**ASSIGNMENT 2 FRONT SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | Unit 20: Advanced Programming | | |
| **Submission date** |  | **Date Received 1st submission** |  |
| **Re-submission Date** |  | **Date Received 2nd submission** |  |
| **Student Name** | Trinh Minh Quan | **Student ID** | GCH200025 |
| **Class** | GCH1102 | **Assessor name** | Do Hong Quan |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice. | | | |
|  |  | **Student’s signature** |  |

**Grading grid**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P3 | P4 | M3 | M4 | D3 | D4 |
|  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **❒ Summative Feedback: ❒ Resubmission Feedback:** | | |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **Lecturer Signature:** | | |

Table of Contents

[1. Introduction 5](#_Toc145622353)

[2. Scenario analysis 5](#_Toc145622354)

[2.1. Scenario 5](#_Toc145622355)

[2.2. Class Diagram 8](#_Toc145622356)

[2.3. User diagram 10](#_Toc145622357)

[3. Implementation 12](#_Toc145622358)

[3.1. Code 12](#_Toc145622359)

[3.2. Program screenshots 20](#_Toc145622360)

[4. Discussion 22](#_Toc145622361)

[4.1 Range of similar patterns 22](#_Toc145622362)

[4.1.1. Abstract Factory Pattern 22](#_Toc145622363)

[4.1.2. Prototype Pattern 25](#_Toc145622364)

[4.1.3. The reason I use design pattern builder instead of other design patterns 27](#_Toc145622365)

[5. Conclusion 27](#_Toc145622366)

[Figure 1: Class diagram 7](#_Toc144755270)

[Figure 2: User case diagram 10](#_Toc144755271)

[Figure 3: Car builder code 12](#_Toc144755272)

[Figure 4: Car builder class diagram 13](#_Toc144755273)

[Figure 5: IBuilder interface 14](#_Toc144755274)

[Figure 6: iCarBuilder 15](#_Toc144755275)

[Figure 7: Class car 17](#_Toc144755276)

[Figure 8: Class car diagram 18](#_Toc144755277)

[Figure 9: Class frame 19](#_Toc144755278)

[Figure 10: Class Engine 19](#_Toc144755279)

[Figure 11: Calss SeatBelt 19](#_Toc144755280)

[Figure 12: WIndscreen 20](#_Toc144755281)

[Figure 13: Program running 21](#_Toc144755282)

[Figure 14: Price 22](#_Toc144755283)

[Figure 15: Car information 22](#_Toc144755284)

# 1. Introduction

Through the previous exercise, I created a scenario with a class diagram and a user case diagram on the topic of building a table, which is the motivation for me to do a larger project to apply the knowledge I have learned. learn about OOP. The project is to create an application that allows users to create their own car2. Scenario analysis

# 2. Scenario analysis

## 2.1. Scenario

The company CBA is developing a smart application that allows customers to customize the components of a car according to their preferences. After customers choose the car components they want, all they need to do next is make the payment and wait for the completed car to be delivered to their doorstep. Each component of the car will have a simple description along with its price, making it user-friendly for customers who may not have extensive knowledge about vehicles. Here are the components that will be available in the car-building application:

* Frame Type:
  + Truck: 2000$
  + Sedan: 3000$
  + SUV: 3500$
  + Pickup: 4500$
  + Bug: 1500$
* Number of Wheels:
  + Customers can choose from 2 to 6 wheels for their car.
    - 2 wheels are default and balanced by the system.
    - Choosing 5 wheels means 4 wheels will be pre-installed, and 1 will be a spare.
    - 60$ per wheel
* Colors:
  + Green
  + Red
  + Purple
  + Yellow
  + Black
  + White
  + Every color is 30$
* Seat Cover:
  + Standard Leather: 20$
  + Premium Leather
    - Hermes: 100$
    - Gucci: 120$
    - Louis Vuitton: 150$
* Windscreen:
  + Tempered Glass: 500$
  + Bulletproof Glass: 700$
  + UV Resistant Glass: 600$
  + Sun Glass: 400$
* Engine Type:
  + Charging Engine: 5000$
  + Gasoline Engine: 6000$
  + Solar Engine: 7000$
  + High Power Engine: 10000$

In conclusion, CBA's smart car customization application simplifies the process of designing a car by allowing customers to select their preferred components. The application provides descriptions and prices for each component, ensuring accessibility for customers with varying levels of automotive knowledge. Whether it's choosing the frame type, the number of wheels, the color, seat upholstery, windscreen, or engine type, customers have the flexibility to create a car that suits their unique preferences and needs. This user-friendly approach to car customization streamlines the entire process, from design to delivery, making it a convenient and enjoyable experience for customers.

