



# INVERSÃO DE CONTROLE E INJEÇÃO DE DEPENDÊNCIAS

COMO CRIAR SISTEMAS FLEXÍVEIS CUJA MANUTENÇÃO E  
EXPANSÃO NÃO GEREM O FAMOSO “EFEITO DOMINÓ”

# O QUE É PROGRAMAÇÃO ESTRUTURADA?

- Utilizar o livro da Elisabeth Andrade *Programação Estruturada*
- Disponibilizado em PDF.

```
1  using System;
2
3  namespace HelloWorld
4  {
5      0 references
6      class Program
7      {
8          0 references
9          static void Main(string[] args)
10         {
11             Console.WriteLine("What is your name?");
12             var name = Console.ReadLine();
13
14             Console.WriteLine("What is your year of birth?");
15             var yearOfBirth = int.Parse(Console.ReadLine());
16
17             var currentDate = DateTime.Now;
18             var age = currentDate.Year - yearOfBirth;
19
20             Console.WriteLine(
21                 $"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}!");
22
23             Console.WriteLine(
24                 $"{Environment.NewLine}You are {age} years old.");
25
26             Console.Write($"{Environment.NewLine}Press any key to exit...");
27             Console.ReadKey(true);
28         }
29     }
30 }
31 }
```

```
1 const express = require('express');
2 const app = express();
3 const port = 3000;
4 const axios = require('axios');

5
6 app.get('/', (req, res) => {
7     let cep = req.query.cep;
8     let urlViaCep = `http://viacep.com.br/ws/${cep}/json/`;

9
10    axios.get(urlViaCep)
11        .then(response => {
12            console.log(response.data);
13            res.json(response.data);
14        })
15        .catch(error => {
16            console.error('Erro:', error.message);
17            res.status(500).json({ error: 'Erro ao buscar CEP' });
18        });
19    });
20
21 app.get('/weather', async (req, res) => {
22     try {
23         const city = req.query.city;
24         const apiKey = 'YOUR_API_KEY_HERE';
25         const url = `http://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;
26         const response = await axios.get(url);
27         res.json(response.data);
28     } catch (error) {
29         console.error('Erro:', error.message);
30         res.status(500).json({ error: 'Erro ao buscar previsão do tempo' });
31     }
32 });

33
```

```
33
34 app.get('/exchange-rate', async (req, res) => {
35   try {
36     const baseCurrency = req.query.base;
37     const apiKey = 'YOUR_API_KEY_HERE';
38     const url = `https://v6.exchangeratesapi.io/latest?base=${baseCurrency}&apikey=${apiKey}`;
39     const response = await axios.get(url);
40     res.json(response.data);
41   } catch (error) {
42     console.error('Erro:', error.message);
43     res.status(500).json({ error: 'Erro ao buscar cotação de moedas' });
44   }
45 });
46
47
48 app.listen(port, () => {
49   console.log(`Example app listening on port ${port}`);
50 });
51
```

# O QUE É PROGRAMAÇÃO ORIENTADA A OBJETOS?

- Paradigma de construção de softwares baseado nos conceitos de classe e objeto. Onde a classe representa o template e o objeto a expressão, realização daquele template. Chamamos de instância da classe.
- Ex de CLASSE:
- A Dell projetou um notebook, vários arquitetos de todas as áreas envolvidos fazem um projeto de como deve ser feito o notebook.
- Ex. de OBJETO:
- Uma fábrica x da Dell montou este meu notebook que projeta este slide, isto é um objeto.
- Cada um de vocês tem um objeto específico que também são projetados da mesma classe.

# OBJETO DA POO X OBJETO ESTRUTURADO

- Na Programação Estruturada já existe um conceito de objetos, porém os mesmos são estáticos. Os objetos da POO possuem ações que podem criar novos objetos e realizar diversos procedimentos dentro de um padrão definido na classe.



# O QUE MAIS POO FAZ?

- Possui alguns mecanismos que permitem a reutilização de códigos e formas de relação entre as classes.
- 
- 

# HERANÇA

- O conceito de herança é essencial dentro do paradigma de orientação a objetos pois ele torna possível uma classe “B” ser criada a partir de uma outra “A” sendo que “B” recebe as características de “A” como uma cópia. Isso permite a reutilização de código diminuindo a repetição de informações.
- Para a classe “B” que herda as características de “A” recebe o nome de “classe filha” ou “subclasse”, enquanto “A” é chamada de “classe pai” ou “superclasse”.



