo.removeVehicle("103");

managerDemo.removeVehicle("104");

managerDemo.removeVehicle("105");

// GENERATED REPORTED

System.out.println(managerDemo.generateReport());

// LOAD ALL THE CONTENT FROM FILE DEMO.CSV

managerDemo.loadFromFile("Demo.csv");

managerDemo.getVehiclesNeedingMaintenance();

managerDemo.maintainAll();

System.out.println(managerDemo.generateReport());

System.out.println("=================================================================\n");



Demo contained in the main function shows Created sample vehicles (one of each type), add to fleet, simulate a journey (e.g., 100 km), generate report, save to file.

Included DEMO.CSV file for testing load/save.

**OUTPUT FOR DEMO :**

================================ DEMO ================================

Car with ID 101 added to the fleet.

Truck with ID 102 added to the fleet.

Bus with ID 103 added to the fleet.

Airplane with ID 104 added to the fleet.

CargoShip with ID 105 added to the fleet.

All compatible vehicles refueled.

Starting all journeys of 10001.0 km...

CAR with ID: 101 is driving for 10001.0km.

Truck ID: 102 is hauling cargo for 10001.0 km.

Bus ID: 103 is transporting passengers and cargo for 10001.0 km.

Airplane ID: 104 is flying at 10.0 feet for 10001.0 km.

CargoShip ID: 105 is sailing with cargo for 10001.0 km.

--- Fleet Status Report ---

=================================

Total Vehicles: 5

Vehicles by Type:

- CargoShip: 1

- Bus: 1

- Airplane: 1

- Car: 1

- Truck: 1

Total Fleet Mileage: 50810.00 km

Average Fuel Efficiency: 9.50 km/l

Maintenance status:

- 101 (Toyota): Needs Maintenance

- 102 (Volvo): Needs Maintenance

- 103 (AshokLeLand): Needs Maintenance

- 104 (Indigo): Needs Maintenance

- 105 (Titanic): Needs Maintenance

=================================

Fleet saved to Demo.csv

Vehicle with ID 101 removed.

Vehicle with ID 102 removed.

Vehicle with ID 103 removed.

Vehicle with ID 104 removed.

Vehicle with ID 105 removed.

The fleet is Empty.

Car with ID 101 added to the fleet.

Truck with ID 102 added to the fleet.

Bus with ID 103 added to the fleet.

Airplane with ID 104 added to the fleet.

CargoShip with ID 105 added to the fleet.

Fleet loaded successfully from Demo.csv

Performing maintenance on vehicles.

Maintenance performed on Car ID: 101

Maintenance performed on Truck ID: 102

Maintenance performed on Bus ID: 103

Maintenance performed on Airplane ID: 104

Maintenance performed on CargoShip ID: 105

--- Fleet Status Report ---

=================================

Total Vehicles: 5

Vehicles by Type:

- CargoShip: 1

- Bus: 1

- Airplane: 1

- Car: 1

- Truck: 1

Total Fleet Mileage: 50810.00 km

Average Fuel Efficiency: 9.50 km/l

Maintenance status:

