UNIDADE 6 – HERANÇA

Nessa unidade veremos um princípio muito importante da orientação objetos. Ele permite que classes compartilhem atributos e métodos, através de "heranças". Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos.

SUMÁRIO

1 HERANÇA

1.1 REUTILIZAÇÃO DE CÓDIGO

Um banco oferece diversos serviços que podem ser contratados individualmente pelos clientes. Quando um serviço é contratado, o sistema do banco deve registrar quem foi o cliente que contratou o serviço, quem foi o funcionário responsável pelo atendimento ao cliente e a data de contratação.

Com o intuito de ser produtivo, a modelagem dos serviços do banco deve diminuir a repetição de código. A ideia é reaproveitar o máximo do código já criado. Essa ideia está diretamente relacionada ao conceito **Don't Repeat Yourself**. Em outras palavras, devemos minimizar ao máximo a utilização do "copiar e colar". O aumento da produtividade e a diminuição do custo de manutenção são as principais motivações do **DRY**.

Em seguida, vamos discutir algumas modelagens possíveis para os serviços do banco. Buscaremos seguir a ideia do DRY na criação dessas modelagens.

1.2 UMA CLASSE PARA TODOS OS SERVIÇOS

Poderíamos definir apenas uma classe para modelar todos os tipos de serviços que o banco oferece.

```
1  class Servico
2  {
3      0 references
4      public Cliente Contratante { get; set; }
5      0 references
6      public Funcionario Responsavel { get; set; }
7      0 references
8      public string DataDeContratacao { get; set; }
   }
```

Tabela 1: Servico.cs

1.2.1 Empréstimo

O empréstimo é um dos serviços que o banco oferece. Quando um cliente contrata esse serviço, são definidos o valor e a taxa de juros mensal do empréstimo. Devemos acrescentar duas propriedades na classe Servico: uma para o valor e outra para a taxa de juros do serviço de empréstimo.

```
1  class Servico
2  {
3      public Cliente Contratante { get; set; }
4      public Funcionario Responsavel { get; set; }
5      public string DataDeContratacao { get; set; }
6      public double Valor { get ; set ; }
7      public double Taxa { get ; set ; }
8  }
```

Tabela 2: Servico.cs

1.2.2 Seguro de veículos

Outro serviço oferecido pelo banco é o seguro de veículos. Para esse serviço devem ser definidas as seguintes informações: veículo segurado, valor do seguro e a franquia. Devemos adicionar três atributos na classe Servico.

```
1  class Servico
2  {
3      // GERAL
4      public Cliente Contratante { get; set; }
5      public Funcionario Responsavel { get; set; }
6      public string DataDeContratacao { get; set; }
7
9      // EMPRESTIMO
10     public double Valor { get; set; }
11     public double Taxa { get; set; }
12
13     // SEGURO DE VEICULO
14     public Veiculo Veiculo { get; set; }
15     public double ValorDoSeguroDeVeiculo { get; set; }
16 }
```

Tabela 3: Servico.cs

Apesar de seguir a ideia do DRY, modelar todos os serviços com apenas uma classe pode dificultar o desenvolvimento. Supondo que dois ou mais desenvolvedores são responsáveis pela implementação dos serviços, eles provavelmente modificariam a mesma classe concorrentemente. Além disso, os desenvolvedores, principalmente os recém chegados no projeto do banco, ficariam confusos com o código extenso da classe Servico.

Outro problema é que um objeto da classe Servico possui atributos para todos os serviços que o banco oferece. Na verdade, ele deveria possuir apenas os atributos relacionados a um serviço. Do ponto de vista de performance, essa abordagem causaria um consumo desnecessário de memória.

