UNIDADE 7 – POLIMORFISMO

Nessa unidade veremos polimorfismo na programação orientada a objetos. O termo polimorfismo é originário do grego e significa "muitas formas" (poli = muitas, morphos = formas). Permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam.

Ele é caracterizado quando duas ou mais classes distintas tem métodos de mesmo nome, de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas, sem necessidade de tratar de forma diferenciada conforme a classe do objeto.

SUMÁRIO

1 POLIMORFISMO

1.1 CONTROLE DE PONTO

O sistema do banco deve possuir um controle de ponto para registrar a entrada e saída dos funcionários. O pagamento dos funcionários depende dessas informações. Podemos definir uma classe para implementar o funcionamento de um relógio de ponto.

```
1  using System;
2
3  namespace Unidades
4  {
5      public class ControleDePonto
6      {
7          public void RegistraEntrada(Gerente g)
8          {
9              DateTime agora = DateTime.Now;
10             string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
11
12             Console.WriteLine(" ENTRADA : " + g.Codigo);
13             Console.WriteLine(" DATA : " + horario);
14         }
15
16         public void RegistraSaida(Gerente g)
17         {
18             DateTime agora = DateTime.Now;
19             string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
20
21             Console.WriteLine(" SAÍDA : " + g.Codigo);
22
23         }
24     }
25 }
```

Tabela 1: ControleDePonto.cs

A classe acima possui dois métodos: o primeiro para registrar a entrada e o segundo para registrar a saída dos gerentes do banco. Contudo, esses dois métodos não são aplicáveis aos outros tipos de funcionários.

Seguindo essa abordagem, a classe `ControleDePonto` precisaria de um par de métodos para cada cargo. Então, a quantidade de métodos dessa classe seria igual a quantidade de cargos multiplicada por dois. Imagine que no banco exista 30 cargos distintos. Teríamos 60 métodos na classe `ControleDePonto`.

Os procedimentos de registro de entrada e saída são idênticos para todos os funcionários. Consequentemente, qualquer alteração na lógica desses procedimentos implicaria na modificação de todos os métodos da classe `ControleDePonto`.

Além disso, se o banco definir um novo tipo de funcionário, dois novos métodos praticamente idênticos aos que já existem teriam de ser adicionados na classe `ControleDePonto`. Analogamente, se um cargo deixar de existir, os dois métodos correspondentes da classe `ControleDePonto` deverão ser retirados.