The application leverages the builder design pattern, which allows the systematic construction of complex objects step by step. This design pattern ensures that customers can build their cars component by component, making choices along the way, just like assembling the pieces of a puzzle. The builder pattern not only enhances the application's flexibility but also provides a structured way to create customized cars efficiently.

## 2.2. Class Diagram

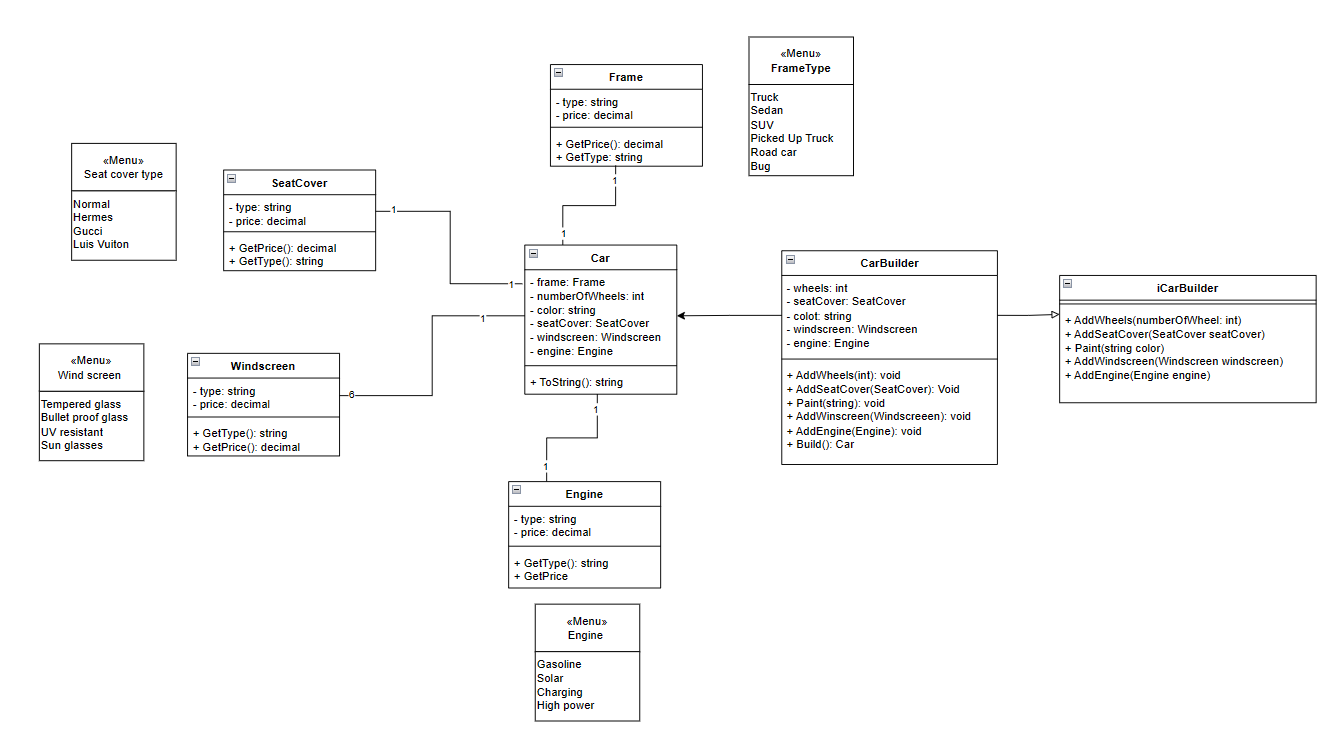


Figure : Class diagram

Explain

The class CarBuilder represents an object that has the task of building a car and it contains attributes corresponding to each part of the car such as wheels of type int, seatCover of type SeatCover, color of type string, windScreen of type WindScreen, engine of type Engine. The methods of this class allow adding components to create a complete car. The methods include AddWheels(int) returning void type that has the role of adding wheels, AddSeatCover(SeatCover) returning void type that has the function of adding seat cover, Paint(string) returning void type that has the function of adding color to the car, AddWheelsScreen(WindScreen) returning void type for users to choose the type of windshield, AddEngine(Engine) returning void type that has the role of adding engine to the car, AddFrame(Frame) returning void type that helps users choose the frame. Finally, the Build() method returns a Car type, which means it will return a complete car after adding all the components.

