UNIDADE 5 – ENCAPSULAMENTO

Nessa unidade veremos uma das ideias mais importantes da orientação a objetos é o encapsulamento. Encapsular significa esconder a implementação dos objetos. O encapsulamento favorece principalmente dois aspectos de um sistema: a manutenção e o desenvolvimento.

Uma das principais características das linguagens orientadas a objetos é prover mecanismos para a implementação de códigos seguros, ou seja, é garantir que as variáveis (campos) de uma classe recebam exatamente os valores que se espera que elas recebam.

SUMÁRIO

1 ENCAPSULAMENTO

1.1 ATRIBUTOS PRIVADOS

No sistema do banco, cada objeto da classe F**uncionário** possui um atributo para guardar o salário do funcionário que ele representa.

```
1  class Funcionario
2  {
3      public double salario;
4  }
```

Tabela 1: Funcionario.cs

O atributo salario pode ser acessado ou modificado por código escrito por qualquer classe. Portanto, o controle do atributo salario é descentralizado.

Para identificar algum erro relacionado a manipulação dos salários dos funcionários, é necessário verificar o código de todos os arquivos onde a classe Funcionário está definida. Quanto maior o número de arquivos, menos eficiente será a manutenção da aplicação.

Podemos obter um controle centralizado tornando o atributo salario **privado** e definindo métodos para implementar todas as lógicas que utilizam ou modificam o valor desse atributo. Em C#, se nenhum modificador de visibilidade for definido para um determinado atributo, esse atributo será considerado privado por padrão. Contudo, é uma boa prática deixar explícito no código que o atributo é privado, adicionando o modificador **private**.

```
1  class Funcionario
2  {
3      private double salario;
4
5      0 references
6      public void AumentaSalario(double aumento)
7      {
8          // lógica para aumentar o salário
9      }
   }
```

Tabela 2: Funcionario.cs

Um atributo privado só pode ser acessado ou alterado por código escrito dentro da classe na qual ele foi definido. Se algum código fora da classe Funcionário tentar acessar ou alterar o valor do atributo privado salario, um erro de compilação será gerado.

Definir todos os atributos como privado e métodos para implementar as lógicas de acesso e alteração é quase uma regra da orientação a objetos. O intuito é ter sempre um controle centralizado dos dados dos objetos para facilitar a manutenção do sistema e a detecção de erros.

1.2 MÉTODOS PRIVADOS

O papel de alguns métodos pode ser o de auxiliar outros métodos da mesma classe. E muitas vezes, não é correto chamar esses métodos auxiliares de fora da sua classe diretamente.

No exemplo abaixo, o método `DescontaTarifa()` é um método auxiliar dos métodos Deposita() e `Saca()`. Além disso, ele não deve ser chamado diretamente, pois a tarifa só deve ser descontada quando ocorre um depósito ou um saque.

```
1  class Conta
2  {
3      private double saldo;
4
5      0 references
6      public void Deposita(double valor)
7      {
8          this.saldo += valor;
9          this.DescontaTarifa();
10     }
11
12     0 references
13     public void Saca(double valor)
14     {
15         this.saldo -= valor;
16         this.DescontaTarifa();
17     }
18
19     2 references
20     void DescontaTarifa()
21     {
22         this.saldo -= 0.1;
23     }
24 }
```

Tabela 3: Conta.cs

Para garantir que métodos auxiliares não sejam chamados por código escrito fora da classe na qual eles foram definidos, podemos torná-los privados, acrescentando o modificador `private`.

```
1  private void DescontaTarifa()
2  {
3      this.saldo -= 0.1;
4  }
```

Tabela 4: Método privado DescontaTarifa()

Qualquer chamada ao método `DescontaTarifa()` realizada fora da classe Conta gera um erro de compilação.

