BOSCH
Invented for life

Docupedia Export

Author: Goncalves Donathan (CtP/ETS)
Date: 15-May-2023 13:02

Table of Contents

1 Padrões de Projeto e o SOLID

Um Padrão de Projeto é um padrão estrutural onde usamos Orientação a Objetos entre outros aspectos da linguagens para resolver problemas específicos tentando melhorar ao máximo a qualidade de código. Hoje e nas próximas aulas aprenderemos a respeito desses padrões, qual seu objetivo e como implementá-los. Muitos desses padrões ajudam a manter o SOLID, que são 5 princípios de orientação a objetos que, em geral, trazem benefícios se mantidos. Vamos, rapidamente, compreender os 5 princípios a seguir.

2 Princípio da Responsabilidade Única

Criado por Robert Cecil Martin, o Single-Responsability Principle diz que as classes devem te apenas uma responsabilidade e não pode realizar duas funções. Mas ele vai mais afundo nisso e diz: As classes devem ter apenas um motivo para mudar. Se você tem uma classe que cria e manipula uma tela em uma aplicação ela possui dois motivos para mudar. Primeiro, caso a tela mude colocando-se novos botões ou retirando outros componentes, por exemplo. Segundo, caso você mude a tecnologia da aplicação, antes usando uma biblioteca e agora outra. Sua classe está errada pois tem 2 motivos para mudar e isso significa que você certamente terá que alterar o código por um dos motivos e pode acabar por criar bugs no outro. Com uma única responsabilidade, a classe fica muito mais robusta e resistente a erros.

3 Princípio do Aberto-Fechado

O Open-Closed Principle diz que uma classe deve estar aberta para extensão e fechada para modificação, isto é, uma classe deve poder ser utilizada por meio de agregação ou herança para ser estendida, ou seja, ter novas implementações em novas classes sem quebrar as antigas bem definidas, e, não deve ser modificada, ou seja, uma vez pronta e integrada, não devemos alterar seu código para evitar quebrar outros componentes do software. Nas palavras de Bertrand Meyer: Um componente é dito aberto se ele pode ser estendível e seu comportamento pode ser usado e aprimorado em outras classes sem mudar seu código fonte. Um componente é dito fechado se ele já está sendo usado por outros componentes do sistema, logo, não deve ter seu código alterado. Assim, todo componente do sistema deve, em algum momento, ser considerado tanto aberto quanto fechado.

4 Princípio da Substituição de Liskov

Introduzido por Barbara Liskov, o Liskov-Substitution Principle é um dos princípios mais complexos de se compreender bem. Ele diz que uma classe mãe deve poder ser substituído por uma subclasse sem quebrar o programa. Mas o que seria realmente esse quebrar? Matematicamente o princípio é escrito da seguinte forma:

$$S < T : (\forall x \in T)\phi(x) \rightarrow (\forall y \in S)\phi(y)$$

Ou seja, sendo S subclasse do tipo T, se existe uma propriedade ϕ que é verdade para todos os objetos possíveis de T, ele deve ser verdade para todos os objetos de S.

Parte de atender esse princípio está garantido pelas linguagens de programação: Contravariância nos parâmetros de funções; Covariância nos retornos de funções; Sobrescrita de métodos respeitando as assinaturas; etc. Mas também traz coisas novas, como por exemplo, se uma classe base lança um conjunto de exceções sobre algumas circunstâncias, a classe base não deve lançar nos e diferentes exceções.

Outro conflito que o princípio pode trazer é o clássico exemplo Retângulo-Quadrado. Sabemos que Quadrado é um Retângulo, mas na programação, fazer com que um Quadrado herde de um Retângulo faz com que o mesmo perca um propriedade importante: Podemos definir uma largura e altura e sua área será o produto das duas. Essa é uma propriedade fundamental que faz com que o Quadrado quebre, já que o mesmo em uma restrição mais forte: Altua e Largura devem ser iguais. Adicionalmente, Liskov nos traz os seguintes pontos que devemos cumprir para manter essa relação: Pré-condições não devem ser mais fortes na classe filha (o quadrado tem pré-condições mais fortes). Pós-condições não devem ser menos fortes na classe filha (Ao definir Largura diferente de Altura, temos a pós-condição que a Altura e Largura se mantém diferentes após o calculo da área, já no Quadrado essa pós-condição é mais fraca pois quadrado alterará uma das suas medidas para manter sua condição de quadrado).

5 Princípio da Segregação de Interface

Ao usa uma interface, se existe alguma possibilidade de que uma classe implemente apenas parte da interface, devemos considerar separar a interface em duas interfaces menores.