# C#

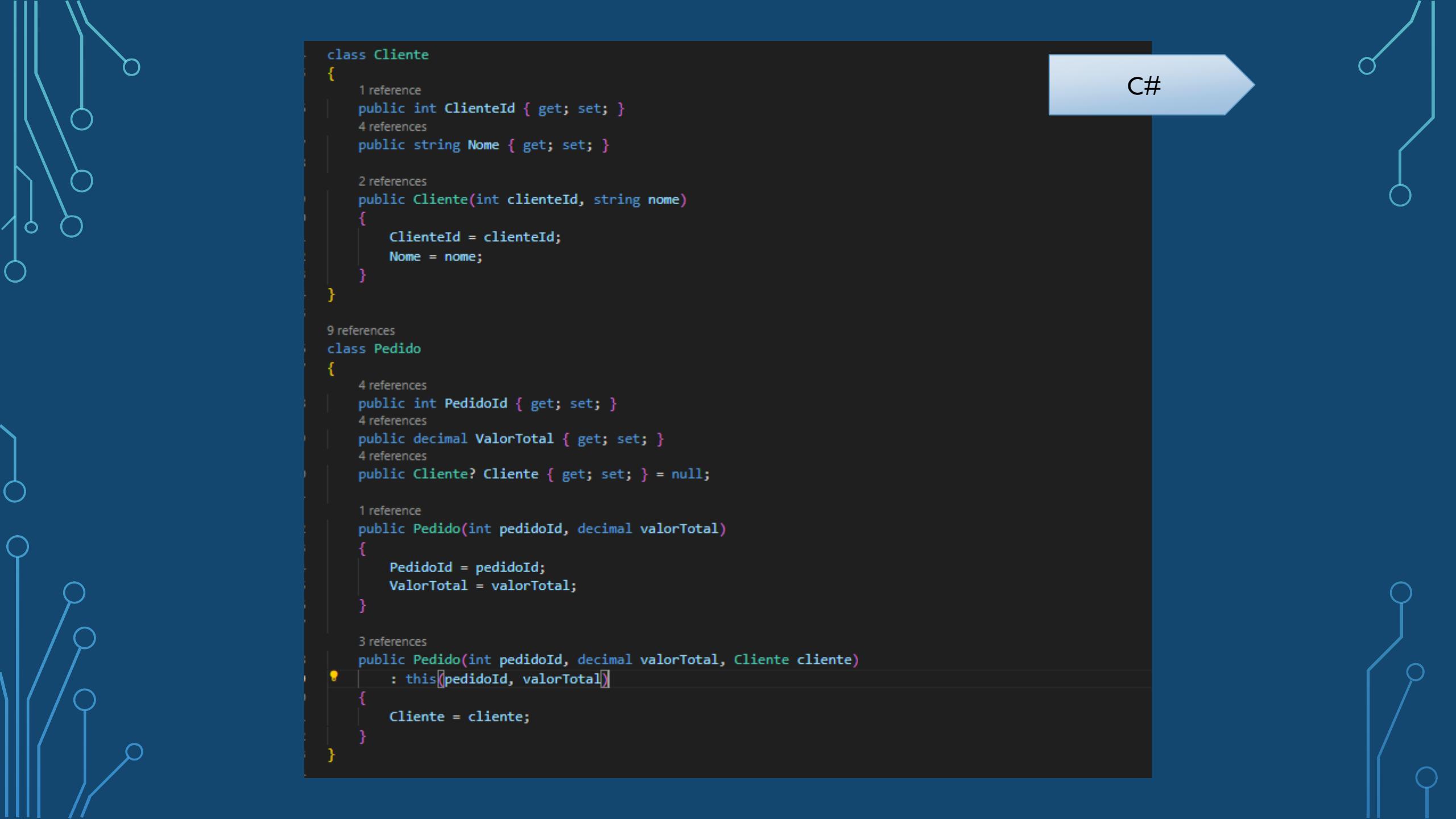
```
1  using System;
2
3  // Classe base (superclasse)
4  class Animal {
5      3 references
6      public string Name { get; set; }
7      1 reference
8      public int Age { get; set; }
9
10     1 reference
11     public void Eat() {
12         Console.WriteLine($"{Name} is eating.");
13     }
14
15     // Classe derivada (subclasse)
16     2 references
17     class Dog : Animal {
18         1 reference
19         public void Bark() {
20             Console.WriteLine($"{Name} is barking.");
21         }
22     }
23 }
```

```
1 references
2
3 class TesteAnimais {
4     0 references
5     static void Execute() {
6         // Criando uma instância da classe derivada (Dog)
7         Dog myDog = new Dog();
8         myDog.Name = "Buddy";
9         myDog.Age = 3;
10
11         // Chamando métodos da classe base (Animal)
12         myDog.Eat();
13
14         // Chamando métodos da classe derivada (Dog)
15         myDog.Bark();
16     }
17 }
```

