19CSE201 – Advanced Programming

(AM.SC.U4CYS23041)

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.

CODE:

```
// CPP PROJECT
// VEHICLE TOLL MANAGEMENT SYSTEM
// This project simulates a toll collection system for different types of vehicles
// based on their axle configuration, weight, engine capacity, and other factors.
// The system processes vehicle details, checks for blacklisted vehicles, and calculates toll
fees
// based on several criteria, including vehicle type, fuel type, payment method, and
distance.

// The 'Vehicle' class is the base class that stores common vehicle attributes like weight,
// axle count, fuel type, and registration details. It also has a method for calculating toll
fees
// based on various conditions like fuel type, vehicle weight, registration state, etc.

// The 'Single_Axle', 'Double_Axle', 'Triple_Axle', and 'Quadruple_Axle' classes are derived
// from the 'Vehicle' class, each representing vehicles with different axle counts and
specific attributes
// like weight, engine capacity, and goods weight. These classes override the
'inputvehicledetails()'
// and 'valid()' methods to handle input and validate vehicle attributes according to the
class's criteria.

// The 'load_blacklist()' function reads a list of blacklisted vehicle plates from a file
// and stores them in a vector. The 'is_blacklisted()' function checks if a given vehicle
// is in the blacklist based on its registration plate.

// MAIN PART OF PROGRAM
// The program starts by opening a file ("Sample.txt") in append mode to store results.
```

```
// A vector 'blacklist' is created to store the blacklisted vehicle plates.
// The 'load_blacklist()' function is called to read and populate the blacklist from a file.

// A vector 'vehicles' is created to hold pointers to 'Vehicle' objects.
// The user is prompted to enter the number of vehicles to be processed.
// A loop iterates over the number of vehicles entered by the user.
// The user selects the type of vehicle (Single Axle, Double Axle, etc.).

// Based on the selected type, the appropriate derived class (Single_Axle, Double_Axle, etc.)
object is created and its details are collected through the 'inputvehicledetails()' function.
// The program checks if the first two vehicles in the list are the same by comparing their
registration details using the overloaded '==' operator.

// After collecting the details for all vehicles, the program writes the toll details to a
file.
// It checks each vehicle against the blacklist using the 'is_blacklisted()' function.
// If the vehicle is blacklisted, it writes the vehicle's plate as blacklisted in the output
file.
// If the vehicle is not blacklisted and valid, it writes the vehicle's plate along with the
calculated toll fee.
// If the vehicle is not valid, it writes that the vehicle is not eligible.

// The program cleans up the dynamically allocated 'Vehicle' objects to prevent memory leaks.
// The program exits successfully with a return value of '0'.



#include <fstream>   // For file input/output operations (ifstream, ofstream, fstream)
#include <iostream>  // For standard input/output operations (cin, cout)
#include <bits/stdc++.h> // Includes all standard C++ libraries
#include <string>    // For string class and related functions
#include <algorithm> // For algorithms like sort, find, max, min, etc.

using namespace std;

// Structure to represent a date
struct Date
{
    int day;    // Stores the day of the date
    int month;  // Stores the month of the date
    int year;   // Stores the year of the date
};

// Structure to represent a vehicle plate
struct Plate
{
    string state; // Stores the state abbreviation or name
    int number;   // Stores the plate number
};
```

```
// Vehicle class: Base class for modeling vehicle details and toll calculations.
// Collects vehicle information, calculates toll fees with adjustments, and provides getters.