6 Princípio da Inversão de Dependência

O Dependecy Inversion Principle diz que abstrações de alto-nível não devem depender das implementações de baixo-nível. Isso significa que precisamos de uma camada de abstração entre as estruturas. Por exemplo, um navegador Web pode ter vários motores de renderização podendo rodar várias linguagens dele, mas ele não deve ter uma referência direta a esses motores e sim a uma interface *IRenderEngine* desconectando a classe diretamente das implementações. Isso significa que fica fácil criar novas classes que herdam de *IRenderEngine* e você não compromete o funcionamento da classe do Navegador Web.

7 Coesão e Acoplamento

Um alto grau de Coesão implica que um módulo possui apenas componentes de software que trabalham juntos afim de realizar a mesma função. Muitas vezes é confundido com Princípio da Responsabilidade Única devido ao fato de que um módulo deve ter um objetivo muito bem definido e seus componentes devem contribuir cada um com sua responsabilidade para realizar esse objetivo, como um módulo de conexão com o banco de dados, por exemplo. Deve-se buscar um alto grau de coesão, mas sem exageros.

Um atlo grau de Acoplamento significa que um módulo está fortemente ligado a outro módulo e precisa dele para funcionar. Um erro no segundo módulo gera erros no primeiro. Em geral busca-se baixo grau de acoplamento, ou seja, desacoplamento. Por exemplo, você pode perceber que o Princípio da Inversão de Dependência reduz o acoplamento do sistema uma vez que as classes começam a depender de uma abstração que pode ter implementações trocadas a qualquer momento, não tornando um módulo altamente dependente de outro.

8 Padrões de Projeto Criacionais

Padrões de Projeto Criacionais permite a criação de estrutura de forma flexível e robusta. Vamos então aprender os nossos primeiros três padrões de projeto.

9 Singleton

Talvez o padrão mais simples de todos e um dos primeiros a se aprender é o Singleton. Este padrão consiste em criar uma classe que só pode ter uma instância no sistema inteiro. Singletons em geral são mais flexíveis que classes puramente estáticas embora sejam baseadas nelas. Observe a estrutura básica:

```
1   // Usa o Singleton
2   Singleton.Current.SomeProperty = "num1";
3   Singleton.Current.OtherProperty = 4;
4   Singleton.Current.SomeMethod();
5
6   // Reinicia o Singleton
7   Singleton.New("num2");
8   // Teóricamente e possível fazer isso: Colocar o obejto singleton numa
    variável
9   var sin = Singleton.Current;
10  sin.OtherProperty = 10;
11  sin.SomeMethod();
12
13  // Em teoria podemos ter duas instâncias do objeto caso você possa criar
    uma nova instância
14  // Isso não necessariamente é ruim
15  Singleton.New("num3");
16  var newSin = Singleton.Current;
17  if (sin.SomeProperty != newSin.SomeProperty)
18      Console.WriteLine("Difernete!");
19
20  public class Singleton
21  {
22      // Construtores Privados, ou seja, ninguém pode criar uma instâcia de
    Singleton
23      // Isso força que a classe seja usada para seu design proposto:
    Existir apenas um.
24      private Singleton() { }
25
26      // Podem existir sobrecargas para incializações dentro da classe
27      private Singleton(string text)
28          => this.SomeProperty = text;
29
30      // Campo estática e privada que pode iniciar com algum valor ou com
    valor nulo.
31      private static Singleton crr = new Singleton();
32
33      // Propriedade estática publica para acessar a insância atual.
34      public static Singleton Current => crr;
35
36      // Várias Propriedades e métodos não estáticos do objeto Singleton
37      public string SomeProperty { get; set; }
38      public int OtherProperty { get; set; }
39
40      public void SomeMethod()
41          => Console.WriteLine($"{SomeProperty} = {OtherProperty}");
42
43      // Métodos estáticos que permitem recriar/manipular a instância atual
    (totalmente opcional)
44      public static void New()
45          => crr = new Singleton();
46      public static void New(string text)
```

```
47          => crr = new Singleton(text);
48  }
```

Esse padrão é muito útil quando queremos uma classe que controle uma instância muito importante da nossa aplicação. Por exemplo, em um jogo a classe Jogo que controla o Save do usuário é interessante. Singletons para conexões com banco de dados também podem ser uma opção, ou seja, existe apenas uma conexão com banco de dados embora ela possa ser recriada e conectada a outro banco.