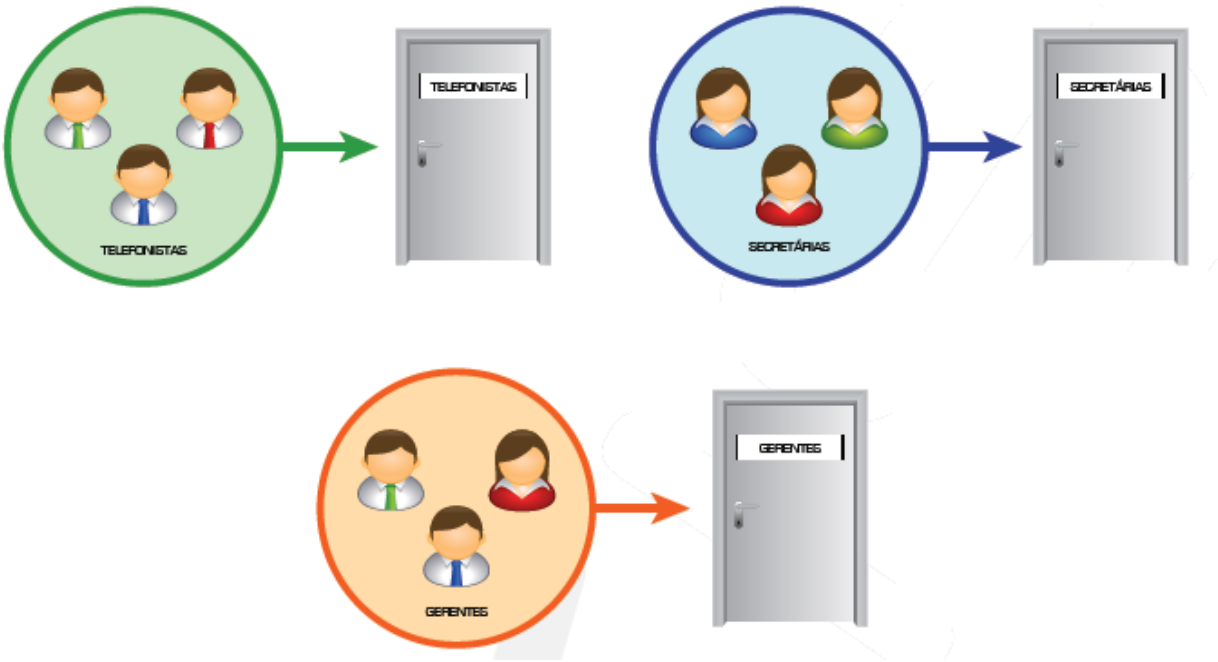

Figura 1: Métodos específicos

1.2 MODELAGEM DOS FUNCIONÁRIOS

Com o intuito inicial de reutilizar código, podemos modelar os diversos tipos de funcionários do banco utilizando o conceito de herança.

```
1  public class Funcionario
2  {
3      public int Codigo { get; set; }
4  }
```

Tabela 2: Funcionario.cs

```
1  public class Gerente : Funcionario
2  {
3      public string Usuario { get; set; }
4      public string Senha { get; set; }
5  }
```

Tabela 3: Gerente.cs

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

Tabela 4: Telefonista.cs

1.3 É UM

Além de gerar reaproveitamento de código, a utilização de herança permite que objetos criados a partir das classes específicas sejam tratados como objetos da classe genérica.

Em outras palavras, a herança entre as classes que modelam os funcionários permite que objetos criados a partir das classes Gerente ou Telefonista sejam tratados como objetos da classe `Funcionario`.

No código da classe Gerente utilizamos o símbolo `:` para indicar que a classe Gerente é uma subclasse de `Funcionario`. Esse símbolo pode ser interpretado como a expressão: É UM ou É UMA.

```
1  class Gerente : Funcionario
2  //Gerente É UM funcionário
```

Tabela 5: Gerente.cs

Como está explícito no código que todo gerente é um funcionário então podemos criar um objeto da classe Gerente e tratá-lo como um objeto da classe `Funcionario` também.

```
1  // Criando um objeto da classe Gerente
2  Gerente g = new Gerente();
3
4  // Tratando um gerente como um objeto da classe Funcionario
5  Funcionario f = g;
```

Tabela 6: Generalizando.cs

Em alguns lugares do sistema do banco será mais vantajoso tratar um objeto da classe Gerente como um objeto da classe Funcionario.

1.4 MELHORANDO O CONTROLE DE PONTO

O registro da entrada ou saída não depende do cargo do funcionário. Não faz sentido criar um método que registre a entrada para cada tipo de funcionário, pois eles serão sempre idênticos. Analogamente, não faz sentido criar um método que registre a saída para cada tipo de funcionário.

Dado que podemos tratar os objetos das classes derivadas de `Funcionario` como sendo objetos dessa classe, podemos implementar um método que seja capaz de registrar a entrada de qualquer funcionário independentemente do cargo. Analogamente, podemos fazer o mesmo para o procedimento de saída.

```
1  using System;
2
3  namespace Unidades
4  {
5      public class ControleDePonto
6      {
7          public void RegistraEntrada(Funcionario f)
8          {
9              DateTime agora = DateTime.Now;
10             string horario = String.Format("{0: d/M/ yyyy HH:mm:ss}", agora);
11
12             Console.WriteLine("ENTRADA: " + f.Codigo);
13             Console.WriteLine("DATA: " + horario);
14
15         }
16
17         public void RegistraSaida(Funcionario f)
18         {
19             DateTime agora = DateTime.Now;
20             string horario = String.Format("{0: d/M/ yyyy HH:mm:ss}", agora);
21
22             Console.WriteLine("SAÍDA : " + f.Codigo);
23             Console.WriteLine("DATA : " + horario);
24         }
25     }
26 }
```

Tabela 7: ControleDePonto.cs

Os métodos `RegistraEntrada()` e RegistraSaid`a()` recebem referências de objetos da classe `Funcionario` como parâmetro. Consequentemente, podem receber referências de objetos de qualquer classe que deriva direta ou indiretamente da classe `Funcionario`.

A capacidade de tratar objetos criados a partir das classes específicas como objetos de uma classe genérica é chamada de polimorfismo.

Aplicando a ideia do polimorfismo no controle de ponto, facilitamos a manutenção da classe `ControleDePonto`. Qualquer alteração no procedimento de entrada ou saída implica em alterações em métodos únicos.

