19CSE201 – Advanced Programming

Project Vehicle Toll System

Overview

This project implements a Toll System that calculates toll fees for vehicles based on their type, attributes, and other factors like distance, axle count, and fuel type. The system incorporates the four main principles of Object-Oriented Programming (OOP): Encapsulation, Inheritance, Polymorphism, and Abstraction.

OOP Principles Usage

1. Encapsulation

Encapsulation ensures that data within a class is hidden from unauthorized access and is only modified through defined functions. In this project:

- The 'Vehicle' class and its derived classes encapsulate vehicle-specific details (like weight, engine capacity, axle count, and registration information).
- Data members like `weight`, `axle`, and `registration` are declared as `protected` or `private` and accessed or modified through public functions like `getRegistrationState()`, `getRegistrationNumber()`, and `inputvehicledetails()`.

Example:

```
 string getPlate() const
        {
            string temp= getRegistrationState() + " " +
to_string(getRegistrationNumber());
            return temp;
        }
```

Here, the encapsulated registration information is accessed through `getRegistrationState()` and `getRegistrationNumber()` functions.

2. Inheritance

Inheritance allows derived classes to inherit properties and behaviors from a base class, enabling code reuse and hierarchy establishment. In this project:

- The `Vehicle` class serves as the base class, and specialized vehicle classes like `Single_Axle`, `Double_Axle`, `Triple_Axle`, and `Quadruple_Axle` inherit from it.
- Each derived class customizes or extends the behavior of the base class by overriding certain functions or adding new data members (e.g., `goods_weight` in `Triple_Axle` and `Quadruple_Axle`).

Example:

```
 class Single_Axle : public Vehicle

{
public:
        /*
        axle=1;
        weight= 100kg -1000kg
        engine capacity- 100cc to 1000cc;
        */
        Single_Axle() : Vehicle() // Default constructor
        {
            axle = 1;
        }

        void input() // Input details
        {
            inputvehicledetails();
        }

        int valid() override
        {
            if ((weight > 100 && weight < 1000) && (engine_capacity > 100 &&
engine_capacity < 1000))
            {
                return 1;
            }
            else
                return 0;
        }
};
```

Here, `Single_Axle` inherits common vehicle attributes and behaviors from `Vehicle` but adapts them for its specific requirements.

3. Polymorphism

Polymorphism allows objects to take on many forms, enabling functions to behave differently depending on the context. In this project:

- The `Vehicle` class declares a pure virtual function `valid()` which is overridden in each derived class to provide specific validation logic for the respective vehicle type.
- The program uses runtime polymorphism when calling `valid()` on pointers of type `Vehicle*` that point to derived class objects.

Example:

```
     virtual int valid() = 0; // Abstract function in base class
```

```
int valid() override
        {
            if ((weight > 100 && weight < 1000) && (engine_capacity > 100 &&
engine_capacity < 1000))
            {
                return 1;
            }
            else
                return 0;
        }
```

Here, the `valid()` function behaves differently for each vehicle type based on the derived class implementation.

4. Abstraction

Abstraction hides implementation details from the user and focuses on essential functionalities. In this project:

- The `Vehicle` class acts as an abstract base class with a pure virtual method `valid()`, forcing derived classes to implement their own validation logic.
- Users interact with high-level methods like `inputvehicledetails()` or `toll()` without needing to know the internal calculations or details.

Example:

```
 virtual int valid() = 0; // Abstract function ensuring derived classes implement
their specific validation logic
```

The abstract `valid()` method ensures that only the derived classes know the specific validation rules, hiding the details from the main program.

File Structure:

- Vehicle Class: Base class encapsulating common vehicle attributes and behaviors.
- Derived Classes:
  - `Single_Axle`
  - `Double_Axle`
  - `Triple_Axle`

- `Quadruple_Axle`

Each derived class implements specific behavior for different axle types.

Main Function: Manages vehicle input, validates vehicles, and calculates toll fees. It also checks if vehicles are blacklisted and writes results to the output file.

Execution Workflow

1. Input: The user provides details for each vehicle, including its type, registration plate, and city.

2. Validation: Each vehicle is validated based on its type using the `valid()` method.

3. Toll Calculation: The toll fee is computed using various factors like distance, axle count, fuel type, and payment method.

4. Blacklist Check: The program checks if the vehicle is blacklisted.

5. Output: Results are saved to `Sample.txt`, indicating the toll fee or ineligibility for each vehicle.