10 Builder

O Builder é um padrão de projeto que facilita criação de objetos complexos e de forma parcelada. Ao invés de usarmos construtores gigantes e confusos podemos usar um Builder para fazer uma construção parcial, e melhor, podemos passar o Builder para outras classes que podem fazer passos da construção para nós.

```
1   // Ao se mudar o Builder, muda-se completamente o funcionamento interno e até mesmo o produto resultante
2   var builder = new ConcreteBuilderB();
3
4   // Bela sintaxe de criação de objeto
5   // Pode ser facilmente modificada com base nas necessidades da aplicação
6   var product = builder
7       .AddNumber(100)
8       .AddStirng("Texto a seguir:\n")
9       .AddManyTimes("\tTexto\n", 3)
10      .AddNumber(3)
11      .Build();
12
13  Console.WriteLine(product.Message);
14
15  // Interface que define o que o Builder precisa ter para construir o objeto, além do método Build que retorna o produto
16  public interface IBuilder
17  {
18      // Os métodos podem ou não retornar IBuilder, se retornarem possibilita sintaxes como no começo do código
19      // que são bem compactas e compreensíveis. Mas não é uma obrigação.
20      IBuilder AddNumber(int num);
21      IBuilder AddStirng(string text);
22      Product Build();
23  }
24
25  // Produto a ser construído pelo Builder
26  public class Product
27  {
28      public Product(string text)
29          => this.Message = text;
30
31      public string Message { get; set; }
32  }
33
34  // Builder Concetro, uma implementação possível do Builder
35  public class ConcreteBuilderA : IBuilder
36  {
37      private string message = string.Empty;
38      public IBuilder AddNumber(int num)
39      {
40          // Cria duas stirngs novas, x = num.Tostring() e y = message + x
41          message += num.ToString();
42          return this;
43      }
44
45      public IBuilder AddStirng(string text)
46      {
47          // Cria uma stirng nova
48          message += text;
49          return this;
```

```
50      }
51
52      public Product Build()
53          => new Product(message);
54  }
55
56  // Builder Concetro, uma implementação possível do Builder
57  // Ele é mais optimizado pois evita a criação de muitas strings criando
    apenas
58  // Uma string nova no método Build.
59  public class ConcreteBuilderB : IBuilder
60  {
61      private List<string> stirngs = new List<string>();
62      public IBuilder AddNumber(int num)
63      {
64          stirngs.Add(num.ToString());
65          return this;
66      }
67
68      public IBuilder AddStirng(string text)
69      {
70          stirngs.Add(text);
71          return this;
72      }
73
74      // Criação inteligente do produto
75      public Product Build()
76      {
77          var strSize = stirngs.Sum(s => s.Length);
78          char[] characters = new char[strSize];
79
80          int i = 0;
81          foreach (var str in stirngs)
82          {
83              foreach (var c in str)
84              {
85                  characters[i] = c;
86                  i++;
87              }
88          }
89
90          var message = new string(characters);
91          return new Product(message);
92      }
93  }
94
95  // Método de Extensão que podem adicionar novas funcionalidades ao Builder
    com base na interface
96  public static class BuilderExtension
97  {
98      public static IBuilder AddManyTimes(this IBuilder builder, string str,
    int times)
99      {
100         for (int i = 0; i < times; i++)
101             builder.AddStirng(str);
102         return builder;
103     }
104 }
```

Com o Builder podemos fazer inicializações de projetos ou até mesmo de objetos simples, como o StringBuilder do C# que funciona de forma semelhante ao *ConcreteBuilderB*, que te ajuda criar strings economizando memória.

