UNIDADE 9 – INTERFACES

Nessa unidade veremos a funcionalidade da Interface na Orientação objetos.

Uma interface contém apenas as assinaturas de métodos, Propriedades, eventos ou indexadores. Uma classe ou estrutura que implementa a interface deve implementar os membros da interface que estão especificados na definição da interface.

SUMÁRIO

1 INTERFACES

1.1 PADRONIZAÇÃO

No dia a dia, estamos acostumados a utilizar aparelhos que dependem de energia elétrica. Esses aparelhos possuem um plugue que deve ser conectado a uma tomada para obter a energia necessária.

Diversas empresas fabricam aparelhos elétricos com plugues. Analogamente, diversas empresas fabricam tomadas elétricas. Suponha que cada empresa decida por conta própria o formato dos plugues ou das tomadas que fabricará. Teríamos uma infinidade de tipos de plugues e tomadas que tornaria a utilização dos aparelhos elétricos uma experiência extremamente desagradável.

Inclusive, essa falta de padrão pode gerar problemas de segurança aos usuários. Os formatos dos plugues ou das tomadas pode aumentar o risco de uma pessoa tomar umchoque elétrico.

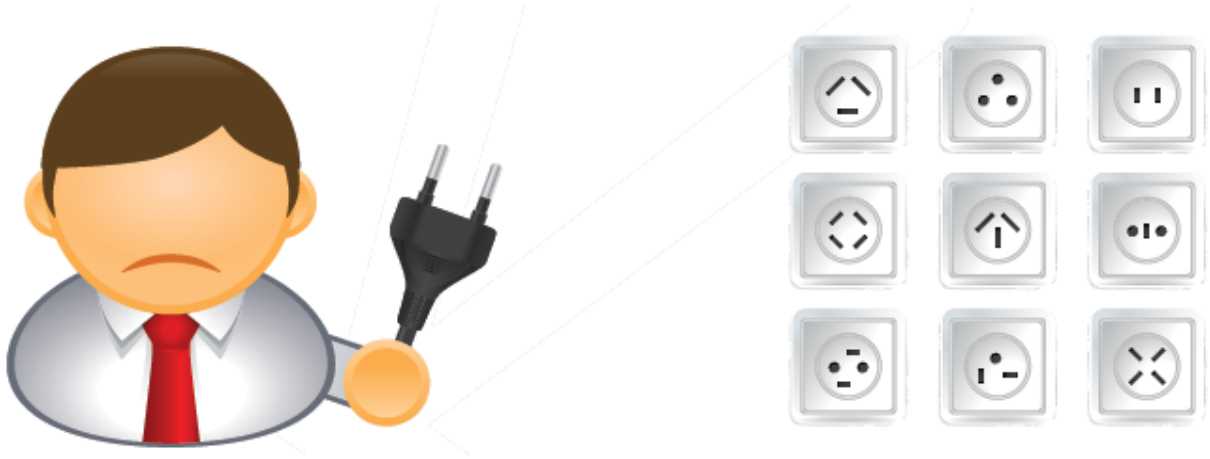

Figura 1: Tomadas despadronizadas

Com o intuito de facilitar a utilização dos consumidores e aumentar a segurança dos mesmos, o governo através dos órgãos responsáveis estabelece padrões para os plugues e tomadas. Esses padrões estabelecem restrições que devem ser respeitadas pelos fabricantes dos aparelhos e das tomadas.

Em diversos contextos, padronizar pode trazer grandes benefícios. Inclusive, no desenvolvimento de aplicações. Mostraremos como a ideia de padronização pode ser contextualizada nos conceitos de orientação a objetos.

1.2 CONTRATOS

Num sistema orientado a objetos, os objetos interagem entre si através de chamadas de métodos (troca de mensagens). Podemos dizer que os objetos se "encaixam" através dos métodos públicos assim como um plugue se encaixa em uma tomada através dos pinos.

Para os objetos de uma aplicação "conversarem" entre si mais facilmente é importante padronizar o conjunto de métodos oferecidos por eles. Assim como os plugues encaixam nas tomadas mais facilmente graças aos padrões definidos pelo governo.

Um padrão é definido através de especificações ou contratos. Nas aplicações orientadas a objetos, podemos criar um "contrato" para definir um determinado conjunto de métodos que deve ser implementado pelas classes que "assinarem" este contrato. Em orientação a objetos, um contrato é chamado de **interface**. Uma interface é composta basicamente por métodos abstratos.