1.3 UMA CLASSE PARA CADA SERVIÇO

Para modelar melhor os serviços, evitando uma quantidade grande de atributos e métodos desnecessários, criaremos uma classe para cada serviço.

```
1  class SeguroDeVeiculo
2  {
3      // GERAL
4      public Cliente Contratante { get; set; }
5      public Funcionario Responsavel { get; set; }
6      public string DataDeContratacao { get; set; }
7
9      // SEGURO DE VEICULO
10     public Veiculo Veiculo { get; set; }
11     public double ValorDoSeguroDeVeiculo { get; set; }
12
       public double Franquia { get; set; }
   }
```

Tabela 4: SeguroDeVeiculo.cs

```
1  class Emprestimo
2  {
3      // GERAL
4      public Cliente Contratante { get; set; }
5      public Funcionario Responsavel { get; set; }
6      public string DataDeContratacao { get; set; }
7
9      // EMPRESTIMO
10     public double Valor { get; set; }
11     public double Taxa { get; set; }
   }
```

Tabela 5: Emprestimo.cs

Criar uma classe para cada serviço torna o sistema mais flexível, pois qualquer alteração em um determinado serviço não causará efeitos colaterais nos outros. Mas, por outro lado, essas classes teriam bastante código repetido, contrariando a ideia do DRY. Além disso, qualquer alteração que deva ser realizada em todos os serviços precisa ser implementada em cada uma das classes.

1.4 UMA CLASSE GENÉRICA E VÁRIAS ESPECÍFICAS

Na modelagem dos serviços do banco, podemos aplicar um conceito de orientação a objetos chamado Herança. A ideia é reutilizar o código de uma determinada classe em outras classes.

Aplicando herança, teríamos a classe Servico com os atributos e métodos que todos os serviços devem ter e uma classe para cada serviço com os atributos e métodos específicos do determinado serviço.

As classes específicas seriam "ligadas" de alguma forma à classe Servico para reaproveitar o código nela definido. Esse relacionamento entre as classes é representado em UML pelo diagrama abaixo.

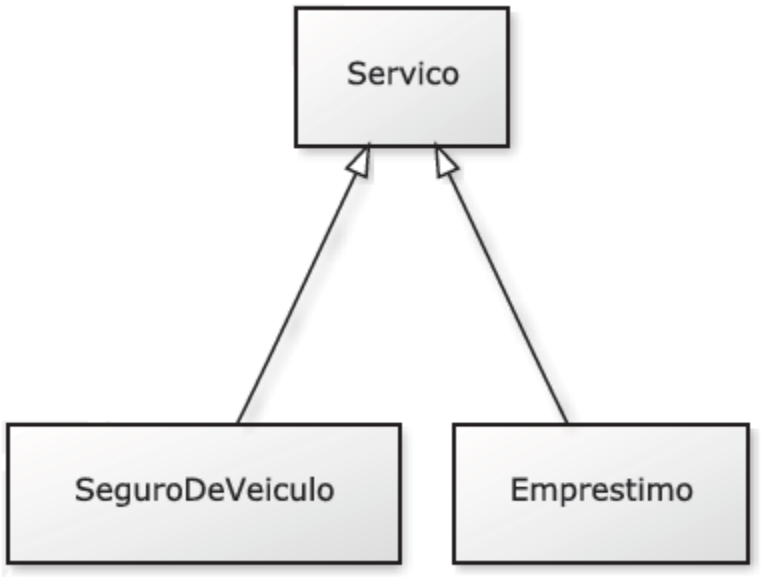

Figura 1: Árvore de herança dos serviços

Os objetos das classes específicas Em prestimo e SeguroDeVeiculo possuiriam tanto os atributos e métodos definidos nessas classes quanto os definidos na classe Servico.

```
1  Emprestimo e = new Emprestimo();
2
3  // Chamando um método da classe Servico
4  e.DataDeContratacao = " 10/10/2010 ";
5
6  // Chamando um método da classe Emprestimo
7  e.Valor = 10000;
```

Tabela 6: Chamando métodos da classe genérica e da específica

As classes específicas são vinculadas a classe genérica utilizando o comando (:). Não é necessário redefinir o conteúdo já declarado na classe genérica.

```
1  class Servico
2  {
3      public Cliente Contratante { get; set; }
4      public Funcionario Responsavel { get; set; }
5      public string DataDeContratacao { get; set; }
6  }
```