# ASSOCIAÇÃO

É um tipo de relacionamento entre classes no qual uma classe se utiliza de outra para realizar uma dada tarefa porém não depende desta para existir.

Repare que no exemplo abaixo a classe Pedido possui uma propriedade do tipo Cliente que é opcional e dois construtores, um com o argumento Cliente e outro sem esse objeto para flexibilizar a instanciação de Pedido de forma independente.



C#

```
class Cliente
{
    1 reference
    public int ClienteId { get; set; }
    4 references
    public string Nome { get; set; }

    2 references
    public Cliente(int clienteId, string nome)
    {
        ClienteId = clienteId;
        Nome = nome;
    }

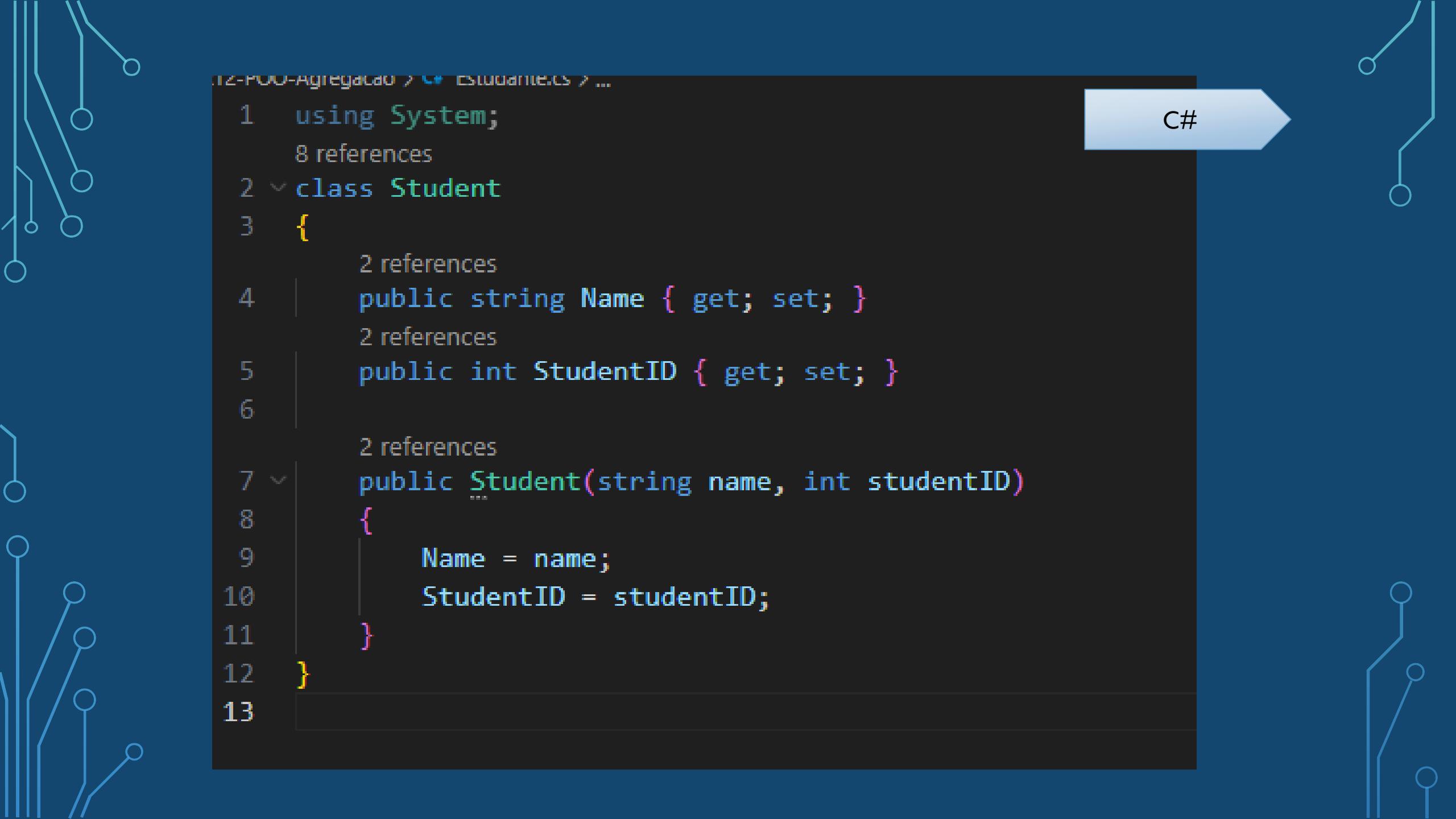
    9 references
    class Pedido
    {
        4 references
        public int PedidoId { get; set; }
        4 references
        public decimal ValorTotal { get; set; }
        4 references
        public Cliente? Cliente { get; set; } = null;

        1 reference
        public Pedido(int pedidoId, decimal valorTotal)
        {
            PedidoId = pedidoId;
            ValorTotal = valorTotal;
        }

        3 references
        public Pedido(int pedidoId, decimal valorTotal, Cliente cliente)
        : this(pedidoId, valorTotal)
        {
            Cliente = cliente;
        }
    }
}
```