1.3 EXEMPLO

No sistema do banco, podemos definir uma interface (contrato) para padronizar as assinaturas dos métodos oferecidos pelos objetos que representam as contas do banco.

```
1  interface IConta
2  {
3      void Deposita(double valor);
4      void Saca(double valor);
5  }
```

Tabela 1: IConta.cs

Observe que somente assinaturas de métodos são declaradas no corpo de uma interface. Todos os métodos de uma interface são públicos e não pode incluir modificadores de acesso. Uma interface só pode conter métodos, propriedades, indexadores e eventos. Por convenção, em C#, o nome de uma interface deve ter o prefixo **I**.

As classes que definem os diversos tipos de contas que existem no banco devem implementar (assinar) a interface IConta. Para isso, devemos aplicar o comando :.

```
1  class ContaPoupanca : IConta
2  {
3      public void Deposita(double valor)
4      {
5          // implementacao
6      }
7
8      public void Saca(double valor)
9      {
10         // implementacao
11     }
12 }
```

Tabela 2: ContaPoupanca.cs

```
1  class ContaCorrente : IConta
2  {
3      public void Deposita(double valor)
4      {
5          // implementacao
6      }
7
8      public void Saca(double valor)
9      {
10         // implementacao
11     }
12 }
```

Tabela 3: ContaCorrente.cs

As classes concretas que implementam uma interface são obrigadas a possuir uma implementação para cada método declarado na interface. Caso contrário, ocorrerá um erro de compilação.

```
1  // Esta classe NÃO compila porque ela não implementou o método Saca ()
2  class ContaCorrente : IConta
3  {
4      public void Deposita(double valor)
5      {
6          // implementacao
7      }
8  }
```

Tabela 4: ContaCorrente.cs

A primeira vantagem de utilizar uma interface é a padronização das assinaturas dos métodos oferecidos por um determinado conjunto de classes. A segunda vantagem é garantir que determinadas classes implementem certos métodos.

1.4 POLIMORFISMO

Se uma classe implementa uma interface, podemos aplicar a ideia do polimorfismo assim como quando aplicamos herança. Dessa forma, outra vantagem da utilização de interfaces é o ganho do polimorfismo.

Como exemplo, suponha que a classe ContaCorrente implemente a interface IConta. Podemos guardar a referência de um objeto do tipo ContaCorrente em uma variável do tipo IConta.

```
1  IConta c = new ContaCorrente();
```

Tabela 5: ContaCorrente.cs

Além disso podemos passar uma variável do tipo ContaCorrente para um método que o parâmetro seja do tipo IConta.

```
1  class GeradorDeExtrato
2  {
3      public void GeraExtrato(IConta c)
4      {
5          // implementação
6      }
7  }
```

Tabela 6: GeradorDeExtrato.cs

```
1  GeradorDeExtrato g = new GeradorDeExtrato();
2  ContaCorrente c = new ContaCorrente();
3  g.GeraExtrato(c);
```

Tabela 7: Aproveitando o polimorfismo

O método GeraExtrato() pode ser aproveitado para objetos criados a partir de classes que implementam direta ou indiretamente a interface IConta.

1.5 INTERFACE E HERANÇA

As vantagens e desvantagens entre interface e herança, provavelmente, é um dos temas mais discutido nos blogs, fóruns e revistas que abordam desenvolvimento de software orientado a objetos.

Muitas vezes, os debates sobre este assunto se estendem mais do que a própria importância desse tópico. Muitas pessoas se posicionam de forma radical defendendo a utilização de interfaces ao invés de herança em qualquer situação.

Normalmente, esses debates são direcionados na análise do que é melhor para manutenção das aplicações: utilizar interfaces ou aplicar herança.

A grosso modo, priorizar a utilização de interfaces permite que alterações pontuais em determinados trechos do código fonte sejam feitas mais facilmente pois diminui as ocorrências de efeitos colaterais indesejados no resto da aplicação. Por outro lado, priorizar a utilização de herança pode diminuir a quantidade de código escrito no início do desenvolvimento de um projeto.

Algumas pessoas propõem a utilização de interfaces juntamente com composição para substituir totalmente o uso de herança. De fato, esta é uma alternativa interessante pois possibilita que um trecho do código fonte de uma aplicação possa ser alterado sem causar efeito colateral no restante do sistema além de permitir a reutilização de código mais facilmente.

Em C#, como não há herança múltipla, muitas vezes, interfaces são apresentadas como uma alternativa para obter um grau maior de polimorfismo.

Por exemplo, suponha duas árvores de herança independentes.