Tabela 6: Servico.cs

```
1  class Emprestimo : Servico
2  {
3      public double Valor { get; set; }
4      public double Taxa { get; set; }
5  }
```

Tabela 8: Emprestimo.cs

```
1  class SeguroDeVeiculo : Servico
2  {
3      public Veiculo Veiculo { get; set; }
4      public double ValorDoSeguroDeVeiculo { get; set; }
5      public double Franquia { get; set; }
6  }
```

Tabela 9: SeguroDeVeiculo

A classe genérica é denominada **super classe**, **classe base** ou **classe mãe**. As classes específicas são denominadas **sub classes**, **classes derivadas** ou **classes filhas**.

Quando o operador new é aplicado em uma sub classe, o objeto construído possuirá os atributos e métodos definidos na sub classe e na super classe.

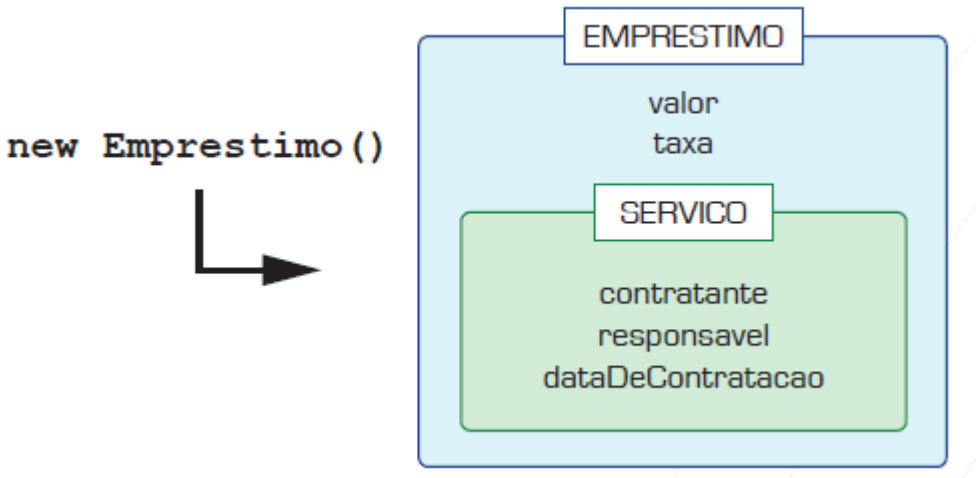

Figura 2: Criando um objeto a partir da sub classe

1.5 Preço Fixo

Suponha que todo serviço do banco possui uma taxa administrativa que deve ser paga pelo cliente que contratar o serviço. Inicialmente, vamos considerar que o valor dessa taxa é igual para todos os serviços do banco.

Neste caso, poderíamos implementar um método na classe Servico para calcular o valor da taxa. Este método será reaproveitado por todas as classes que herdam da classe Servico.

```
1  class Servico
2  {
3      // propriedades
4
5      public double CalculaTaxa()
6      {
7          return 10;
8      }
9  }
```

Tabela 10: Servico.cs

```
1  Emprestimo e = new Emprestimo();
2
3  SeguroDeVeiculo sdv = new SeguroDeVeiculo();
4
5  Console.WriteLine("Emprestimo : " + e.CalculaTaxa());
6
7  Console.WriteLine("SeguroDeVeiculo : " + sdv.CalculaTaxa());
```

Tabela 11: Chamando o método CalculaTaxa()

1.6 Reescrita de Método

Suponha que o valor da taxa administrativa do serviço de empréstimo é diferente dos outros serviços, pois ele é calculado a partir do valor emprestado ao cliente. Como esta lógica é específica para o serviço de empréstimo, devemos acrescentar um método para implementar esse cálculo na classe Emprestimo.

```
1  class Emprestimo : Servico
2  {
3      //propriedades
4
5      public double CalculaTaxaDeEmprestimo()
6      {
7          return this.Valor*0.1;
8      }
9  }
```

Tabela 12: Servico.cs

Para os objetos da classe `Emprestimo`, devemos chamar o método `CalculaTaxaDeEmprestimo`(). Para todos os outros serviços, devemos chamar o método `CalculaTaxa`().