11 Abstract Factory

O Abstract Factory é o padrão mais complexo de criação de objetos e permite criar vários produtos diferentes a partir de várias implementações de fábricas. Fábricas são objetos que retornam sempre o mesmo objeto. O que muda o comportamento do software é qual fábrica concreta foi escolhida.

```
1  // Trabalhando com um produto desconhecido
2  int data = int.Parse(Console.ReadLine() ?? "0");
3  IAbstractFactory factory = data % 2 == 0
4      ? new ConcreteFactory1()
5      : new ConcreteFactory2();
6
7  var pA = factory.CreateProductA();
8  var pB = factory.CreateProductB();
9
10
11 // Trabalhamos com Produtos Abstratos
12 public abstract class AbstractProductA
13 {
14     public float PropertyA { get; set; }
15     public abstract float MethodA();
16 }
17
18 public abstract class AbstractProductB
19 {
20     public float PropertyB { get; set; }
21     public abstract float MethodB();
22 }
23
24 // Cada Produto Abstrato pode ter muitos produtos Concretos que funcionam
   de formas diferentes
25 public class ConcreteProductA1 : AbstractProductA
26 {
27     public override float MethodA()
28         => 2 * PropertyA;
29 }
30
31 public class ConcreteProductA2 : AbstractProductA
32 {
33     public override float MethodA()
34         => PropertyA * PropertyA;
35 }
36 public class ConcreteProductB1 : AbstractProductB
37 {
38     public override float MethodB()
39         => PropertyB + 2;
40 }
41
42 public class ConcreteProductB2 : AbstractProductB
43 {
44     public override float MethodB()
45         => PropertyB - 10;
46 }
47
48 // Uma Fábrica pode criar qualquer produto mas você nunca sabe qual
49 public interface IAbstractFactory
50 {
51     AbstractProductA CreateProductA();
```

```
52      AbstractProductB CreateProductB();
53  }
54
55  // As implementações das fábricas decidem qual os produtos concretos que
    você receberá
56  public class ConcreteFactory1 : IAbstractFactory
57  {
58      public AbstractProductA CreateProductA()
59          => new ConcreteProductA1();
60
61      public AbstractProductB CreateProductB()
62          => new ConcreteProductB1();
63  }
64
65  public class ConcreteFactory2 : IAbstractFactory
66  {
67      public AbstractProductA CreateProductA()
68          => new ConcreteProductA2();
69
70      public AbstractProductB CreateProductB()
71          => new ConcreteProductB2();
72  }
```

Agora vamos a um exemplo usando os três padrões para que fique mais claro como funciona na prática.

12 Exemplo: Sistema Financerio Adptável a Diversas Leis

Considere que você abre uma empresa tanto no Brasil quanto na Argentina. Como garantir que a implementação respeite as Leis dos diferentes países sem encher de If's por todos os lados. Como permitir que os diferentes processos sejam executados de forma diferente de acordo com simples configurações básicas. Observe os códigos a seguir:

Process.cs

```
1  public abstract class Process
2  {
3      public abstract string Title { get; }
4  }
```

Employe,cs

```
1  public class Employe
2  {
3      public string Name { get; set; }
4      public decimal Wage { get; set; }
5  }
```

DismissalProcess.cs

```
1  public abstract class DismissalProcess : Process
2  {
3      public abstract void Apply(DismissalArgs args);
4  }
```

WagePaymentProcess.cs

```
1  public abstract class WagePaymentProcess : Process
2  {
3      public abstract void Apply(WagePaymentArgs args);
4  }
```

Args.cs

```
1  public class ProcessArgs
2  {
3      private static ProcessArgs empty = new ProcessArgs();
4      public static ProcessArgs Empty => empty;
5  }
6
7  public class DismissalArgs : ProcessArgs
```

```
{
    public Company Company { get; set; }
    public Employe Employe { get; set; }
}

public class WagePaymentArgs : ProcessArgs
{
    public Company Company { get; set; }
    public Employe Employe { get; set; }
}
```

IProcessFactory.cs

```
public interface IProcessFactory
{
    public WagePaymentProcess CreateWagePaymentProcess();
    public DismissalProcess CreateDismissalProcess();
}
```

Brazil.cs

```
public class BrazilDismissalProcess : DismissalProcess
{
    public override string Title => "Demissão de Funcionário";

    public override void Apply(DismissalArgs args)
    {
        args.Company.Money -= 2 * args.Employe.Wage;
    }
}

public class BrazilWagePaymentProcess : WagePaymentProcess
{
    public override string Title => "Pagamento de Salário";

    public override void Apply(WagePaymentArgs args)
    {
        args.Company.Money -= 1.45m * args.Employe.Wage + 500;
    }
}

public class BrazilProcessFactory : IProcessFactory
{
    public DismissalProcess CreateDismissalProcess()
        => new BrazilDismissalProcess();

    public WagePaymentProcess CreateWagePaymentProcess()
        => new BrazilWagePaymentProcess();
}
```

Argentina.cs

```
1  public class ArgentinaDismissalProcess : DismissalProcess
2  {
3      public override string Title => "Despido de Empleados";
4
5      public override void Apply(DismissalArgs args)
6      {
7          args.Company.Money -= 3 * args.Employe.Wage;
8      }
9  }
10
11 public class ArgentinaWagePaymentProcess : WagePaymentProcess
12 {
13     public override string Title => "Pago de salario";
14
15     public override void Apply(WagePaymentArgs args)
16     {
17         args.Company.Money -= 1.65m * args.Employe.Wage + 700;
18     }
19 }
20
21 public class ArgentinaProcessFactory : IProcessFactory
22 {
23     public DismissalProcess CreateDismissalProcess()
24         => new ArgentinaDismissalProcess();
25
26     public WagePaymentProcess CreateWagePaymentProcess()
27         => new ArgentinaWagePaymentProcess();
28 }
```

