

Task 1: Vehicle Rental System

In this project, goal is to implement The Vehicle Rental System Console Application, designed to manage the rental and return processes of various types of vehicles, including cars, motorcycles and cargo vans.

The System calculates rental and insurance costs based on specific business rules and generates an invoice for the user upon the return of the rented vehicle.

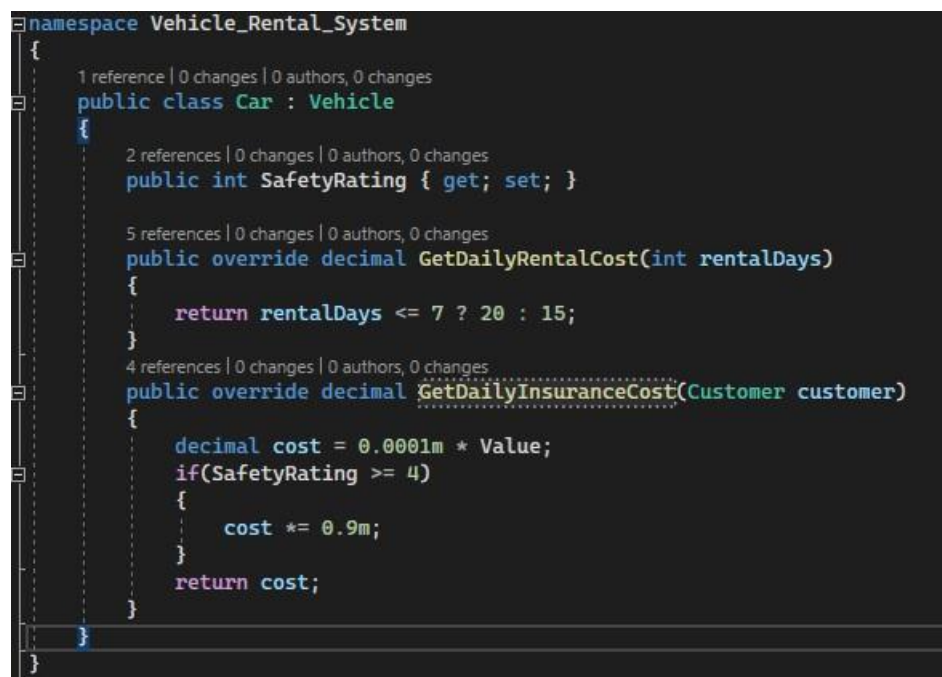
This project is great way for demonstrate **OOP Concepts** and **SOLID Principles** also.

- ***OOP Principles and Concepts***

OOP Principles and Concepts used in this Console Applications are:

1. Ecapsulation

Encapsulation involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class. It also restricts direct access to some of the object's components, which can prevent the accidental modification of data.



```
namespace Vehicle_Rental_System
{
    1 reference | 0 changes | 0 authors, 0 changes
    public class Car : Vehicle
    {
        2 references | 0 changes | 0 authors, 0 changes
        public int SafetyRating { get; set; }

        5 references | 0 changes | 0 authors, 0 changes
        public override decimal GetDailyRentalCost(int rentalDays)
        {
            return rentalDays <= 7 ? 20 : 15;
        }

        4 references | 0 changes | 0 authors, 0 changes
        public override decimal GetDailyInsuranceCost(Customer customer)
        {
            decimal cost = 0.0001m * Value;
            if(SafetyRating >= 4)
            {
                cost *= 0.9m;
            }
            return cost;
        }
    }
}
```

This is a great example of how encapsulation is used in project. The **Car** class encapsulates the attributes **Safety Rating** and methods **GetDailyRentalCost** and **GetDailyInsuranceCost**, which operate on the Car's data.

