**Scenario**:

Imagine you're managing a transportation system, and you need to represent different types of vehicles like cars and trucks. Instead of creating a new object for every single vehicle, you want to optimize memory usage by sharing common parts among similar vehicles.

**Flyweight Design Pattern Explained**:

* **Efficient Resource Usage**: The Flyweight design pattern is like a smart way to manage objects, especially when you have many objects that share common attributes. It's like reusing a set of LEGO pieces to build different structures, instead of using unique pieces for each structure.
* **Common Interface**: You start by defining a common interface, in this case, IVehicle, that all types of vehicles should follow. This is like setting a basic structure for all vehicles.
* **Concrete Implementations**: Concrete classes like Car and Truck implement the IVehicle interface. These are like specific vehicles with their own unique features.
* **Flyweight Factory**: The VehicleFactory acts like a resource manager. Instead of creating new vehicles every time, it checks if a vehicle of the same type already exists. If it does, it hands out that existing vehicle. If not, it creates a new one and keeps track of it for future use. This is like a car rental agency that gives you a car from its fleet if available.
* **Using the Factory**: In the Main method, you use the VehicleFactory to get instances of vehicles. You might notice that when you ask for the same type of vehicle multiple times, you get the same instance. This is because the factory is reusing existing instances, saving memory.

**In Simple Words**: Imagine you're playing with toy cars and trucks. You have a set of car and truck pieces that you can assemble in different ways to create different cars and trucks. You don't need to build a new car or truck from scratch each time; you reuse the same pieces to make different ones. This way, you're using your resources (pieces) efficiently.

In the code, the Flyweight pattern allows you to create instances of different vehicles without consuming too much memory. Instead of creating a new vehicle every time, you reuse existing instances if they match the requested type. This helps optimize memory usage, which is important when you have a lot of similar objects.

**Step 1: Define Shared Interface**

You start by defining an interface IVehicle that represents the common behavior of different types of vehicles. In this case, the Drive() method represents the action of driving a vehicle at a certain speed.

public interface IVehicle

{

void Drive(int speed);

}

**Step 2: Create Concrete Flyweight Classes**

You create concrete classes Car and Truck that implement the IVehicle interface. These classes represent specific types of vehicles and provide their own implementation of the Drive() method.

public class Car : IVehicle

{

public void Drive(int speed)

{

Console.WriteLine($"Driving a car at speed {speed} mph");

}

}

public class Truck : IVehicle

{

public void Drive(int speed)

{

Console.WriteLine($"Driving a truck at speed {speed} mph");

}

}

**Step 3: Flyweight Factory**

You create a VehicleFactory class that acts as a flyweight factory. It maintains a dictionary of vehicle instances (\_vehicles). When you request a vehicle of a specific type, the factory either returns an existing instance or creates a new instance and stores it in the dictionary.

public class VehicleFactory

{

private Dictionary<string, IVehicle> \_vehicles = new Dictionary<string, IVehicle>();

public IVehicle GetVehicle(string type)

{

if (!\_vehicles.TryGetValue(type, out IVehicle vehicle))

{

**// Create new vehicle instance based on type**

**// Store the instance in the dictionary**

}

return vehicle;

}

}

**Step 4: Using the Flyweight Factory**

In the Main method, you create an instance of VehicleFactory. Then, you use the factory to get instances of Car and Truck by calling GetVehicle() with different types. Finally, you call the Drive() method on each vehicle instance to simulate driving.

static void Main(string[] args)

{

VehicleFactory vehicleFactory = new VehicleFactory();

IVehicle car1 = vehicleFactory.GetVehicle("car");

IVehicle car2 = vehicleFactory.GetVehicle("car");

IVehicle truck1 = vehicleFactory.GetVehicle("truck");

IVehicle truck2 = vehicleFactory.GetVehicle("truck");

car1.Drive(60);

car2.Drive(75);

truck1.Drive(50);

truck2.Drive(65);

}

**Flyweight Design Pattern Explained**:

The Flyweight design pattern focuses on optimizing memory usage by sharing common parts of objects among multiple instances, rather than duplicating the same information in each instance.

In this example,

* IVehicle interface represents the shared interface that all concrete vehicles (Car and Truck) implement.
* The VehicleFactory class acts as the flyweight factory. It creates instances of vehicles based on the requested type and stores instances in a dictionary to reuse them.
* When you request a vehicle from the factory, it either returns an existing instance from the dictionary or creates a new instance and stores it for future use. This sharing of instances helps reduce memory usage.
* In the Main method, you demonstrate how instances of Car and Truck are reused. Multiple requests for the same vehicle type result in sharing the same instances, which is evident when the Drive() method is called on different instances of the same type.

The Flyweight pattern is beneficial when you have a large number of objects that share common properties and you want to optimize memory usage by reusing these shared properties across instances.