Company.cs

```
1  using System.Linq;
2  using System.Collections.Generic;
3
4  public class Company
5  {
6      // Estrutura Singleton
7      private Company() { }
8      private static Company crr = null;
9      public static Company Current => crr;
10
11     public string Name { get; set; }
12     public decimal Money { get; set; }
13
14     private List<Employe> employes = new List<Employe>();
15     public IEnumerable<Employe> Employes => employes;
16
17     private DismissalProcess dismissalProcess = null;
18     private WagePaymentProcess wagePaymentProcess = null;
19
20     public void Contract(Employe employe)
21     {
22         employes.Add(employe);
23     }
```

```
24
25      public void Dismiss(string name)
26      {
27          var employe = this.Employes.FirstOrDefault(x => x.Name == name);
28
29          if (employe == null)
30              return;
31
32          DismissalArgs args = new DismissalArgs();
33          args.Employe = employe;
34          args.Company = this;
35
36          dismissalProcess.Apply(args);
37
38          employes.Remove(employe);
39      }
40
41      public void PayWages()
42      {
43          foreach (var employe in this.Employes)
44          {
45              WagePaymentArgs args = new WagePaymentArgs();
46              args.Employe = employe;
47              args.Company = this;
48
49              wagePaymentProcess.Apply(args);
50          }
51      }
52
53      // Classes Aninhadas: Isso permite que CompanyBuilder veja todos os
    campos privados de Company
54      public class CompanyBuilder
55      {
56          private Company company = new Company();
57
58          public Company Build()
59              => this.company;
60
61          public CompanyBuilder SetName(string name)
62          {
63              company.Name = name;
64              return this;
65          }
66
67          public CompanyBuilder SetFactory(IProcessFactory factory)
68          {
69              company.dismissalProcess = factory.CreateDismissalProcess();
70              company.wagePaymentProcess =
    factory.CreateWagePaymentProcess();
71              return this;
72          }
73
74          public CompanyBuilder SetInitialCapital(decimal money)
75          {
76              company.Money = money;
77              return this;
78          }
79
80          public CompanyBuilder AddEmploye(string name, decimal wage)
81          {
```

```
            Employe employe = new Employe();
            employe.Name = name;
            employe.Wage = wage;
            company.Contract(employe);
            return this;
        }
    }

    public static CompanyBuilder GetBuilder()
        => new CompanyBuilder();

    public static void New(CompanyBuilder builder)
        => crr = builder.Build();
}
```

CompanyBuilderExtension.cs

```
public static class CompanyBuilderExtension
{
    public static Company.CompanyBuilder InBrazil(this
 Company.CompanyBuilder builder)
    {
        builder.SetFactory(new BrazilProcessFactory());
        return builder;
    }
    public static Company.CompanyBuilder InArgentina(this
 Company.CompanyBuilder builder)
    {
        builder.SetFactory(new ArgentinaProcessFactory());
        return builder;
    }
}
```

Program.cs

```
var builder = Company.GetBuilder();

builder
    .SetName("Mercado Libre")
    .InArgentina()
    .SetInitialCapital(20_000_000);

builder
    .AddEmploye("Marquitos Guapo", 50_000)
    .AddEmploye("Paulito Pino", 20_000);

Company.New(builder);

// Me rendí, me voy a Brasil
builder = Company.GetBuilder();

builder
    .SetName("Mercado Livre")
    .InBrazil()
    .SetInitialCapital(1_000_000);
```

```
21
22  builder
23      .AddEmploye("Marcos Bonito", 2_500)
24      .AddEmploye("Paulo Pinheiro", 1_000);
25
26  Company.New(builder);
27
28  Employe employe = new Employe();
29  employe.Name = "Xispita";
30  employe.Wage = 2_000;
31  Company.Current.Contract(employe);
32
33  Company.Current.Dismiss("Marcos Bonito");
34
35  Company.Current.PayWages();
```

13 Exercícios

1. Modifique a Abstração anterior para possibilitar que Empresa tenha a operação de contratação personalizável a cada país, bem como pagamento de salários e demissão.
2. Modifique a Abstração anterior para possibilitar que Empresa opere nos Estados Unidos da América.