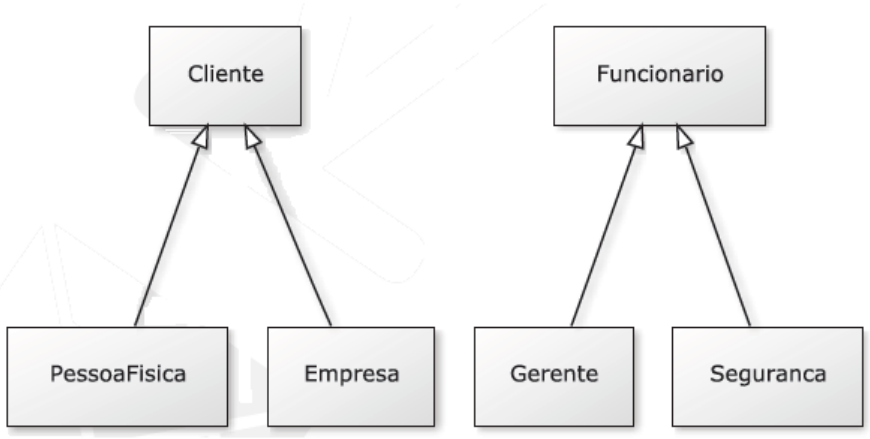

Figura 2: Duas árvores de herança independentes

Suponha que os gerentes e as empresas possam acessar o sistema do banco com um nome de usuário e uma senha. Seria interessante utilizar um único método para implementar a autenticação desses dois tipos de objetos. Mas, qual seria o tipo de parâmetro deste método? Lembrando que ele deve aceitar gerentes e empresas.

```
1  class AutenticadorDeUsuario
2  {
3      public bool Autentica()
4      {
5          // implementação
6      }
7  }
```

Tabela 8: AutenticadorDeUsuario.cs

De acordo comas árvores de herança, não há polimorfismo entre objetos da classe Gerente e da classe Empresa. Para obter polimorfismo entre os objetos dessas duas classes somente com herança, deveríamos colocá-las na mesma árvore de herança. Mas, isso não faz sentido pois uma empresa não é um funcionário e o gerente não é cliente. Neste caso, a solução é utilizar interfaces para obter o polimorfismo entre desejado.

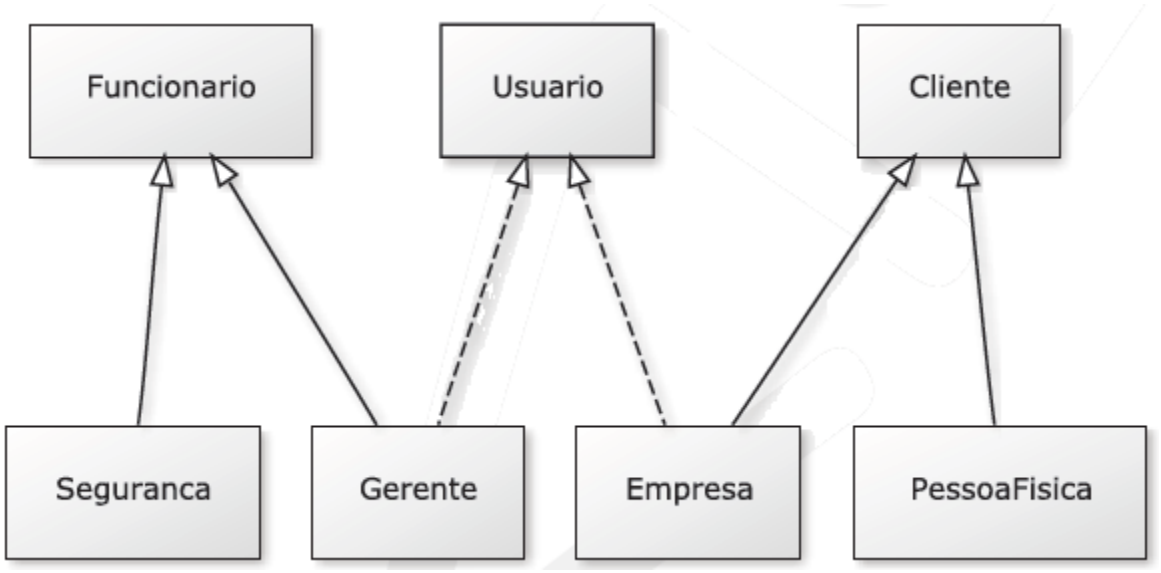

Figura 3: Obtendo mais polimorfismo

Agora, conseguimos definir o que o método Autentica() deve receber como parâmetro para trabalhar tanto com gerentes quanto com empresas. Ele deve receber um parâmetro do tipo IUsuario.

```
1  class AutenticadorDeUsuario
2  {
3      public bool Autentica(IUsuario u)
4      {
5          // implementação
6      }
7  }
```

Tabela 9: AutenticadorDeUsuario.cs

1.6 Exercícios de Fixação

1) Crie um projeto no chamado **Unidade9** e uma pasta **ExercicioFixacao**.