2. Abstraction

Abstraction simplifies complex system by modeling classes appropriate to the problem, reducing programming complexity. It involves creating simple, abstract representation of complex real – world entities.

```

namespace Vehicle_Rental_System
{
    6 references | 0 changes | 0 authors, 0 changes
    public abstract class Vehicle
    {
        4 references | 0 changes | 0 authors, 0 changes
        public string Brand { get; set; }
        4 references | 0 changes | 0 authors, 0 changes
        public string Model { get; set; }
        6 references | 0 changes | 0 authors, 0 changes
        public decimal Value { get; set; }
        3 references | 0 changes | 0 authors, 0 changes
        public string Type { get; set; }

        7 references | 0 changes | 0 authors, 0 changes
        public abstract decimal GetDailyRentalCost(int rentalDays);
        6 references | 0 changes | 0 authors, 0 changes
        public abstract decimal GetDailyInsuranceCost(Customer customer);
    }
}

```

In this class, the **Vehicle** class provides an abstract representation of a vehicle, with methods that must be implemented by any specific vehicle type class such as **Car**, **Motorcycle** and **CargoVan**.

3. Inheritance

Inheritance allows to inherit attributes and methods from another class, promoting code reusability and establish a natural hierarchy between classes.

```

namespace Vehicle_Rental_System
{
    6 references | 0 changes | 0 authors, 0 changes
    public abstract class Vehicle
    {
        4 references | 0 changes | 0 authors, 0 changes
        public string Brand { get; set; }
        4 references | 0 changes | 0 authors, 0 changes
        public string Model { get; set; }
        6 references | 0 changes | 0 authors, 0 changes
        public decimal Value { get; set; }
        3 references | 0 changes | 0 authors, 0 changes
        public string Type { get; set; }

        7 references | 0 changes | 0 authors, 0 changes
        public abstract decimal GetDailyRentalCost(int rentalDays);
        6 references | 0 changes | 0 authors, 0 changes
        public abstract decimal GetDailyInsuranceCost(Customer customer);
    }
}

```

The **Car** class inherits from **Vehicle**, meaning it has access to all its attributes and must implement the abstract methods **GetDailyRentalCost** and **GetDailyInsuranceCost**.

4. Polymorphism

Polymorphism allows object to be treated as instances of their parent class rather than their actual class. This can be achieved through method overriding where derived classes provide specific implementations of methods defined in their base class.

```

namespace Vehicle_Rental_System
{
    1 reference | 0 changes | 0 authors, 0 changes
    public class RentalService
    {
        1 reference | 0 changes | 0 authors, 0 changes
        public decimal CalculateRentalCost(Vehicle vehicle, int rentalDays, int actualDays)
        {
            if(actualDays < rentalDays)
            {
                return vehicle.GetDailyRentalCost(actualDays) * actualDays +
                    (vehicle.GetDailyRentalCost(rentalDays) * (rentalDays - actualDays)) / 2;
            }
            else
            {
                return vehicle.GetDailyRentalCost(actualDays) * actualDays;
            }
        }

        1 reference | 0 changes | 0 authors, 0 changes
        public decimal CalculateInsuranceCost(Vehicle vehicle, int rentalDays, int actualDays, Customer customer)
        {
            if(actualDays < rentalDays)
            {
                return vehicle.GetDailyInsuranceCost(customer) * actualDays;
            }
            else
            {
                return vehicle.GetDailyInsuranceCost(customer) * actualDays;
            }
        }
    }
}

```

In this example, the methods **CalculateRentalCost** and **CalculateInsuranceCost** can handle any object of type **Vehicle**, allowing for flexible and dynamic method calls.

- ***SOLID Principles***

This application also uses SOLID principles:

1. **Single Responsibility Principle (SRP)**

Each class should have only one reason to change, meaning it should have only one job or responsibility.

In this application, class **Rental Service** focuses only on calculating rental and insurance costs.

2. **Open/Closed Principle (OCP)**

OCP means that software entities should be open for extension but closed for modification. This means that the behavior of a module can be extended without modifying its source code.

3. **Liskov Substitution Principle (LSP)**

Subtypes must be substitutable for their base types without alerting the correctness of the program.

For example, in application, the **CalculateRentalCost** method works with any subclasses of **Vehicle** adhering to LSP.

4. **Interface Segregation Principle (ISP)**

Clients should not be forced to depend on methods they do not use. According to this, each class only implements the necessary methods.

5. Dependency Inversion Principle (DIP)

High – level modules should not depend on low – level modules. Both should depend on abstractions. Abstractions should not depend on details, details should depend on abstractions.