Mesmo assim, nada impediria que o método `CalculaTaxa`() fosse chamado em um objeto da classe `Emprestimo`, pois ela herda esse método da classe `Servico`. Dessa forma, existe o risco de alguém erroneamente chamar o método incorreto.

Seria mais seguro "substituir" a implementação do método `CalculaTaxa`() herdado da classe `Servico` na classe `Emprestimo`. Por padrão, as implementações dos métodos de uma superclasse não podem ser substituídas pelas subclasses. Para alterar esse padrão, devemos acrescentar o modificador **virtual**.

```
1  class Servico
2  {
3      //propriedades
4
5      public virtual double CalculaTaxa()
6      {
7          return 10;
8      }
9  }
```

Tabela 13: Servico.cs

Depois que a classe mãe `Servico` autorizou a substituição da implementação do método `CalculaTaxa` através do modificador virtual, basta reescrever o método `CalculaTaxa()` na classe `Emprestimo` com a mesma assinatura que ele possui na classe `Servico` e com o modificador **override**.

```
1  class Emprestimo : Servico
2  {
3      public double Valor { get; set; }
4      public double Taxa { get; set; }
5
6      public override double CalculaTaxa()
7      {
8          return this.Valor*0.1;
9      }
10 }
```

Tabela 14: Emprestimo.cs

Os métodos das classes específicas têm prioridade sobre os métodos das classes genéricas. Em outras palavras, se o método chamado existe na classe filha ele será chamado, caso contrário o método será procurado na classe mãe.

Quando definimos um método com a mesma assinatura na classe base e em alguma classe derivada, estamos aplicando o conceito de Reescrita de Método.

1.7 Fixo + Específico

Suponha que o preço de um serviço é a soma de um valor fixo mais um valor que depende do tipo do serviço. Por exemplo, o preço do serviço de empréstimo é 5 reais mais uma porcentagem do valor emprestado ao cliente. O preço do serviço de seguro de veículo é 5 reais mais uma porcentagem do valor do veículo segurado. Em cada classe específica, podemos reescrever o método `CalculaTaxa()`.

```
1  class Emprestimo : Servico
2  {
3      public double Valor { get; set; }
4      public double Taxa { get; set; }
5
6      public override double CalculaTaxa()
7      {
8          return 5 + this.Valor * 0.1;
9      }
10 }
```

Tabela 15: Emprestimo.cs

```
1  class SeguroDeVeiculo : Servico
2  {
3      public Veiculo Veiculo { get; set; }
4      public double ValorDoSeguroDeVeiculo { get; set; }
5      public double Franquia { get; set; }
6
7      public override double CalculaTaxa()
8      {
9          return 5 + this.Veiculo.Valor * 0.05;
       }
   }
```

Tabela 16: SeguraDeVeiculo.cs

Se o valor fixo dos serviços for atualizado, todas as classes específicas devem ser modificadas. Outra alternativa seria criar um método na classe Servico para calcular o valor fixo de todos os serviços e chamá-lo dos métodos reescritos nas classes específicas.

```
1  class Servico
2  {
3      //propriedades
4
5      public virtual double CalculaTaxa()
6      {
7          return 5;
8      }
9  }
```

Tabela 17: Servico.cs

```
1  class Emprestimo : Servico
2  {
3      public double Valor { get; set; }
4      public double Taxa { get; set; }
5
6      public override double CalculaTaxa()
7      {
8          return base.CalculaTaxa() + this.Valor * 0.1;
       }
9  }
```

Tabela 18: Emprestimo.cs

Dessa forma, quando o valor padrão do preço dos serviços é alterado, basta modificar o método na classe Servico.