// Includes a pure virtual function 'valid()' which is further used in derived classes.
class Vehicle
{

protected:
        float weight;
        int axle;
        int toll_distance;
        string fuel;
        Date manufacture_date;
        Plate registration; // state of registration
        string toll_type; // one way trip/two way trip/oneday pass/monthly pass
        int engine_capacity;
        bool public_transport;
        string payment; // cash or fasttag
        float toll_fee;

public:
    //base self constructor
        Vehicle() : axle(0), toll_fee(0.0), toll_distance(0), public_transport(false) {}

         virtual void inputvehicledetails()
        {
        cout << "Enter Registration Plate: " << endl; // Prompt user for plate input
        cin >> registration.state >> registration.number; // Input state and plate number

        cout << "Enter Date of Manufacture (DD-MM-YYYY): " << endl; // Prompt for manufacture
date
        cin >> manufacture_date.day >> manufacture_date.month >> manufacture_date.year; //
Input day, month, year

        cout << "Enter Weight of the Vehicle: " << endl; // Prompt for vehicle weight
        cin >> weight; // Input weight

        cout << "Enter Fuel type (Petrol, Electric, CNG, Diesel): " << endl; // Prompt for
fuel type
        cin >> fuel; // Input fuel type

        cout << "Enter Toll Type (Single, Round, Toll Pass): " << endl; // Prompt for toll
type
        cin.ignore(); // Ignore any leftover newline character
        getline(cin, toll_type); // Use getline for toll_type to handle spaces

        cout << "Enter Engine Capacity: " << endl; // Prompt for engine capacity
```

```
        cin >> engine_capacity; // Input engine capacity

        cout << "Public Transport Vehicle? (1 for true/ 0 for false): " << endl; // Ask if
it's public transport
        cin >> public_transport; // Input true/false for public transport

        cout << "Enter Payment Type (Cash/Digital): " << endl; // Ask for payment type
        cin >> payment; // Input payment method

        toll_distance = get_toll_distance(); // Calculate toll distance based on city
        }

        // Vehicle(float weight, string city, string fuel, Date manufacture_date, string
registation, string toll_type, int engine_capacity, bool public_transport, string payment)
        // {
        //     this->weight = weight;
        //     this->city = city;
        //     this->fuel = fuel;
        //     this->manufacture_date = manufacture_date;
        //     this->registation = registration;
        //     this->toll_type = toll_type;
        //     this->engine_capacity = engine_capacity;
        //     this->public_transport = public_transport;
        //     this->payment = payment;
        // }

        // Function to get the registration state of the vehicle
        string getRegistrationState() const
        {
        return registration.state; // Return the state part of the registration plate
        }

        // Function to get the registration number of the vehicle
        int getRegistrationNumber() const
        {
        return registration.number; // Return the number part of the registration plate
        }

        // Function to return the full plate as a string
        string getPlate() const
        {
        return getRegistrationState() + " " + to_string(getRegistrationNumber()); // Combine
state and number
        }

        // Function to determine toll distance based on city selection
        int get_toll_distance()
        {
```

```
        int city; // Variable to store city choice
        cout << "Select the city: 1.Kochi 2.Trivandrum 3.Chennai 4.Ernakulam 5.Bangalore " <<
endl;

        while (true) // Loop until valid input is given
        {
            cin >> city; // Input city choice
            if (cin.fail() || city < 1 || city > 5) // Check for invalid input
            {
                cin.clear(); // Clear error state
                cin.ignore(INT_MAX, '\n'); // Ignore invalid input
                cout << "Invalid input. Please enter a number between 1 and 5: "; // Error
message
            }
            else
            {
                break; // Valid input; break the loop
            }
        }

        // Return toll distance based on the selected city
        switch (city)
        {
        case 1:
            return 108; // Kochi
        case 2:
            return 107; // Trivandrum
        case 3:
            return 718; // Chennai
        case 4:
            return 400; // Ernakulam
        case 5:
            return 1024; // Bangalore
        default:
            return -1; // Default case (should never happen due to validation)
        }
        }

        // Pure virtual function to ensure derived classes implement a `valid` function
        virtual int valid() = 0;

        // Function to calculate the toll fee
        int toll()
        {

            toll_fee = axle * 2 * toll_distance; // Base calculation

            // Weight surcharge for heavy vehicles
```

```
            if (axle >= 3)
            {
                toll_fee *= 1.1; // 10% surcharge for heavy vehicles
            }

            // Apply fuel type discounts
            if (fuel == "Electric" || fuel == "CNG")
                toll_fee *= 0.9; // 10% discount
            else if (fuel == "Petrol")
                toll_fee *= 0.95; // 5% discount

            // State registration discount
            if (registration.state == "KL")
                toll_fee *= 0.9; // 10% discount

            // Payment mode discount
            if (payment == "Digital")
                toll_fee *= 0.9; // 10% discount

            // Public transport exemption
            if (public_transport)
                toll_fee = 0; // Exempt from toll

            // Manufacture year surcharge
            if (manufacture_date.year > 2014)
                toll_fee *= 1.5; // 50% surcharge

            // Toll type adjustments
            if (toll_type == "Round")
                toll_fee *= 1.8;
            // 80% additional for round trip

            else if (toll_type == "Day Pass")
                toll_fee *= 1.2;
            // 20% additional for day pass




            return toll_fee;
         }
         // Overloaded equality operator to compare two vehicles based on their registration
details
        bool operator==(const Vehicle &other) const
        {
        return this->registration.state == other.registration.state &&
               this->registration.number == other.registration.number; // Compare state and
number
        }
```

```
};


//SINGLE AXLE VEHICLES
/*
    Single_Axle Class:
    - Derived from the base Vehicle class.