2) Defina uma interface para padronizar as assinaturas dos métodos das contas do banco.

```
1  // prefixo I para seguir a convenção
2  internal interface IConta
3  {
4      void Deposita(double valor);
5      void Saca(double valor);
6      double Saldo { get; set; }
7  }
```

Tabela 10: IConta.cs

3) Agora, crie algumas classes para modelar tipos diferentes de conta.

```
class ContaCorrente : IConta
{
    public double Saldo { get; set; }
    private double taxaPorOperacao = 0.45;

    public void Deposita(double valor)
    {
        this.Saldo += valor - this.taxaPorOperacao;
    }

    public void Saca(double valor)
    {
        this.Saldo -= valor + this.taxaPorOperacao;
    }
}
```

Tabela 11: ContaCorrente.cs

```
class ContaPoupanca : IConta
{
    public double Saldo { get; set; }

    public void Deposita(double valor)
    {
        this.Saldo += valor;
    }

    public void Saca(double valor)
    {
        this.Saldo -= valor;
    }
}
```

Tabela 12: ContaPoupanca.cs

4) Faça um teste simples com as classes criadas anteriormente.

```
class TestaContas
{
    private static void Main()
    {
        ContaCorrente c1 = new ContaCorrente();
        ContaPoupanca c2 = new ContaPoupanca();

        c1.Deposita(500);
```

```
8           c2.Deposita(500);
9
10          c1.Saca(100);
11          c2.Saca(100);
12
13          Console.WriteLine(c1.Saldo);
14          Console.WriteLine(c2.Saldo);
15      }
    }
```

Tabela 13: TestaContas.cs

5) Altere a assinatura do método Deposita() na classe ContaCorrente. Você pode acrescentar um "r" no nome do método. O que acontece? Obs: desfaça a alteração depois deste exercício.

6) Crie umgerador de extratos comummétodo que pode trabalhar com todos os tipos de conta.

```
1   class GeradorDeExtrato
2   {
3       public void GeraExtrato(IConta c)
4       {
5           Console.WriteLine(" EXTRATO ");
6           Console.WriteLine(" SALDO : " + c.Saldo);
7       }
8   }
```

Tabela 14: GeradorDeExtrato.cs

7) Teste o gerador de extrato.

```
1   class TestaGeradorDeExtrato
2   {
3       private static void Main()
4       {
5           ContaCorrente c1 = new ContaCorrente();
6           ContaPoupanca c2 = new ContaPoupanca();
7
8           c1.Deposita(500);
9           c2.Deposita(500);
10
11          GeradorDeExtrato g = new GeradorDeExtrato();
12          g.GeraExtrato(c1);
13          g.GeraExtrato(c2);
14      }
15  }
```

Tabela 14: GeradorDeExtrato.cs

1.7 EXERCÍCIOS COMPLEMENTARES

1) Na mesma Unidade crie uma pasta ExerciciosComplementares.

2) Cria uma interface com o nome Prova com as seguintes declarações:

```
// Por favor, jamais misture inglês com português, ou é um ou é outro!
//Ex: GetPontuacao(), SetDataEvento() ← Evite

int BuscaPontuacao();
void MudaDataEvento(int ano, int mes, int dia);
int BuscaDiaEvento();
int BuscaMesEvento();
int BuscaAnoEvento();
int BuscaNumElementos();
void MudaPontuacao(int p);
```

a) Cria a Classe Futebol que implemente a interface Prova com os seguintes métodos e atributos.

```
Futebol(string nom, int n) // Construtor
int dia,mes,ano, elementosEquipe;// data do jogo, nº elementos da equipa
char resultado; // resultado "V-Vitória, E-Empate, D-Derrota
string nome; / /Nome da equipe
Int Pontuacao ; //Pontuação da Equipe
```

b) Implemente na classe Program, ou Principal, as seguintes funcionalidades:

- Criação de um array de provas de futebol
- Preenchimento da informação relativa das provas.
- Listagem do nº da equipe e da pontuação.
- Não precisa de Menu nem de Cadastro, apenas mostre os dados preenchidos. Esses dados podem ser aleatoriamente. (O número da equipe, corresponde ao índice do array mais 1).

Bons Estudos!