1.8 CONSTRUTORES E HERANÇA

Quando temos uma hierarquia de classes, as chamadas dos construtores são mais complexas do que o normal. Pelo menos um construtor de cada classe de uma mesma sequência hierárquica deve ser chamado ao instanciar um objeto. Por exemplo, quando um objeto da classe `Emprestimo` é criado, pelo menos um construtor da própria classe `Emprestimo` e um da classe `Servico` devem ser executados. Além disso, os construtores das classes mais genéricas são chamados antes dos construtores das classes específicas.

```
1  class Servico
2  {
3      //propriedades
4
5      public Servico()
6      {
7          Console.WriteLine("Servico");
8      }
9  }
```

Tabela 19: Servico.cs

```
1  class Emprestimo : Servico
2  {
3      //propriedades
4
5      public Emprestimo()
6      {
7          Console.WriteLine("Emprestimo");
8      }
9  }
```

Tabela 20: Emprestimo.cs

Por padrão, todo construtor chama o construtor sem argumentos da classe mãe se não existir nenhuma chamada de construtor explícita.

1.9 Exercícios de Fixação

1) Crie um projeto no chamado Unidade 6.

2) Defina uma classe para modelar os funcionários do banco. Sabendo que todo funcionário possui nome e salário, inclua as propriedades dos atributos.

```
1  public class Funcionario
2  {
3      public string Nome { get; set; }
4      public double Salario { get; set; }
5  }
```

Tabela 21: Funcionario.cs

3) Crie uma classe para cada tipo específico de funcionário herdando da classe Funcionario. Considere apenas três tipos específicos de funcionários: gerentes, telefonistas e secretarias. Os gerentes possuem um nome de usuário e uma senha para acessar o sistema do banco. As telefonistas possuem um código de estação de trabalho. As secretarias possuem um número de ramal.

```
1  public class Gerente : Funcionario
2  {
3      public string Usuario { get; set; }
4      public string Senha { get; set; }
   }
```

Tabela 22: Gerente.cs

```
1  class Telefonista : Funcionario
2  {
3      public int EstacaoDeTrabalho { get; set; }
4  }
```

Tabela 23: Telefonista.cs

```
1  class Secretaria : Funcionario
2  {
3      public int Ramal { get; set; }
4  }
```

Tabela 24: Secretaria.cs

4) Teste o funcionamento dos três tipos de funcionários criando um objeto de cada uma das classes: `Gerente, Telefonista e Secretaria.`

```
1  class TestaFuncionarios
2  {
3      private static void Main()
4      {
5          Gerente g = new Gerente();
6          g.Nome = " Rafael Cosentino ";
7          g.Salario = 2000;
8          g.Usuario = " rafael.cosentino ";
9          g.Senha = " 12345 ";
10
11         Telefonista t = new Telefonista();
12         t.Nome = " Carolina Mello ";
13         t.Salario = 1000;
14         t.EstacaoDeTrabalho = 13;
15
16         Secretaria s = new Secretaria();
17         s.Nome = " Tatiane Andrade ";
18         s.Salario = 1500;
19         s.Ramal = 198;
20
21         Console.WriteLine(" GERENTE ");
22         Console.WriteLine(" Nome : " + g.Nome);
23         Console.WriteLine(" Salário : " + g.Salario);
24         Console.WriteLine(" Usuário : " + g.Usuario);
25         Console.WriteLine(" Senha : " + g.Senha);
26
27         Console.WriteLine(" TELEFONISTA ");
28         Console.WriteLine(" Nome : " + t.Nome);
29         Console.WriteLine(" Salário : " + t.Salario);
30         Console.WriteLine(" Estacao de trabalho : " + t.EstacaoDeTrabalho);
31
32         Console.WriteLine(" SECRETARIA ");
33         Console.WriteLine(" Nome : " + s.Nome);
34         Console.WriteLine(" Salário : " + s.Salario);
35         Console.WriteLine(" Ramal : " + s.Ramal);
36     }
37 }
```