    - Models vehicles with a single axle.
    - Criteria for Single Axle vehicles:
        - Axle = 1
        - Weight = 100kg to 1000kg
        - Engine Capacity = 100cc to 1000cc
    - Overrides 'inputvehicledetails()' and 'valid()' methods.
*/
class Single_Axle : public Vehicle
{

public:
        /*
        criteria for this class
            axle=1;
            weight= 100kg -1000kg
            engine capacity- 100cc to 1000cc;
        */

        Single_Axle() : Vehicle() // Default constructor
        {
            axle = 1;// Set the axle count to 1
        }

        // overrides inputvehicledetails() fn from base class and gets input from user
        void inputvehicledetails () override
        {
            Vehicle::inputvehicledetails(); // Call base class method to input vehicle
details
        }
    /*
        Overridden valid() function:
        - Checks if the vehicle meets the criteria:
            - Weight is between 100kg and 1000kg.
            - Engine capacity is between 100cc and 1000cc.
        - Returns:
            - 1 (true) if criteria are met.
            - 0 (false) otherwise.
        */

        int valid() override
        {
```

```
        // Check if weight and engine capacity fall within the specified range
        if ((weight > 100 && weight < 1000) && (engine_capacity > 100 && engine_capacity <
1000))
        {
            return 1; // Valid vehicle
        }
        else
            return 0; // Invalid vehicle
        }
};


//DOUBLE AXLE VEHICLES
/*
    Double_Axle Class:
    - Derived from the base Vehicle class.
    - Models vehicles with two axles.
    - Criteria for Double Axle vehicles:
        - Axle = 2
        - Weight = 1000kg to 5000kg
        - Engine Capacity = 1000cc to 5000cc
    - Overrides 'inputvehicledetails()' and 'valid()' methods.
*/
class Double_Axle : public Vehicle
{

public:
        /*
        criteria for this class
            axle=2;
            weight= 1000kg - 5000kg;
            engine capacity- 1000cc to 5000cc
        */

        Double_Axle() : Vehicle() // Default constructor
        {
            axle = 2;// Set the axle count to 2
        }

    // overrides inputvehicledetails() fn from base class and gets input from user
        void inputvehicledetails () override
        {
            Vehicle::inputvehicledetails(); // Call base class method to input vehicle
details
        }


        /*
```

```
        Overridden valid() function:
        - Checks if the vehicle meets the criteria:
            - Weight is between 1000kg and 5000kg.
            - Engine capacity is between 1000cc and 5000cc.
        - Returns:
            - 1 (true) if criteria are met.
            - 0 (false) otherwise.
        */
        int valid() override
        {
        // Check if weight and engine capacity fall within the specified range
        if ((weight > 1000 && weight < 5000) && (engine_capacity > 1000 && engine_capacity <
5000))
        {
            return 1; // Valid vehicle
        }
        else
            return 0; // Invalid vehicle
    }
};


//TRIPLE AXLE VEHICLES
/*
    Triple_Axle Class:
    - Derived from the base Vehicle class.
