Aula 4 - Padrões de Projeto Estruturais

Docupedia Export

Author:Belizario Marcos (CtP/ETS)
Date:18-May-2023 13:05

Table of Contents

- Flyweigth
- Padrões de Projeto Estruturais
- Facade
- Decorator
- Composite
- Exemplo: Abstração Algébrica de Funções
- Exercícios

1 Flyweigth

Flyweigth é um padrão criacional que nos permite reduzir a repetição de objetos pequenos na memória, por exemplo, exceções. Ele lembra o Singleton na sua estrutura, mas permite que o objeto existe muitas vezes. A ideia é reduzir os gastos de memória.

```
1  using static System.Console;
2  using System.Collections.Generic;
3  using static System.Diagnostics.Process;
4
5  List<MyObject> list = new List<MyObject>();
6
7  list.Add(Flyweigth.ObjectA);
8  list.Add(Flyweigth.ObjectB);
9  list.Add(Flyweigth.ObjectC);
10
11 list.Add(Flyweigth.ObjectA);
12 list.Add(Flyweigth.ObjectA);
13 list.Add(Flyweigth.ObjectB);
14 list.Add(Flyweigth.ObjectB);
15 list.Add(Flyweigth.ObjectC);
16 list.Add(Flyweigth.ObjectC);
17
18 list.Add(Flyweigth.ObjectA);
19 list.Add(Flyweigth.ObjectC);
20 list.Add(Flyweigth.ObjectC);
21 list.Add(Flyweigth.ObjectC);
22 list.Add(Flyweigth.ObjectC);
23
24 foreach (var obj in list)
25     obj.Show();
26
27 // Vê gasto de memória da aplicação
28 WriteLine(GetCurrentProcess().PrivateMemorySize64);
29
30 public static class Flyweigth
31 {
32     private static MyObjectA objA = null;
33     public static MyObjectA ObjectA
34     {
```

```
        get
        {
            if (objA == null)
                objA = new MyObjectA();
            return objA;
        }
    }

    private static MyObjectB objB = null;
    public static MyObjectB ObjectB
    {
        get
        {
            if (objB == null)
                objB = new MyObjectB();
            return objB;
        }
    }

    private static MyObjectC objC = null;
    public static MyObjectC ObjectC
    {
        get
        {
            if (objC == null)
                objC = new MyObjectC();
            return objC;
        }
    }
}

// Classe Base Opcional
public class MyObject
{
    public virtual void Show()
    {
        WriteLine("Olá, Mundo!");
    }
}
```

```
74
75  public class MyObjectA : MyObject
76  {
77      public override void Show()
78      {
79          Write("Olá,");
80      }
81  }
82
83
84  public class MyObjectB : MyObject
85  {
86      public override void Show()
87      {
88          Write(" Mun");
89
90      }
91  }
92
93  public class MyObjectC : MyObject
94  {
95      public override void Show()
96      {
97          WriteLine("do!");
98      }
99  }
```

Podemos testar e observar a pequena economia de memória já neste exemplo simples.

2 Padrões de Projeto Estruturais

Enquanto padrões criacionais nos ajudam a inicializar e criar objetos e a forma como usamos, padrões estruturais permite que criemos estrutura complexas para representar coisas complexas.

3 Facade

Facade é um padrão simples bem como o Singleton e é um dos primerios a serem aprendidos. Basicamente a ideia dele é junta várias classes e subsistemas complexos em uma única classe que contra uma ou mais operações.

```
1  using static System.Console;
2
3  Facade facade = new Facade();
4  facade.Print();
5
6
7  // Pode, muitas vezes, ser estático
8  public class Facade
9  {
10     // Usa o subsistema para fazer uma tarefa simples e desejada
11     public void Print()
12     {
13         var a = new ClassA();
14         var b = new ClassB();
15         var c = new ClassC();
16         var d = new ClassD();
17         string result = a.Get() + b.Get() + c.Get() + d.Get();
18         WriteLine(result);
19     }
20 }
21
22 // Subsistema complexo abaixo
23 public class ClassA
24 {
25     public string Get()
26     {
27         return "Olá";
28     }
29 }
30
31 public class ClassB
32 {
33     public string Get()
34     {
```

```
35          return ", ";
36      }
37  }
38
39  public class ClassC
40  {
41      public string Get()
42      {
43          return "Mundo";
44      }
45  }
46
47  public class ClassD
48  {
49      public string Get()
50      {
51          return "!";
52      }
53  }
```