Tabela 25: TestaFuncionarios.cs

5) Suponha que todos os funcionários recebam uma bonificação de 10% do salário. Acrescente um método na classe Funcionario para calcular essa bonificação.

```
1  class Funcionario
2  {
3      public string Nome { get; set; }
4      public double Salario { get; set; }
5
6      public double CalculaBonificacao()
7      {
8          return this.Salario*0.1;
9      }
   }
```

Tabela 26: Funcionario.cs

6) Altere a classe `TestaFuncionarios` para imprimir a bonificação de cada funcionário, além dos dados que já foram impressos. Depois, execute o teste novamente.

```
1  class TestaFuncionarios
2  {
3      private static void Main()
4      {
5          Gerente g = new Gerente();
6          g.Nome = " Rafael Cosentino ";
7          g.Salario = 2000;
8          g.Usuario = " rafael.cosentino ";
9          g.Senha = " 12345 ";
10
11         Telefonista t = new Telefonista();
12         t.Nome = " Carolina Mello ";
13         t.Salario = 1000;
14         t.EstacaoDeTrabalho = 13;
15
16         Secretaria s = new Secretaria();
17         s.Nome = " Tatiane Andrade ";
18         s.Salario = 1500;
19         s.Ramal = 198;
20
21         Console.WriteLine(" GERENTE ");
22         Console.WriteLine(" Nome : " + g.Nome);
           Console.WriteLine(" Salário : " + g.Salario);
           Console.WriteLine(" Usuário : " + g.Usuario);
           Console.WriteLine(" Senha : " + g.Senha);
           Console.WriteLine(" Bonificação : " + g.CalculaBonificacao());
23
24         Console.WriteLine(" TELEFONISTA ");
25         Console.WriteLine(" Nome : " + t.Nome);
26         Console.WriteLine(" Salário : " + t.Salario);
27         Console.WriteLine(" Estacao de trabalho : " + t.EstacaoDeTrabalho);
28         Console.WriteLine(" Bonificação : " + t.CalculaBonificacao());
29
           Console.WriteLine(" SECRETARIA ");
           Console.WriteLine(" Nome : " + s.Nome);
           Console.WriteLine(" Salário : " + s.Salario);
           Console.WriteLine(" Ramal : " + s.Ramal);
           Console.WriteLine(" Bonificação : " + s.CalculaBonificacao());
       }
```

```
30
31
32
33
34
35
36
```

Tabela 27: TestaFuncionarios.cs

7) Suponha que os gerentes recebam uma bonificação maior que os outros funcionários. Reescreva o método CalculaBonificacao() na classe Gerente. Porém, devemos permitir que as classes filhas possam reescrever o método e para tal precisamos alterá-lo na classe `Funcionario` acrescentando o modificador virtual.

```
1  class Funcionario
2  {
3      public string Nome { get; set; }
4      public double Salario { get; set; }
5
6      public virtual double CalculaBonificacao()
7      {
8          return this.Salario * 0.1;
9      }
10 }
```

Tabela 28: Funcionarios.cs

Reescreva o método CalculaBonificao() e execute o teste novamente.

```
1  class Gerente : Funcionario
2  {
3      public string Usuario { get; set; }
4      public string Senha { get; set; }
5
6      public override double CalculaBonificacao()
7      {
8          return this.Salario * 0.6 + 100;
9      }
10 }
```

Tabela 29: Gerente.cs

1.10 Exercícios Complementares

1) Defina na classe Funcionario um método para imprimir na tela o nome, salário e bonificação dos funcionários.

2) Reescreva o método que imprime os dados dos funcionários nas classes Gerente, Telefonista e Secretaria para acrescentar a impressão dos dados específicos de cada tipo de funcionário.

3) Modifique a classe TestaFuncionarios para utilizar o método MostraDados().

Bons Estudos!