When the CarBuilder class implements the ICarBuilder interface, it must provide the body for all the methods defined in the ICarBuilder interface. Implementing the interface allows CarBuilder to follow the rules and interface that the interface defines. This ensures that CarBuilder has all the methods necessary to build a car. The relationship between the CarBuilder class and the iCarBuilder interface is expressed by the fact that the CarBuilder class implements the iCarBuilder interface. This means that the CarBuilder class must provide implementations for all the methods defined in the iCarBuilder interface. Thus, when an object of the CarBuilder class is created, it will be able to use the AddWheels, AddSeatCover, AddWindscreen, AddColor and AddEngine methods to add components to the car being built. We can have different classes implementing the iCarBuilder interface, each providing a different way of building a car. This allows us to create different types of cars from the same common framework. Using interfaces also helps increase the flexibility and extensibility of the code. We can add new classes implementing the iCarBuilder interface without affecting other classes that already exist. This makes code maintenance and development easier.

The Car class in this diagram represents a car. It has attributes such as frame of type Frame, numberOfWheels of type integer, color of type string, seatCover of type SeatCover, windScreen of type WindScreen and engine of type Engine to describe the components of the car. In addition, the Car class also has a ToString() method to return information about the car after it has been fully entered as a string. The types Frame, SeatCover, WindScreen, Engine are types created to represent the components of the car. They are classes that have separate attributes and methods that combine with each other to create the Car class. All these classes have a common attribute of type of type string and price of type decimal, they also have common methods of GetPrice() returning decimal type and GetType() returning string type. The reason I use decimal type is because this data type has a very high accuracy calculation ability, the price of the cars after being made is not cheap. The difference of these classes is that they have different options for users to choose from. Specifically, the Frame class has FrameType as Truck, Sedan, SUV, Picked Up Truck, Road car, Bug. The SeatCover class has types Normal, Hermes, Gucci, Luis Vuiton. The WindScreen class has types Tempered glass, Bullet proof glass, UV resistant, Sun glasses. The Engine class has types Gasoline, Solar, Charging, High power. Each type of each class has different prices that I have listed in the scenario part.

## 2.3. User diagram



Figure : User case diagram

Users can choose vehicle components including chassis, engine, seat leather, color, and windshield. In addition, customers can view information about the car they created and see the price

# 3. Implementation

## 3.1. Code

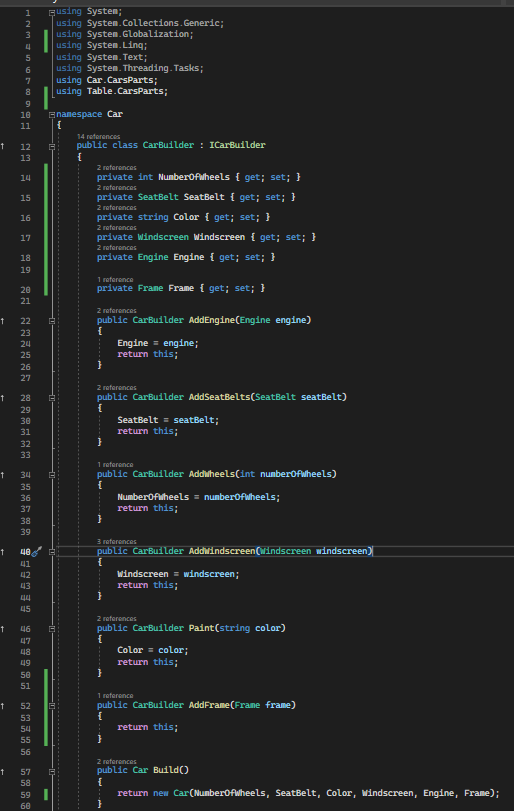


Figure : Car builder code

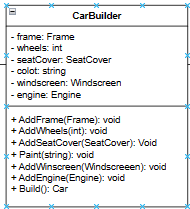


Figure : Car builder class diagram

The provided code defines the CarBuilder class, which is part of a system for constructing custom cars according to customer preferences. Here's a detailed breakdown of the code:

* private int NumberOfWheels { get; set; }: This is a private property to store the number of wheels of the car being built. This property uses auto-implemented properties (get; set;) to allow reading and setting its value from outside the class.
* private SeatBelt SeatBelt { get; set; }: Similarly, this is a property to store information about the type of seatbelt for the car. It also uses auto-implemented properties.
* private string Color { get; set; }: A property to store the color of the car. It also uses auto-implemented properties.
* private Windscreen Windscreen { get; set; }: A property to store information about the car's windshield type. It also uses auto-implemented properties.
* private Engine Engine { get; set; }: A property to store information about the car's engine type. It also uses auto-implemented properties.
* private Frame Frame { get; set; }: A property to store information about the car's frame type. It also uses auto-implemented properties.
* The following methods are also implemented in the CarBuilder class:
* public CarBuilder AddEngine(Engine engine): This method takes an Engine object, assigns it to the Engine property of the CarBuilder, and then returns the current CarBuilder object. This allows for chaining method calls.
* public CarBuilder AddSeatBelts(SeatBelt seatBelt): Similar to AddEngine, this method takes a SeatBelt object, assigns it to the SeatBelt property, and returns the current CarBuilder object.
* public CarBuilder AddWheels(int numberOfWheels): This method takes the number of wheels, assigns it to the NumberOfWheels property, and returns the current CarBuilder object.
* public CarBuilder AddWindscreen(Windscreen windscreen): Like the previous methods, this one takes a Windscreen object, assigns it to the Windscreen property, and returns the current CarBuilder object.
* public CarBuilder Paint(string color): This method takes a color represented as a string, assigns it to the Color property, and returns the current CarBuilder object.
* public Car Build(): This method is responsible for creating a complete Car object with the information set in the properties of the CarBuilder and returns the newly created Car object.

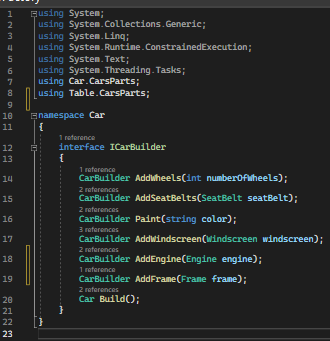


Figure : IBuilder interface

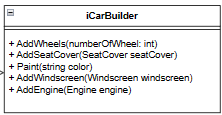


Figure : iCarBuilder

Here's a detailed explanation of the code:

* interface ICarBuilder: This line declares the interface named ICarBuilder. An interface is like a blueprint that defines a set of method signatures that must be implemented by any class that claims to implement this interface.
* CarBuilder AddWheels(int numberOfWheels);: Inside the interface, there's a method declaration called AddWheels. This method takes an integer parameter numberOfWheels and returns an instance of a class that implements the ICarBuilder interface. It suggests that any class implementing this interface should provide an implementation for adding wheels to a car.
* CarBuilder AddSeatBelts(SeatBelt seatBelt);: Similar to the previous method, this one is named AddSeatBelts and takes a SeatBelt object as a parameter. It also returns an instance of a class that implements the ICarBuilder interface. It implies that implementing classes should allow adding seatbelts to a car.
* CarBuilder Paint(string color);: This method, named Paint, takes a string parameter color and returns an instance of a class that implements the ICarBuilder interface. It suggests that implementing classes should enable painting a car with a specified color.
* CarBuilder AddWindscreen(Windscreen windscreen);: Similar to the previous methods, this one is named AddWindscreen and takes a Windscreen object as a parameter. It also returns an instance of a class that implements the ICarBuilder interface. It implies that implementing classes should allow adding a windscreen to a car.
* CarBuilder AddEngine(Engine engine);: This method, named AddEngine, takes an Engine object as a parameter and returns an instance of a class that implements the ICarBuilder interface. It suggests that implementing classes should enable adding an engine to a car.
* CarBuilder AddFrame(Frame frame);: This method, named AddFrame, takes a Frame object as a parameter and returns an instance of a class that implements the ICarBuilder interface. It implies that implementing classes should allow adding a frame to a car.
* Car Build();: Finally, there's a method named Build. It doesn't take any parameters and returns an instance of a Car class. This method indicates that any class implementing the ICarBuilder interface should provide an implementation to create and return a complete Car object, potentially using the information gathered from the previous method calls.