1.3 MÉTODOS PÚBLICOS

Os métodos que devem ser chamados a partir de qualquer parte do sistema devem possuir o modificador de visibilidade public.

```
1  class Conta
2  {
3      private double saldo;
4
5      0 references
6      public void Deposita(double valor)
7      {
8          this.saldo += valor;
9          this.DescontaTarifa();
10     }
11
12     0 references
13     public void Saca(double valor)
14     {
15         this.saldo -= valor;
16         this.DescontaTarifa();
17     }
18
19     0 references
20     private void DescontaTarifa()
21     {
22         this.saldo -= 0.1;
23     }
24 }
```

Tabela 5: Conta.cs

1.4 Implementação e Interface de Uso

Dentro de um sistema orientado a objetos, cada objeto realiza um conjunto de tarefas de acordo com as suas responsabilidades. Por exemplo, os objetos da classe Conta realizam as operações de saque, depósito, transferência e geração de extrato.

Para descobrir o que um objeto pode fazer, basta olhar para as assinaturas dos métodos públicos definidos na classe desse objeto. A assinatura de um método é composta pelo seu nome e seus parâmetros. As assinaturas dos métodos públicos de um objeto formam a sua interface de uso.

Por outro lado, para descobrir como um objeto da classe Conta realiza as suas operações, devemos observar o corpo de cada um dos métodos dessa classe. Os corpos dos métodos constituem a implementação das operações dos objetos.

1.5 Por quê encapsular?

Uma das ideias mais importantes da orientação a objetos é o encapsulamento. Encapsular significa esconder a implementação dos objetos. O encapsulamento favorece principalmente dois aspectos de um sistema: a manutenção e o desenvolvimento.

A manutenção é favorecida pois, uma vez aplicado o encapsulamento, quando o funcionamento de um objeto deve ser alterado, em geral, basta modificar a classe do mesmo.

O desenvolvimento é favorecido pois, uma vez aplicado o encapsulamento, conseguimos determinar precisamente as responsabilidades de cada classe da aplicação.

O conceito de encapsulamento pode ser identificado em diversos exemplos do cotidiano. Mostraremos alguns desses exemplos para esclarecer melhor a ideia.

1.6 Celular - Escondendo a complexidade

Hoje em dia, as pessoas estão acostumadas com os celulares. Os botões, a tela e os menus de um celular formam a interface de uso do mesmo. Em outras palavras, o usuário interage com esses aparelhos através dos botões, da tela e dos menus. Os dispositivos internos de um celular e os processos que transformam o som capturado pelo microfone em ondas que podem ser transmitidas para uma antena da operadora de telefonia móvel constituem a implementação do celular.

Do ponto de vista do usuário de um celular, para fazer uma ligação, basta digitar o número do telefone desejado e clicar no botão que efetua a ligação. Porém, diversos processos complexos são realizados pelo aparelho para que as pessoas possam conversar através dele. Se os usuários tivessem que

possuir conhecimento de todo o funcionamento interno dos celulares, certamente a maioria das pessoas não os utilizariam.

No contexto da orientação a objetos, aplicamos o encapsulamento para criar objetos mais simples de serem utilizados em qualquer parte do sistema.

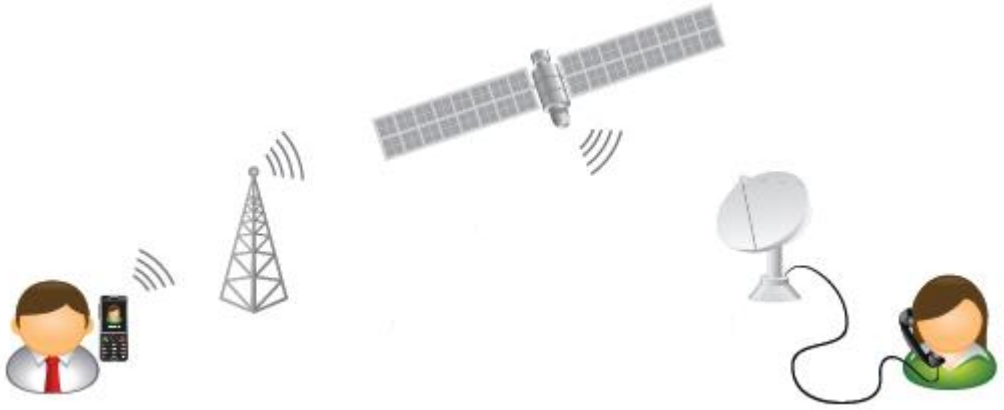

Figura 1: Celular

1.7 CARRO - EVITANDO EFEITOS COLATERAIS

A interface de uso de um carro é composta pelos dispositivos que permitem que o motorista conduza o veículo (volante, pedais, alavanca do câmbio, etc).