4 Decorator

O Decorator permite decorar um objeto com uma implementação extra. A ideia é usar agregação (e não composição) para que um objeto possa estender o comportamento de outro. Observe:

```
1  using static System.Console;
2
3  var a = new ClassA();
4  var b = new ClassB();
5
6  var d1 = new Decorator();
7  var d2 = new Decorator();
8  var d3 = new Decorator();
9
10 d1.Wrapped = a;
11 WriteLine(d1); //Olá, Mundo!
12
13 d2.Wrapped = b;
14 WriteLine(d2); //Xispita!
15
16 d3.Wrapped = d2;
17 WriteLine(d3); //Xispita!!
18
19 public abstract class BaseClass
20 {
21     public abstract string GetString();
22
23     public override string ToString()
24         => GetString();
25 }
26
27 public class ClassA : BaseClass
28 {
29     public override string GetString()
30     {
31         return "Olá, Mundo";
32     }
33 }
34
```

```
35  public class ClassB : BaseClass
36  {
37      public override string GetString()
38      {
39          return "Xispita";
40      }
41  }
42
43  // Pode ser uma classe base para vários decorators
44  public class Decorator : BaseClass
45  {
46      public BaseClass Wrapped { get; set; }
47      public override string GetString()
48      {
49          return Wrapped.GetString() + "!";
50      }
51  }
```

5 Composite

Composite é semelhante ao decorator, mas ele agrega uma lista de componentes podendo criar uma relação hierárquica:

```
1  using static System.Console;
2  using System.Collections.Generic;
3  using System.Text;
4
5  var a = new ClassA();
6  var b = new ClassB();
7
8  var c1 = new Composite();
9  var c2 = new Composite();
10
11 c1.Add(a);
12 c1.Add(c2);
13
14 c2.Add(a);
15 c2.Add(b);
16
17 WriteLine(c1); //Olá, MundoOlá, MundoXispita
18 /*
19  *    c1
20  *   /  \
21  *  a   c2
22  *     /  \
23  *    a    b
24  */
25
26 public abstract class BaseClass
27 {
28     public abstract string GetString();
29
30     public override string ToString()
31         => GetString();
32 }
33
34 public class ClassA : BaseClass
35 {
```

```
36      public override string GetString()
37      {
38          return "Olá, Mundo";
39      }
40  }
41
42  public class ClassB : BaseClass
43  {
44      public override string GetString()
45      {
46          return "Xispita";
47      }
48  }
49
50  public class Composite : BaseClass
51  {
52      private List<BaseClass> list = new List<BaseClass>();
53      public IEnumerable<BaseClass> Classes => list;
54
55      public void Add(BaseClass obj)
56          => list.Add(obj);
57
58      public override string GetString()
59      {
60          StringBuilder sb = new StringBuilder();
61          foreach (var obj in list)
62              sb.Append(obj);
63          return sb.ToString();
64      }
65  }
```

6 Exemplo: Abstração Algébrica de Funções

Vamos agora a um exemplo prático de onde usar os padrões aprendidos no dia de hoje. Vamos implementar um sistema de abstração de funções, isto é, uma abstração em que podemos construir uma biblioteca matemática onde podemos representar funções matemáticas como:

- $f(x) = x$
- $f(x) = cos(x^2)$
- $f(x) = ln(x) + sin(x^3 + 2)$
- $f(x) = \frac{e^x + 3 * x + 1}{sin(x) + cos(x + 0.1)} - 2$

Para isso precisaremos representar várias estruturas complexas que nos ajude a controlar e utilizar dessas funções.