# AGREGAÇÃO

- Agregação é a possibilidade de uma classe ser possuir objetos de uma ou mais classes. A classe “agregadora” possui diversas instâncias de outras classes chamadas “agregadas”.



C#

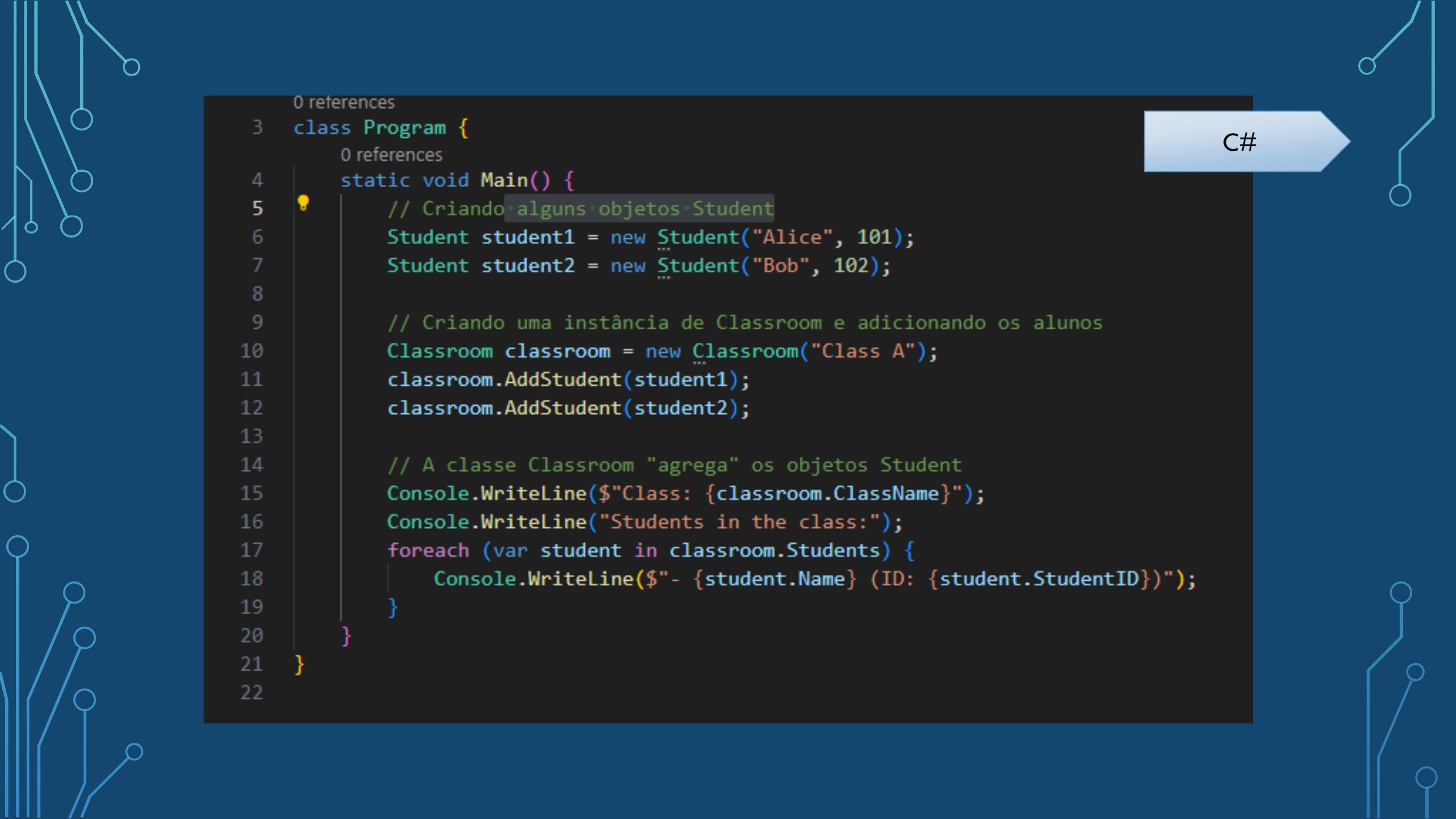
12-POO-Aggregation > Estudiantes > ...

```
1 using System;
  8 references
2 ~ class Student
3 {
  2 references
4   | public string Name { get; set; }
  2 references
5   | public int StudentID { get; set; }
6
  2 references
7   | public Student(string name, int studentID)
8   {
9     |   Name = name;
10    |   StudentID = studentID;
11  }
12}
13
```



C#

```
1  using System.Collections.Generic;
2
3  class Classroom {
4      public string ClassName { get; set; }
5      public List<Student> Students { get; } = new List<Student>();
6
7      public Classroom(string className) {
8          ClassName = className;
9      }
10
11     public void AddStudent(Student student) {
12         Students.Add(student);
13     }
14 }
15
16
```



C#

```
0 references
3 class Program {
    0 references
4     static void Main() {
5         // Criando alguns objetos Student
6         Student student1 = new Student("Alice", 101);
7         Student student2 = new Student("Bob", 102);

8
9         // Criando uma instância de Classroom e adicionando os alunos
10        Classroom classroom = new Classroom("Class A");
11        classroom.AddStudent(student1);
12        classroom.AddStudent(student2);

13
14        // A classe Classroom "agrega" os objetos Student
15        Console.WriteLine($"Class: {classroom.ClassName}");
16        Console.WriteLine("Students in the class:");
17        foreach (var student in classroom.Students) {
18            Console.WriteLine($"- {student.Name} (ID: {student.StudentID})");
19        }
20    }
21 }
22 }
```

# COMPOSIÇÃO

- A relação de composição é uma relação entre classes mais forte do que a agregação se diferenciando pois na agregação a classe agregadora cria instâncias da classe agregada enquanto na composição uma instância da classe agregada é passada para a classe composta (classe que utiliza a outra) de maneira que a classe composta não existe sem os objetos da classe componente.