    - Models vehicles with three axles.
    - Criteria for Triple Axle vehicles:
        - Axle = 3
        - Weight = 5000kg to 8000kg
        - Engine Capacity = 5000cc to 8000cc
        - Additional condition: Goods weight (max 10000kg) can be managed.
    - Overrides 'inputvehicledetails()' and 'valid()' methods.
*/
class Triple_Axle : public Vehicle
{

public:
        /*
        criteria for this class
            axle =3;
            weight= 5000kg - 8000kg
            engine capacity-  5000cc - 8000cc
            goods_weight - max 10000kg
        */

        Triple_Axle() : Vehicle() // Default constructor
        {
            axle = 3; // Set the axle count to 3
```

```
        }

        // we override the function to call the parent class function followed by additional
input of attribut specific to the derived class
        void inputvehicledetails() override
        {
            Vehicle::inputvehicledetails();
        }

     /*
        Overridden valid() function:
        - Checks if the vehicle meets the criteria:
            - Weight is between 5000kg and 8000kg.
            - Engine capacity is between 5000cc and 8000cc.
        - Returns:
            - 1 (true) if criteria are met.
            - 0 (false) otherwise.
    */
        int valid() override
        {
        // Check if weight and engine capacity fall within the specified range
        if ((weight > 5000 && weight < 8000) && (engine_capacity > 5000 && engine_capacity <
8000))
        {
            return 1; // Valid vehicle
        }
        else
            return 0; // Invalid vehicle
    }
};



//QUADRUPLE AXLE VEHICLES
/*
    Quadruple_Axle Class:
    - Derived from the base Vehicle class.
    - Models vehicles with four axles.
    - Criteria for Quadruple Axle vehicles:
        - Axle = 4
        - Weight = 8000kg to 10000kg
        - Engine Capacity = 8000cc+
        - Goods Weight = 10000kg to 20000kg
    - Overrides 'inputvehicledetails()' and 'valid()' methods.
*/
class Quadruple_Axle : public Vehicle
{
private:
    int goods_weight; //additional attribute applicable for quadruple vehicle
public:
```

```
        /*
        criteria for this class
            axle=4;
            weight= 8000kg to 10000kg
            engine capacity=8000cc+
            goods_weight - 10000kg to 20000kg
        */

        Quadruple_Axle() : Vehicle() // Default constructor
        {
            axle = 4;  // Set the axle count to 4
        }

        // we override the function to call the parent class function followed by additional
input of attribut specific to the derived class
    void inputvehicledetails () override
        {
            Vehicle::inputvehicledetails();// Call base class method to input vehicle details
            cout << "Weight of Goods? " << endl;
                cin >> goods_weight;// Input goods weight for Quadruple Axle vehicles
        }

        /*
        Overridden valid() function:
        - Checks if the vehicle meets the criteria:
            - Weight is between 8000kg and 10000kg.
            - Engine capacity is more than 8000cc.
            - Goods weight is between 10000kg and 20000kg.
        - Returns:
            - 1 (true) if criteria are met.
            - 0 (false) otherwise.