Function.cs (versão 1)

```
1  // Componente base da estrutura, representa uma função qualquer
2  public abstract class Function
3  {
4      // Permite usar y = f[10]
5      public double this[double x]
6          => calcule(x);
7
8      protected abstract double calcule(double x);
9
10     // Bônus para quem sabe derivar
11     public abstract Function Derive();
12 }
```

Constant.cs (versão 1)

```
1  // Componente qualquer, representa função constante, por exemplo, f(x) = 4
2  public class Constant : Function
3  {
4      private double value;
5      public Constant(double value)
6          => this.value = value;
7
8      protected override double calcule(double x) => this.value;
9
```

```
    // Bônus para quem sabe derivar
    public override Function Derive() => new Constant(0);
}
```

Linear.cs (versão 1)

```
// Componente qualquer, representa função linear f(x) = x
public class Linear : Function
{
    protected override double calcule(double x) => x;

    // Bônus para quem sabe derivar
    public override Function Derive() => new Constant(1);
}
```

Aggregation.cs

```
// Classe base para decorators
public abstract class Aggregation : Function
{
    public Function InnerFunction { get; set; }

    protected abstract double calcule(Function f, double x);

    protected override double calcule(double x)
        => calcule(InnerFunction, x);
}
```

Cosine.cs

```
using System;
// Função cosseno que decora uma função interna
public class Cosine : Aggregation
{
```

```
    protected override double calcule(Function f, double x)
        => Math.Cos(f[x]);

    public override Function Derive()
    {
        // sin(f)' = cos(f) * f'
        throw new NotImplementedException();
    }
}
```

Sine.cs

```
using System;
// Função seno que decora uma função interna
public class Sine : Aggregation
{
    protected override double calcule(Function f, double x)
        => Math.Sin(f[x]);

    public override Function Derive()
    {
        // cos(f)' = -sin(f) * f'
        throw new NotImplementedException();
    }
}
```

Composition.cs

```
using System.Collections.Generic;

// Classe base para composições de funções
public abstract class Composition : Function
{
    protected List<Function> functions = new List<Function>();
    public IEnumerable<Function> InnerFunctions => functions;

```

```
    public void Add(Function function)
        => this.functions.Add(function);
}
```

Sum.cs

```
using System.Linq;

// Composição Concreta, no caso representa soma de funções
public class Sum : Composition
{
    protected override double calcule(double x)
        => functions.Select(f => f[x]).Sum();

    public override Function Derive()
    {
        Sum sum = new Sum();

        foreach (var f in this.functions)
            sum.Add(f.Derive());

        return sum;
    }
}
```

FunctionPool.cs

```
// Flyweigth
public static class FunctionPool
{
    private static Function linearFunction = null;
    public static Function LinearFunction
    {
        get
        {
            linearFunction ??= new Linear();
```

```
            return linearFunction;
        }
    }
}
```

MathLib.cs

```
// Facade
public static class MathLib
{
    // Geralmente começamos com letras maiúsculas em membros publicos
    // usando PascalCase, aqui está uma das poucas ou únicas situações
    // onde é aceitável quebar essa regra: Convenção.
    // Na matemática é comum usar letras minúsculas para tudo, e como
    // esse seria um sistema para matemáticos, isso se torna propicio.
    public static Function x => FunctionPool.LinearFunction;
    public static Function cos(Function x)
    {
        var f = new Cosine();
        f.InnerFunction = x;
        return f;
    }
    public static Function sin(Function x)
    {
        var f = new Sine();
        f.InnerFunction = x;
        return f;
    }
}
```

Function.cs

```
public abstract class Function
{
    public double this[double x]
        => calcule(x);
```

```
5
6       protected abstract double calcule(double x);
7
8       public abstract Function Derive();
9
10      // Conversões implitas e operações personalizadas para tornar mais agradável o uso da biblioteca
11      public static implicit operator Function(double c)
12          => new Constant(c);
13
14      public static Function operator +(Function f, Function g)
15      {
16          Sum sum = new Sum();
17          sum.Add(f);
18          sum.Add(g);
19          return sum;
20      }
21  }
```

Constant.cs

```
1   // Classe melhorada graças a conversão implícita
2   public class Constant : Function
3   {
4       private double value;
5       public Constant(double value)
6           => this.value = value;
7
8       protected override double calcule(double x) => this.value;
9
10      public override Function Derive() => 0; // Mais simples!
11  }
```

Linear.cs

```
1   // Classe melhorada graças a conversão implícita
2   public class Linear : Function
```

```
{
    protected override double calcule(double x) => x;

    public override Function Derive() => 1; // Mais simples!
}
```

Program.cs

```
using static MathLib;

var f = cos(x + 3) + 1;
var y = f[0.14159265359];
Console.WriteLine(y);
```

7 Exercícios

1. Implemente a função Ln(x) (Log na base e = 2.71828182846...
2. Implemente a multiplicação de funções