Figure : Class car

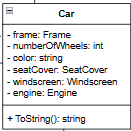


Figure : Class car diagram

Here's a detailed explanation of the code:

* public class Car: This line declares the Car class, which is used to represent a car object. Objects of this class will hold information about a car's components, such as wheels, seat belts, color, windscreen, engine, and frame.
* Inside the class, there are private properties that store information about the car's components, such as the number of wheels, seat belts, color, windscreen, engine, and frame. These properties are marked as private because they should not be directly accessible from outside the class. Instead, they are typically accessed through public methods or constructors.
* public Car(int numberOfWheels, SeatBelt seatBelt, string color, Windscreen windscreen, Engine engine, Frame frame): This is the constructor of the Car class. It takes several parameters to initialize the properties of a car object. When an instance of the Car class is created, this constructor is called to set the initial values of the car's components.
* public override string ToString(): This method overrides the ToString method from the base Object class. It returns a string representation of the Car object's properties. Inside the method, a content variable is used to build a string containing information about the car, including the number of wheels, brand of seat belts, color, windscreen brand, engine, and frame. The method then returns this content as a string.

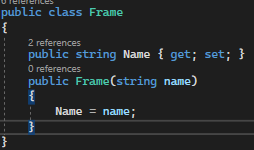


Figure : Class frame

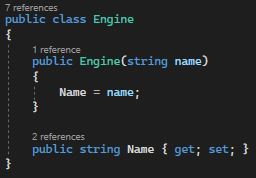


Figure : Class Engine

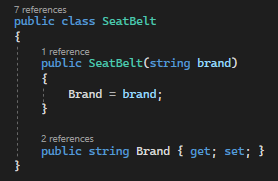


Figure : Calss SeatBelt

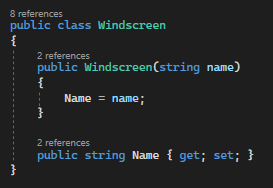


Figure : WIndscreen

The classes above all have similar constructors, including the public string Name { get; set; } property to store information about the components of the car. This property has both a getter and a setter, allowing reading and modification of the Name value. In summary, these classes are used to store the names of the car's components, enabling the creation and management of objects representing car components within the application. As for the prices of these items, they will be calculated in the main function of the program.

## 3.2. Program screenshots

This is the information I entered to create a car:

* I choose the car model is SUV
* Number of wheels is 2
* The leather upholstery is high quality with the brand name Luis Vuitton
* The type of glass is UV resistant
* Color is red
* Engine type is high power

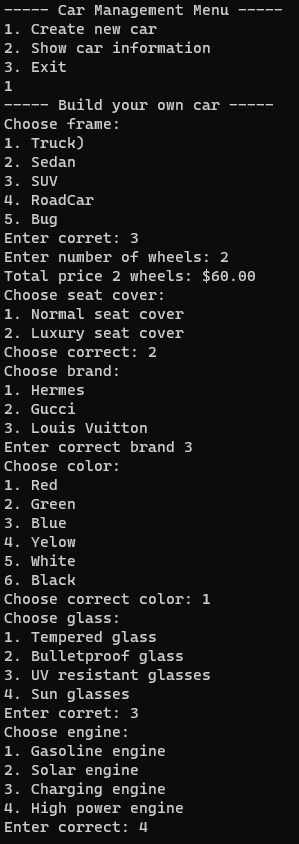


Figure : Program running

Then the price appears:



Figure : Price

After seeing the price, customers can see the information of the car

Figure : Car information

The most important function of the program is to synthesize the components together to create a car. The price has been calculated completely according to the scenario, the only error is the information of the windshield and engine. The engine and chassis do not appear in the information view, I will fix it in the next updates of the application

# 4. Discussion

## 4.1 Range of similar patterns

### 4.1.1. Abstract Factory Pattern

Similar: Both Builder and Abstract Factory are used to build objects with complex structures. Builder allows building objects step by step and customizing each component, while Abstract Factory creates a set of related or similar objects based on a specific context or product type.

Difference: The main difference is how each design achieves this goal. Builder focuses on building a single object and allows customization of each of its components, while Abstract Factory creates a collection of objects with similar or related structures.

Example:

Abstract Factory Pattern creates a set of related or similar objects (CPU, RAM, hard drive) through a specific Factory. I don't get involved in building each component one by one, you just get pre-built components from the Factory. Using the Abstract Factory Pattern, I choose a specific Factory to create the components that the Factory has defined, while ensuring consistency between components in the computer.

