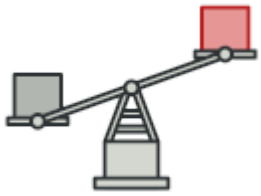

Flyweight en C#

Flyweight es un patrón de diseño estructural que permite a los programas soportar grandes cantidades de objetos manteniendo un bajo uso de memoria.

El patrón lo logra compartiendo partes del estado del objeto entre varios objetos. En otras palabras, el Flyweight ahorra memoria RAM guardando en caché la misma información utilizada por distintos objetos.

Aprende más sobre el patrón Flyweight →

Complejidad: ★★★

Popularidad: ★☆☆

Ejemplos de uso: El patrón Flyweight tiene un único propósito: minimizar el consumo de memoria. Si tu programa no tiene problemas de escasez de RAM, puedes ignorar este patrón por una temporada.

Identificación: El patrón Flyweight puede reconocerse por un método de creación que devuelve objetos guardados en caché en lugar de crear objetos nuevos.

Navegación

Program
Output

Ejemplo conceptual

Este ejemplo ilustra la estructura del patrón de diseño **Flyweight**. Se centra en responder las siguientes preguntas:

- ¿De qué clases se compone?
- ¿Qué papeles juegan esas clases?
- ¿De qué forma se relacionan los elementos del patrón?

Program.cs: Ejemplo conceptual

```
using System;
using System.Collections.Generic;
using System.Linq;
// Use Json.NET library, you can download it from NuGet Package Manager
using Newtonsoft.Json;

namespace RefactoringGuru.DesignPatterns.Flyweight.Conceptual
{
    // The Flyweight stores a common portion of the state (also called intrinsic
    // state) that belongs to multiple real business entities. The Flyweight
    // accepts the rest of the state (extrinsic state, unique for each entity)
    // via its method parameters.
    public class Flyweight
    {
        private Car _sharedState;

        public Flyweight(Car car)
        {
            this._sharedState = car;
        }

        public void Operation(Car uniqueState)
        {
            string s = JsonConvert.SerializeObject(this._sharedState);
            string u = JsonConvert.SerializeObject(uniqueState);
            Console.WriteLine($"Flyweight: Displaying shared {s} and unique {u} state.");
        }
    }
```

```
// The Flyweight Factory creates and manages the Flyweight objects. It
// ensures that flyweights are shared correctly. When the client requests a
// flyweight, the factory either returns an existing instance or creates a
// new one, if it doesn't exist yet.
public class FlyweightFactory
{
    private List<Tuple<Flyweight, string>> flyweights = new List<Tuple<Flyweight, str

    public FlyweightFactory(params Car[] args)
    {
        foreach (var elem in args)
        {
            flyweights.Add(new Tuple<Flyweight, string>(new Flyweight(elem), this.get
        }
    }

    // Returns a Flyweight's string hash for a given state.
    public string getKey(Car key)
    {
        List<string> elements = new List<string>();

        elements.Add(key.Model);
        elements.Add(key.Color);
        elements.Add(key.Company);

        if (key.Owner != null && key.Number != null)
        {
            elements.Add(key.Number);
            elements.Add(key.Owner);
        }

        elements.Sort();

        return string.Join("_", elements);
    }

    // Returns an existing Flyweight with a given state or creates a new
    // one.
    public Flyweight GetFlyweight(Car sharedState)
    {
        string key = this.getKey(sharedState);

        if (flyweights.Where(t => t.Item2 == key).Count() == 0)
        {
            Console.WriteLine("FlyweightFactory: Can't find a flyweight, creating new
            this.flyweights.Add(new Tuple<Flyweight, string>(new Flyweight(sharedStat
        }
        else
        {
            Console.WriteLine("FlyweightFactory: Reusing existing flyweight.");
        }
        return this.flyweights.Where(t => t.Item2 == key).FirstOrDefault().Item1;
```

```
        }

        public void listFlyweights()
        {
            var count = flyweights.Count;
            Console.WriteLine($"\nFlyweightFactory: I have {count} flyweights:");
            foreach (var flyweight in flyweights)
            {
                Console.WriteLine(flyweight.Item2);
            }
        }
    }

    public class Car
    {
        public string Owner { get; set; }

        public string Number { get; set; }

        public string Company { get; set; }

        public string Model { get; set; }

        public string Color { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // The client code usually creates a bunch of pre-populated
            // flyweights in the initialization stage of the application.
            var factory = new FlyweightFactory(
                new Car { Company = "Chevrolet", Model = "Camaro2018", Color = "pink" },
                new Car { Company = "Mercedes Benz", Model = "C300", Color = "black" },
                new Car { Company = "Mercedes Benz", Model = "C500", Color = "red" },
                new Car { Company = "BMW", Model = "M5", Color = "red" },
                new Car { Company = "BMW", Model = "X6", Color = "white" }
            );
            factory.listFlyweights();

            addCarToPoliceDatabase(factory, new Car {
                Number = "CL234IR",
                Owner = "James Doe",
                Company = "BMW",
                Model = "M5",
                Color = "red"
            });

            addCarToPoliceDatabase(factory, new Car {
                Number = "CL234IR",
                Owner = "James Doe",
```

```
                Company = "BMW",
                Model = "X1",
                Color = "red"
            });

            factory.listFlyweights();
        }

        public static void addCarToPoliceDatabase(FlyweightFactory factory, Car car)
        {
            Console.WriteLine("\nClient: Adding a car to database.");

            var flyweight = factory.GetFlyweight(new Car {
                Color = car.Color,
                Model = car.Model,
                Company = car.Company
            });

            // The client code either stores or calculates extrinsic state and
            // passes it to the flyweight's methods.
            flyweight.Operation(car);
        }
    }
}
```

Output.txt: Resultado de la ejecución

```
FlyweightFactory: I have 5 flyweights:
Camaro2018_Chevrolet_pink
black_C300_Mercedes Benz
C500_Mercedes Benz_red
BMW_M5_red
BMW_white_X6

Client: Adding a car to database.
FlyweightFactory: Reusing existing flyweight.
Flyweight: Displaying shared {"Owner":null,"Number":null,"Company":"BMW","Model":"M5","Cc

Client: Adding a car to database.
FlyweightFactory: Can't find a flyweight, creating new one.
Flyweight: Displaying shared {"Owner":null,"Number":null,"Company":"BMW","Model":"X1","Cc

FlyweightFactory: I have 6 flyweights:
Camaro2018_Chevrolet_pink
black_C300_Mercedes Benz
C500_Mercedes Benz_red
BMW_M5_red
```

```
BMW_white_X6
BMW_red_X1
```

LEER SIGUIENTE

Proxy en C# / Patrones de diseño →

VOLVER

← Facade en C# / Patrones de diseño

Khmelnitske shosse 19 / 27, Kamianets-Podilskyi, Ucrania, 32305
Email: support@refactoring.guru