Python

```
1 class Motor:  
2     def __init__(self, tipo):  
3         self.tipo = tipo  
4  
5     def ligar(self):  
6         print("Motor ligado.")  
7  
8     def desligar(self):  
9         print("Motor desligado.")  
10  
11 You, 5 months ago | 1 author (You)  
12 class Carro:  
13     def __init__(self, motor):  
14         self.motor = motor  
15  
16     def ligar_carro(self):  
17         self.motor.ligar()  
18         print("Carro ligado.")  
19  
20     def desligar_carro(self):  
21         self.motor.desligar() You, 5 months ago • add  
22         print("Carro desligado.")  
23
```

```
motor = Motor("Gasolina")
```

```
carro = Carro(motor)
```

```
carro.ligar_carro()
```

```
carro.desligar_carro()
```

You, 42 seconds ago • Uncommi

```
carro.motor.ligar()
```

```
carro.motor.desligar()
```

# POLIMORFISMO

- Permite que objetos de diferentes classes sejam tratados de maneira uniforme. Essa capacidade de objetos de diferentes classes responderem de maneira semelhante a mensagens ou chamadas de métodos é fundamental para a flexibilidade e extensibilidade de sistemas de software.

# POLIMORFISMO DE SOBRECARGA (COMPILE-TIME POLYMORPHISM)

- Ocorre quando várias versões de um método têm o mesmo nome, mas parâmetros diferentes (tipos ou número). O compilador determina qual versão do método chamar com base nos argumentos passados durante a compilação.

# EXEMPLO DE POLIMORFISMO DE SOBRECARGA EM JAVA:

```
Calculator.java:1 ~ 9
```

```
1 class Calculator {  
2     int add(int a, int b) {  
3         return a + b;  
4     }  
5  
6     double add(double a, double b) {  
7         return a + b;  
8     }  
9 }
```

```
Calculator calc = new Calculator();  
  
calc.add(4, 8);  
  
calc.add(10.20, 15.35);
```

Java

# POLIMORFISMO DE TEMPO DE EXECUÇÃO (RUNTIME POLYMORPHISM OU POLIMORFISMO DE HERANÇA)

- Ocorre quando uma classe filha herda de uma classe pai e substitui (sobrescreve) seus métodos. A decisão sobre qual método chamar é tomada em tempo de execução, com base no tipo real do objeto.

# EXEMPLO DE POLIMORFISMO EM TEMPO DE EXECUÇÃO EM JAVA:

POO > Polimorfismo > TempoExecucao >  Animal.java

Java

```
1 class Animal {  
2     public void MakeSound() {  
3         System.out.println("Animal makes a sound");  
4     }  
5 }  
6  
1 class Cat extends Animal {  
2     @Override  
3     public void MakeSound() {  
4         System.out.println("Cat meows");  
5     }  
6 }  
7  
1 class Dog extends Animal {  
2     @Override  
3     public void MakeSound() {  
4         System.out.println("Dog barks");  
5     }  
6 }
```