        */
     int valid() override
        {
        // Check if the vehicle meets all the specified criteria
        if ((weight > 8000 && weight <= 10000) && (engine_capacity > 8000) && (goods_weight
<= 20000))
        {
            return 1; // Valid vehicle
        }
        else
            return 0; // Invalid vehicle
    }
};


// function used to open and read from a file for blacklisted vehicle plates
```

```cpp
void load_blacklist(vector<string> &blacklist)
{
    // Open the "Blacklist.txt" file for reading
    ifstream blacklist_file("Blacklist.txt");

    // Check if the file was successfully opened
    if (!blacklist_file)
    {
        // Print an error message if the file could not be opened
        cerr << "Blacklist.txt could not be opened!\n";
        return; // Exit the function if the file is not opened
    }

    string entry; // Variable to hold each line read from the file
    // Read each line from the blacklist file and add it to the blacklist vector
    while (getline(blacklist_file, entry))
    {
        blacklist.push_back(entry); // Add the plate entry to the blacklist vector
    }

    // Close the file after reading
    blacklist_file.close();
}



//function used to crosscheck the blacklist file and the vehicle entries in the program

bool is_blacklisted(const vector<string> &blacklist, const string &plate)
{
    // Check if the plate is present in the blacklist using std::find
    // If the plate is found in the blacklist, return true, otherwise false
    return find(blacklist.begin(), blacklist.end(), plate) != blacklist.end();
}



// Main function analysing the vehicles for their validity and for calculating the toll
int main()
{
    //opening the file to write the entries into
    std::ofstream outf{ "Sample.txt", std::ios::app };
    //checking if file was opened successfully if not print the message
    if (!outf)
    {
        std::cerr << "Sample.txt could not be opened for writing!\n";
        return 1;
    }

    vector<string> blacklist;// Vector to hold blacklisted vehicle plates
```

```
    //calls fn that opens and reads the blacklist file
    load_blacklist(blacklist);

    vector<Vehicle *> vehicles; // Vector to hold pointers to the vehicles
    int num_vehicles; // Number of vehicles to process
    cout << "Enter the number of vehicles: "; // Prompt user to input the number of vehicles
    cin >> num_vehicles; // Read the number of vehicles from user input

    // Loop through each vehicle and create an object based on user input
    for (int i = 0; i < num_vehicles; ++i)
    {
        //here the user gets to select which derived class their vehicle belongs to
        cout << "Vehicle " << i + 1 << ":\n"; // Prompt for the vehicle number
        cout << "Select type (1: Single Axle, 2: Double Axle, 3: Triple Axle, 4: Quadruple
Axle): ";
        int type;
        cin >> type; // Get the vehicle type from user input

        Vehicle *vehicle = nullptr; // Pointer to hold the vehicle object
        // Create the appropriate vehicle object based on user selection
        switch (type)
        {
        case 1:
            vehicle = new Single_Axle(); // Create Single_Axle object
            break;
        case 2:
            vehicle = new Double_Axle(); // Create Double_Axle object
            break;
        case 3:
            vehicle = new Triple_Axle(); // Create Triple_Axle object
            break;
        case 4:
            vehicle = new Quadruple_Axle(); // Create Quadruple_Axle object
            break;
        default:
            cout << "Invalid\n"; // If an invalid choice is made, show an error message
            continue; // Skip to the next iteration of the loop
        }

        vehicle->inputvehicledetails(); // Collect the details for the current vehicle
        vehicles.push_back(vehicle); // Add the newly created vehicle to the vehicles vector
    }

    // Compare the first two vehicles in the list and check if they are the same
    if (*vehicles[0] == *vehicles[1])
    {
        //prints the below message if the vehicles are the same
```

```
        cout << "The two vehicles are the same!" << endl;
    }

    // Prints this after the successful execution of all the functions
    cout << "Toll Details Printed Successfully.\n";

     // Loop through each vehicle and process its details
    for (const auto &vehicle : vehicles)
    {
        //calls the fn that crosschecks the entries with blacklisted plates
        if (is_blacklisted(blacklist, vehicle->getPlate()))
        {
            // if blacklisted it writes to the file the below message
            outf << "Vehicle " << vehicle->getPlate() << " is Blacklisted.\n";
        }
        //here we verify if the given vehicle fits various criterias for it to actually
belong to the forementioned category
        else if (vehicle->valid())
        {
            outf << "Vehicle " << vehicle->getPlate() << " is Eligible." << "The Toll Fee of
Vehicle " << vehicle->getPlate() << " is: " << vehicle->toll() << endl;
        }
        else
        {
            // if not eligible it writes to the file the below message
            outf << "Vehicle " << vehicle->getPlate() << " Not Eligible" << endl;
        }
    }

    // cleans up the dynamicaaly allocated vehicle to prevent memeory leaks
    for (auto vehicle : vehicles)
    {
        delete vehicle;// delets vehicle object
    }

    return 0;//to indicate successful execution of program
}
```