A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos.

Nos carros mais antigos, o dispositivo interno que leva o combustível para o motor é o carburador. Nos carros mais novos, o carburador foi substituído pelo sistema de injeção eletrônica. Inclusive, algumas oficinas especializadas substituem o carburador pelo sistema de injeção eletrônica. Essa alteração na implementação do carro não afeta a maneira que o motorista dirige. Todo mundo que sabe dirigir um carro com carburador também sabe dirigir um carro com injeção eletrônica.

Hoje em dia, as montadoras fabricam veículos com câmbio mecânico ou automático. O motorista acostumado a dirigir carros com câmbio mecânico pode ter dificuldade para dirigir carros com câmbio automático e vice-versa. Quando a interface de uso do carro é alterada, a maneira de dirigir é afetada, fazendo com que as pessoas que sabem dirigir tenham que se adaptar

No contexto da orientação a objetos, aplicando o conceito do encapsulamento, as implementações dos objetos ficam “escondidas”. Dessa forma, podemos modificá-las sem afetar a maneira de utilizar esses

objetos. Por outro lado, se alterarmos a interface de uso que está exposta, afetaremos a maneira de usar os objetos.

Considere, por exemplo, a mudança do nome de um método público. Todas as chamadas a esse método devem ser alteradas, o que pode causar diversos efeitos colaterais nas classes da aplicação.

Figura 2: Substituição de um volante por um joystick

1.8 Máquinas de Porcarias - Aumentando o controle

Estamos acostumados a utilizar máquinas de refrigerantes, de salgadinhos, de doces, de café, etc. Em geral, essas máquinas oferecem uma interface de uso composta por:

- Entradas para moedas ou cédulas.
- Botões para escolher o produto desejado.
- Saída do produto.
- Saída para o troco.

Normalmente, essas máquinas são extremamente protegidas. Elas garantem que nenhum usuário mal intencionado (ou não) tente alterar a implementação da máquina, ou seja, tente alterar como a máquina funciona por dentro. Levando essa ideia para um sistema orientado a objetos, um objeto deve ser bem protegido para que outros objetos não prejudiquem o seu funcionamento interno.

Figura 3: Máquina de Porcarias

1.9 ACESSANDO OU MODIFICANDO ATRIBUTOS

Aplicando a ideia do encapsulamento, os atributos deveriam ser todos privados. Consequentemente, os atributos não podem ser acessados ou modificados por código escrito fora da classe na qual eles foram definidos.

Porém, muitas vezes, as informações armazenadas nos atributos precisam ser consultadas de qualquer lugar do sistema. Nesse caso, podemos disponibilizar métodos para consultar os valores dos atributos.

```
1  class Cliente
2  {
3      private string nome;
4      -references
5      public string ConsultaNome()
6      {
7          return this.nome;
8      }
   }
```

Tabela 6: Cliente.cs

Da mesma forma, eventualmente, é necessário modificar o valor de um atributo a partir de qualquer lugar do sistema. Nesse caso, também poderíamos criar um método para essa tarefa.

```
1  class Cliente
2  {
3      private string nome;
4      -references
5      public void AlteraNome(string nome)
6      {
7          this.nome = nome;
8      }
   }
```

Tabela 7: Clinte.cs

Muitas vezes, é necessário consultar e alterar o valor de um atributo a partir de qualquer lugar do sistema. Nessa situação, podemos definir os dois métodos discutidos anteriormente. Mas, o que é melhor? Criar os dois métodos (um de leitura e outro de escrita) ou deixar o atributo público?

Quando queremos consultar a quantidade de combustível de um automóvel, olhamos o painel ou abrimos o tanque de combustível?

Quando queremos alterar o toque da campainha de um celular, utilizamos os menus do celular ou desmontamos o aparelho?

Acessar ou modificar as propriedades de um objeto manipulando diretamente os seus atributos é uma abordagem que normalmente gera problemas. Por isso, é mais seguro para a integridade dos objetos e, consequentemente, para a integridade da aplicação, que esse acesso ou essa modificação sejas realizados através de métodos do objeto. Utilizando métodos, podemos controlar como as alterações e as consultas são realizadas. Ou seja, temos um controle maior.