Além disso, novos tipos de funcionários podem ser definidos sem a necessidade de qualquer alteração na classe `ControleDePonto`. Analogamente, se algum cargo deixar de existir, nada precisará ser modificado na classe `ControleDePonto`.

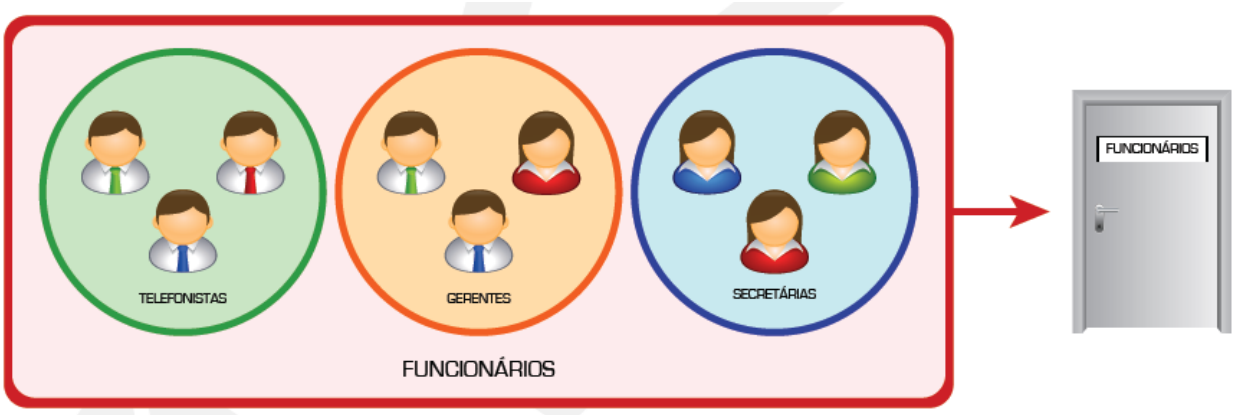

Figura 2: Método genérico

1.5 Exercícios de Fixação

1) Crie um projeto no chamado Unidade 7.

2) Defina uma classe genérica para modelar as contas do banco.

```
1  class Conta
2  {
3      public double Saldo { set; get; }
4  }
```

Tabela 8: Conta.cs

3) Defina duas classes específicas para dois tipos de contas do banco: poupança e corrente.

```
1  class ContaPoupanca : Conta
2  {
3      public int DiaDoAniversario { get; set; }
4  }
```

Tabela 9: ContaPoupanca.cs

```
1  class ContaCorrente : Conta
2  {
3      public double Limite { get; set; }
4  }
```

Tabela 10: ContaCorrente.cs

4) Defina uma classe para especificar um gerador de extratos.

```
1  public class GeradorDeExtrato
2  {
3      public void ImprimeExtratoBasico(Conta c)
4      {
5          DateTime agora = DateTime.Now;
6          string horario = String.Format(" {0: d/M/ yyyy HH:mm:ss}", agora);
7
8          Console.WriteLine("DATA : " + horario);
9          Console.WriteLine("SALDO : " + c.Saldo);
10     }
11 }
```

Tabela 10: GeradorDeExtrato.cs

5) Faça um teste para o gerador de extratos.

```
1  class TestaGeradorDeExtrato
2  {
3      static void Main()
4      {
5          GeradorDeExtrato gerador = new GeradorDeExtrato();
6
7          ContaPoupanca cp = new ContaPoupanca();
8          cp.Saldo = 2000;
9
10         ContaCorrente cc = new ContaCorrente();
11         cc.Saldo = 1000;
12
13         gerador.ImprimeExtratoBasico(cp);
14         gerador.ImprimeExtratoBasico(cc);
15     }
16 }
```

Tabela 11: TestaGeradorDeExtrato.cs

1.6 EXERCÍCIOS COMPLEMENTARES

1) Defina uma classe para modelar de forma genérica os funcionários do banco.

2) Implemente duas classes específicas para modelar dois tipos particulares de funcionários do banco: os gerentes e as telefonistas.

3) Implemente o controle de ponto dos funcionários. Crie uma classe com dois métodos: o primeiro para registrar a entrada dos funcionários e o segundo para registrar a saída.

4) Teste a lógica do controle de ponto, registrando a entrada e a saída de um gerente e de uma telefonista.

Bons Estudos!