- 101 (Toyota): Doesn't need Maintenance

- 102 (Volvo): Doesn't need Maintenance

- 103 (AshokLeLand): Doesn't need Maintenance

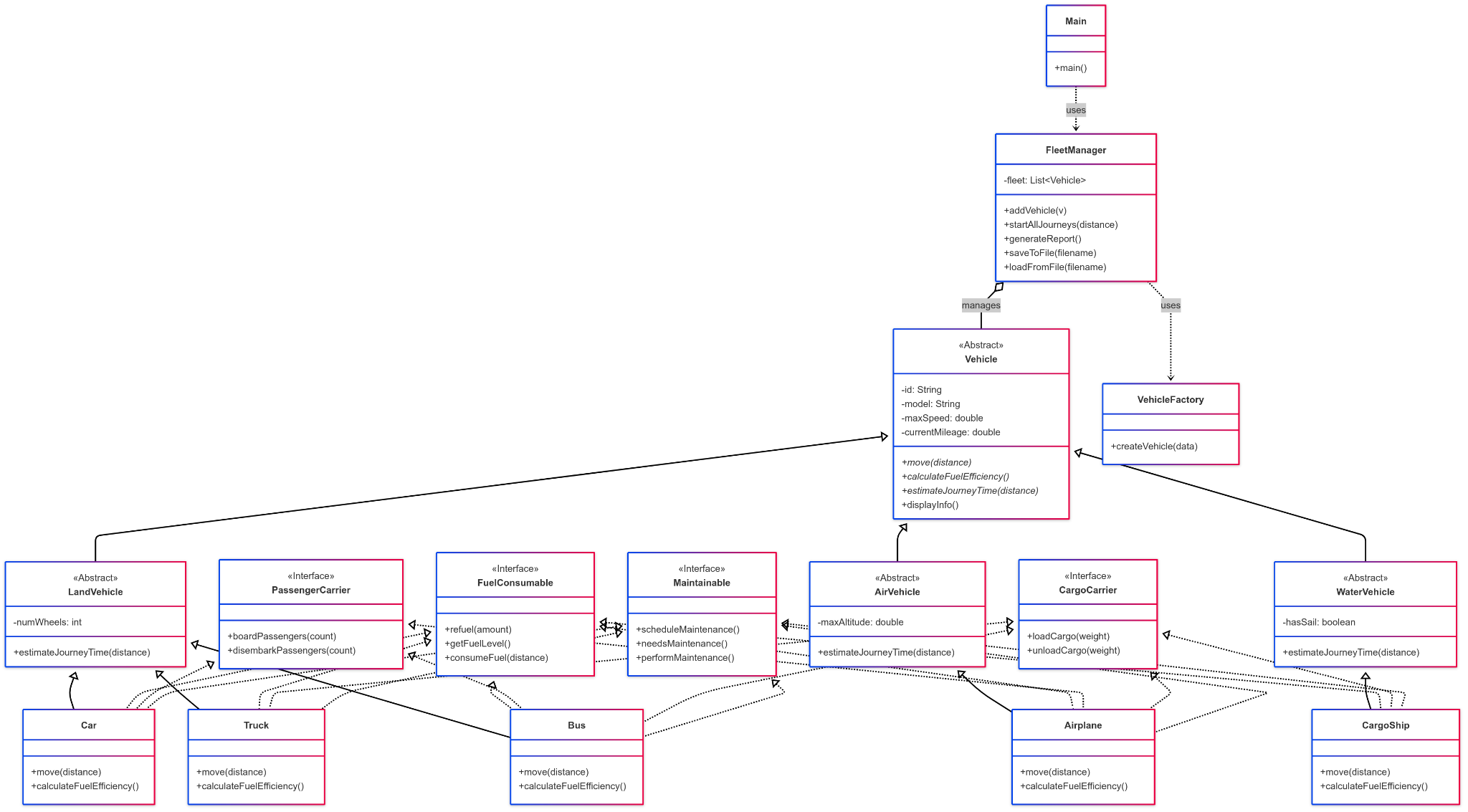
- 104 (Indigo): Doesn't need Maintenance

- 105 (Titanic): Doesn't need Maintenance

=================================



# **7. Transportation Fleet Management System - UML Diagram**



This diagram shows the complete class and interface structure of the project.

### **Key:**

* <<Abstract>>: Indicates an abstract class.
* <<Interface>>: Indicates an interface.
* **Solid Line with Hollow Arrow (<|--)**: Represents Inheritance (extends).
* **Dashed Line with Hollow Arrow (<|..)**: Represents Implementation (implements).
* **Solid Line with Open Diamond (o--)**: Represents Aggregation ("has-a" relationship).
* **Dashed Line with Arrow (..>)**: Represents Dependency ("uses-a" relationship).

### **Relationship Descriptions**

This section explains the different types of relationships shown in the diagram and how they are used in the project.

#### **1. Inheritance (Generalization)**

* **Symbol**: Solid line with a hollow arrow (<|--).
* **Meaning**: Represents an "is-a" relationship. The subclass (at the tail of the arrow) inherits properties and methods from the superclass (at the head of the arrow).
* **Examples in this project**:
  + LandVehicle, AirVehicle, and WaterVehicle **are-a** type of Vehicle.
  + Car, Truck, and Bus **are-a** type of LandVehicle.
  + Airplane **is-a** type of AirVehicle.
  + CargoShip **is-a** type of WaterVehicle.

#### **2. Implementation (Realization)**

* **Symbol**: Dashed line with a hollow arrow (<|..).
* **Meaning**: Represents an "implements" relationship. A class (at the tail of the arrow) provides concrete implementations for all the abstract methods defined in an interface (at the head of the arrow).
* **Examples in this project**:
  + Car **implements** FuelConsumable, PassengerCarrier, and Maintainable.
  + Truck **implements** FuelConsumable, CargoCarrier, and Maintainable.
  + Bus **implements** FuelConsumable, PassengerCarrier, CargoCarrier, and Maintainable.
  + Airplane **implements** FuelConsumable, PassengerCarrier, CargoCarrier, and Maintainable.
  + CargoShip **implements** CargoCarrier, Maintainable, and FuelConsumable.

#### 

#### 

#### **3. Aggregation**

* **Symbol**: Solid line with an open diamond (o--).
* **Meaning**: Represents a "has-a" relationship. It's a whole-part relationship where the "part" can exist independently of the "whole".
* **Example in this project**:
  + FleetManager o-- "1..\*" Vehicle: A FleetManager **has-a** collection of Vehicle objects. The vehicles can exist on their own, but the FleetManager is responsible for managing them as a group. The "1..\*" indicates that the FleetManager can contain one or more vehicles.

#### **4. Dependency**

* **Symbol**: Dashed line with an arrow (..>).
* **Meaning**: Represents a "uses-a" relationship. It's the weakest relationship, where a change in one class may affect the other class that uses it. The dependent class uses the other class as a parameter, a local variable, or a static method call.
* **Examples in this project**:
  + FleetManager ..> VehicleFactory: The FleetManager's loadFromFile method **uses** the VehicleFactory's static createVehicle method to create new vehicle objects.
  + Main ..> FleetManager: The Main class **uses** a FleetManager object to perform all the fleet operations requested by the user.