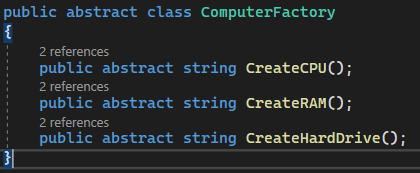


Figure : Abstract class

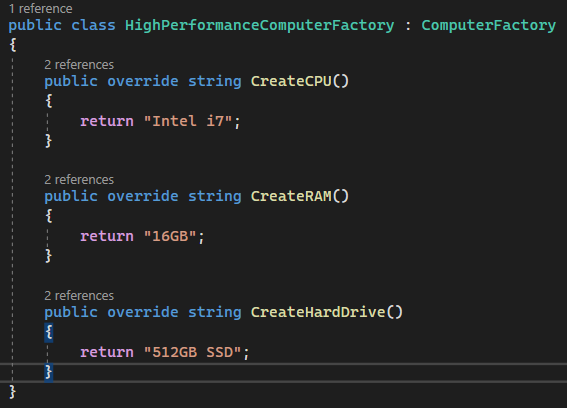


Figure : Create computer

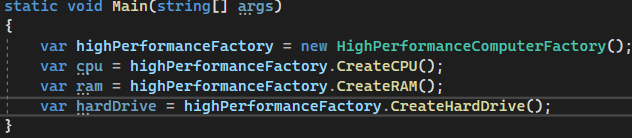


Figure : Abstract factory main

In short, both Builder and Abstract Factory are design patterns used to build complex objects, but the way they achieve this and their main goals are different. Builder focuses on building a single object, while Abstract Factory creates a collection of related or similar objects

### 4.1.2. Prototype Pattern

Similar: Both Builder and Prototype are used to create complex objects.

Difference: In Builder Pattern, you build the object from scratch and configure it as you want. In the Prototype Pattern, you create a copy of an existing object (prototype) and then modify it (if necessary) to create a new object.

Example:

Prototype Pattern creates a copy of an existing object (prototype) and then modifies it (if necessary) to create a new object. In this example, we create a copy of the original book object and adapt it to create a new book with the same properties. Using Prototype Pattern, I create a copy of the original object that I want to duplicate..

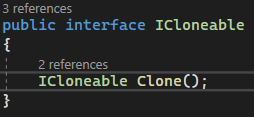


Figure : Prototype pattern interface

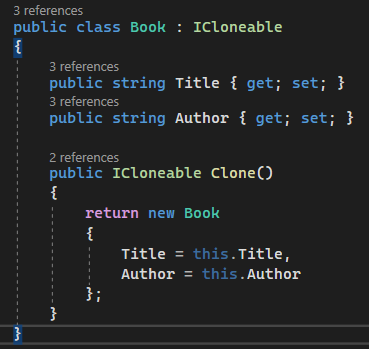


Figure : Clone object

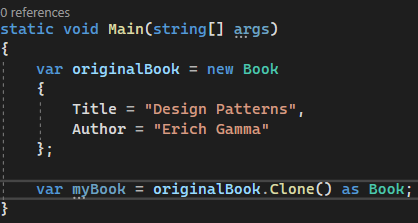


Figure :Prototype main

### 4.1.3. The reason I use design pattern builder instead of other design patterns

In the code I wrote, the Builder design pattern was the most suitable choice for building car objects with diverse and customizable properties. I also found the Abstract Factory Method to be quite suitable for this project of mine, but the Builder design pattern provided a more flexible and clear approach when I needed to build complex objects with many different properties.

Some strengths of the Builder Pattern in my code include:

* Flexibility: Builder Pattern allows me to configure each component of the vehicle independently and customize to my needs. I can set the number of wheels, motor type, color, and other components easily through Builder methods.
* Easy to read and maintain: Using the Builder Pattern makes the source code easier to read and maintain. Each part of the vehicle is configured in a separate way, reducing complexity and increasing code understandability.
* Consistency: Using the Builder Pattern ensures consistency when building car objects. I never miss an important property and can tightly control object construction.
* Create multiple variations: Builder Pattern allows me to create multiple variations of cars based on flexible combinations of attributes. This is especially useful when I have multiple vehicles or want to create custom versions for each customer.

In short, the Builder Pattern helps me build and customize car objects efficiently, while maintaining the flexibility and maintainability of the source code. It is suitable for situations when I need to build complex objects with many customizable properties in an organized and structured way.

# 5. Conclusion

Through this exercise, I have a better understanding of object-oriented programming. I will try harder to incorporate OOP into many other projects of my own.