1.10 PROPRIEDADES

A linguagem C# disponibiliza uma outra maneira para acessar os atributos: as propriedades. Uma propriedade, basicamente, agrupa os métodos de consulta e alteração dos atributos.

```
1  class Cliente
2  {
3      private string nome;
4
5      -references
6      public string Nome
7      {
8          get
9          {
10             return this.nome;
11         }
12         set
13         {
14             this.nome = value;
15         }
16     }
17 }
```

Tabela 8: Clinte.cs

A sintaxe de utilização das propriedades é semelhante a de utilização dos atributos públicos.

```
1  Cliente c = new Cliente();
2  c.Nome = "Thiago Sartor";
```

Tabela 9: Cliente.cs

1.11 PROPRIEDADES AUTOMÁTICAS

Muitas vezes, a lógica das propriedades é trivial. Ou seja, queremos apenas realizar uma atribuição ou devolver um valor;

```
1  class Cliente
2  {
3      private string nome;
4
5      -references
6      public string Nome
7      {
8          get
9          {
10             return this.nome;
11         }
12         set
13         {
14             this.nome = value;
15         }
16     }
17 }
```

Tabela 10: Cliente.cs

Nesses casos, podemos aplicar o recurso de propriedades automáticas. O código fica mais simples e prático.

```
1  class Cliente
2  {
3      -references
4      public string Nome { get; set; }
5  }
```

Tabela 11: Cliente.cs

1.12 Exercícios de Fixação

1) Crie um novo projeto Unidade 5.

2) Defina uma classe para representar os funcionários do banco com um atributo para guardar os salários e outro para os nomes.

```
1  class Funcionario
2  {
3      public double salario;
4      public string nome;
5  }
```

Tabela 12: Funcionario.cs

3) Teste a classe Funcionário criando um objeto e manipulando diretamente os seus atributos.

```
1  class Teste
2  {
3      -references
4      private static void Main()
5      {
6          Funcionario f = new Funcionario();
7
8          f.nome = "Thiago Sartor";
9          f.salario = 2000;
10
11         Console.WriteLine(f.nome);
12         Console.WriteLine(f.salario);
13     }
14 }
```

Tabela 13: Teste.cs

4) Compile a classe Teste e perceba que ela pode acessar ou modificar os valores dos atributos de um objeto da classe Funcionário. Execute o teste e observe o console.

5) Aplique a ideia do encapsulamento tornando os atributos definidos na classe Funcionário privados.

```
1  class Funcionario
2  {
3      public double salario;
4      public string nome;
5  }
```

Tabela 14: Funcionario.cs

6) Tente compilar novamente a classe Teste. Observe os erros de compilação. Lembre-se que um atributo privado só pode ser acessado por código escrito na própria classe do atributo.

7) Crie propriedades com nomes padronizados para os atributos definidos na classe Funcionário.

```
1  class Funcionario
2  {
3      private double salario;
4      private string nome;
5
6      -references
7      public double Salario
8      {
9          get
10         {
11             return this.salario;
12
13
```

```
        }
        set
        {
            this.salario = value;
        }
    }

    0 references
    public string Nome
    {
        get
        {
            return this.nome;
        }
        set
        {
            this.nome = value;
        }

    }
}
```

Tabela 15: Funcionario.cs

8) Altere a classe Teste para que ela utilize as propriedades ao invés de manipular os atributos do objeto da classe Funcionário diretamente.

```
class Teste
{
    0 references
    private static void Main()
    {
        Funcionario f = new Funcionario();

        f.Nome = "Thiago Sartor";
        f.Salario = 2000;

        Console.WriteLine(f.Nome);
        Console.WriteLine(f.Salario);
    }
}
```

Tabela 16: Teste.cs

9) Altere a classe Funcionário substituindo a propriedade e o atributo por uma propriedade automática.

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

Tabela 17: Funcionario.cs

1.13 EXERCÍCIOS COMPLEMENTARES

1) Implemente uma classe para modelar de forma genérica as contas do banco.

2) Crie objetos da classe que modela as contas do banco e utilize as propriedades para alterar os valores dos atributos.

Bons Estudos!