# COESÃO E RESPONSABILIDADE

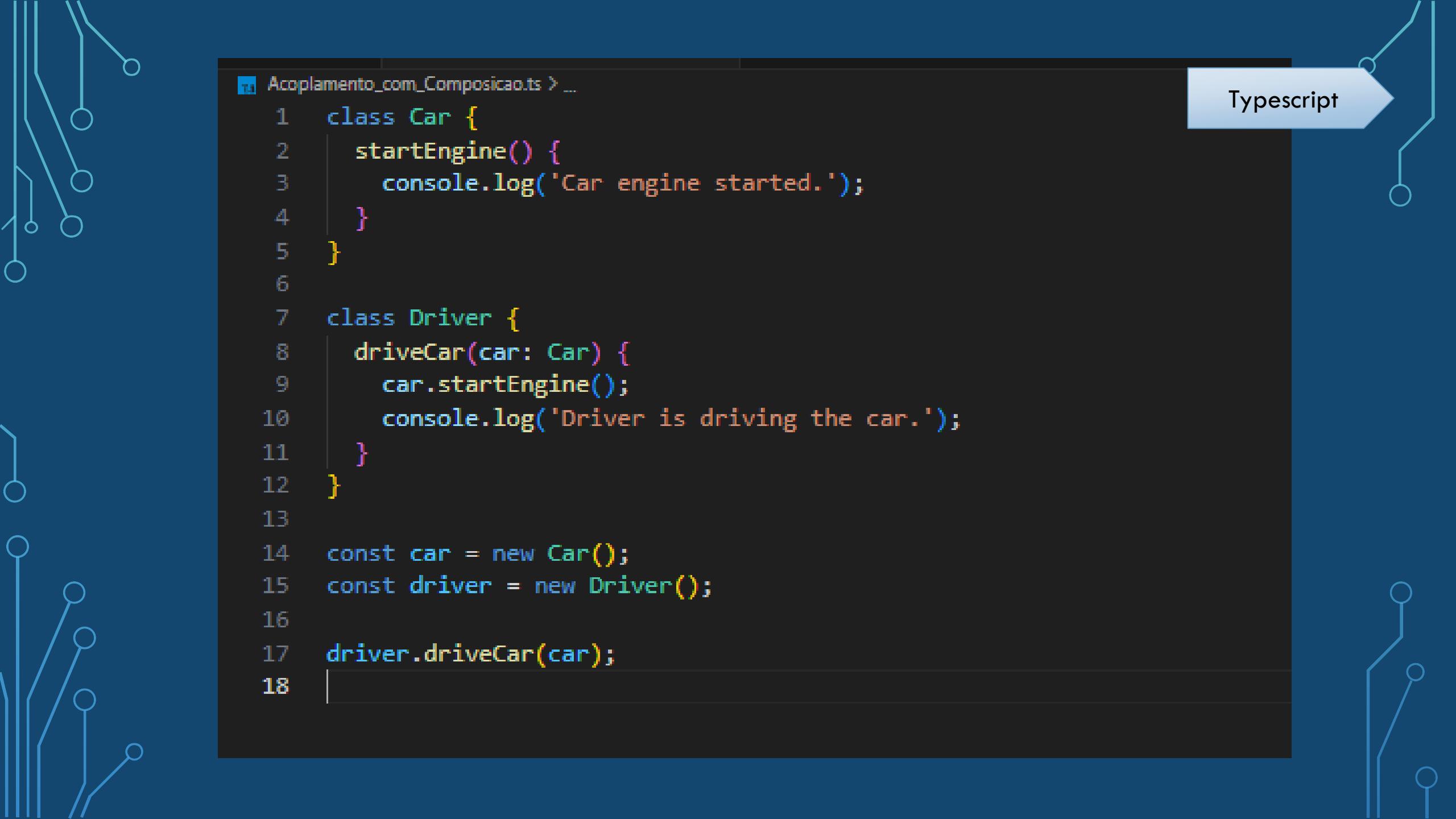
- Uma classe coesa possui apenas uma responsabilidade.
- O que poderia fazer esta classe crescer?

## TypeScript

```
1  class Funcionario {
2    constructor(public cargo: string) { }
3  }
4
5  class CalculadoraDeSalario {
6    private static DESENVOLVEDOR = "DESENVOLVEDOR";
7    private static DBA = "DBA";
8    private static TESTER = "TESTER";
9
10   private static dezOuVintePorcento(funcionario: Funcionario): number [
11     // Implemente a lógica de cálculo para DESENVOLVEDOR aqui
12     // Por exemplo, retornando 10% ou 20% do salário
13     return 0;
14   ]
15
16   private static quinzeOuVinteCincoPorcento(funcionario: Funcionario): number {
17     // Implemente a lógica de cálculo para DBA e TESTER aqui
18     // Por exemplo, retornando 15% ou 25% do salário
19     return 0;
20   }
21
22   public static calcula(funcionario: Funcionario): number {
23     if (CalculadoraDeSalario.DESENVOLVEDOR === funcionario.cargo) {
24       return CalculadoraDeSalario.dezOuVintePorcento(funcionario);
25     }
26     if (
27       CalculadoraDeSalario.DBA === funcionario.cargo ||
28       CalculadoraDeSalario.TESTER === funcionario.cargo
29     ) {
30       return CalculadoraDeSalario.quinzeOuVinteCincoPorcento(funcionario);
31     }
32     throw new Error("Funcionário inválido");
33   }
34 }
```

# ACOPLAMENTO

- Refere-se ao grau de dependência entre as classes. Maior acoplamento significa que mais uma classe depende de outra para funcionar.



TypeScript

```
File: Acoplamento_com_Composicao.ts ...  
1 class Car {  
2     startEngine() {  
3         console.log('Car engine started.');//  
4     }  
5 }  
6  
7 class Driver {  
8     driveCar(car: Car) {  
9         car.startEngine();  
10        console.log('Driver is driving the car.');//  
11    }  
12 }  
13  
14 const car = new Car();  
15 const driver = new Driver();  
16  
17 driver.driveCar(car);  
18 |
```

## TypeScript

```
Desacoplando_Com_interface.ts > ...
1  interface IVehicle {
2      startEngine(): void;
3  }
4
5  class Carro implements IVehicle {
6      startEngine() {
7          console.log('Motor ligado.');
8      }
9  }
10
11 class Motorista {
12     dirigeVeiculo(vehicle: IVehicle) {
13         vehicle.startEngine();
14         console.log('Driver is driving the vehicle.');
15     }
16 }
17
18 const carro = new Carro();
19 const motorista = new Motorista();
20
21 motorista.dirigeVeiculo(carro);
22
```

# ENCAPSULAMENTO

Permite ocultar detalhes internos da classe gerenciando o nível de segurança para cada membro de uma classe (método ou atributo). Em vez disso, o acesso a esses detalhes é controlado por meio de métodos públicos que fornecem uma maneira limitada para interação com o objeto.

# MODIFICADORES DE ACESSO

- **Private:** só pode ser acessado dentro da própria classe em que foi definido. É a forma mais restrita de acesso.
- **Public:** pode ser acessado de qualquer lugar no código, tanto dentro da classe que o define quanto externamente.
- **Protected:** pode ser acessado dentro da classe em que foi definido e também por subclasses que herdam dessa classe.
- **Internal:** pode ser acessado apenas dentro do assembly (conjunto de código compilado) em que está definido. É uma forma de limitar o acesso a componentes dentro de um projeto ou assembly específico.

## TypeScript

```
1 class Person {  
2     private name: string;  
3     private age: number;  
4  
5     constructor(name: string, age: number) {  
6         this.name = name;  
7         this.age = age;  
8     }  
9  
10    public greet() {  
11        console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);  
12    }  
13  
14    public getAge() {  
15        return this.age;  
16    }  
17  
18    public setAge(age: number) {  
19        if (age >= 0) {  
20            this.age = age;  
21        } else {  
22            console.error('Age cannot be negative.');  
23        }  
24    }  
25}  
26
```

```
const person = new Person('Alice', 30);  
  
person.greet();  
console.log(`Age: ${person.getAge()}`);  
person.setAge(35);
```

## TypeScript

```
1  class Encryptor {
2      encrypt(data: string): string {
3          // Implementação da lógica de encriptação
4          return `Encrypted: ${data}`;
5      }
6  }
7
8  class DataService {
9      private encryptor: Encryptor;
10
11     constructor() {
12         this.encryptor = new Encryptor(); // Dependência forte
13     }
14
15    saveData(data: string): void {
16        const encryptedData = this.encryptor.encrypt(data);
17        // Salvar os dados encriptados
18    }
19
20
21 const dataService = new DataService();
22 dataService.saveData("Dados confidenciais");
23
```

# CÓDIGO LIMPO

**Michael Feathers, autor de *Working Effectively with Legacy Code***

*Eu poderia listar todas as qualidades que vejo em um código limpo, mas há uma predominante que leva a todas as outras. Um código limpo sempre parece que foi escrito por alguém que se importava. Não há nada de óbvio no que se pode fazer para torná-lo melhor. Tudo foi pensado pelo autor do código, e se tentar pensar em algumas melhorias, você voltará ao início, ou seja, apreciando o código deixado para você por alguém que se importa bastante com essa tarefa.*

One word: care. É esse o assunto deste livro. Talvez um subtítulo apropriado seria *Como se importar com o código*.



# LEITURA COMO NARRATIVA

O código deve ser legível por outras pessoas sem grande esforço intelectual.

O código é em si a própria documentação se bem escrito.

# NOMES SIGNIFICATIVOS

Tudo deve ser bem nomeado e renomeado para que o nome represente a função no sistema. Variáveis e métodos com nomes muitos genéricos dificultam a manutenção e engana os leitores.

# RESPONSABILIDADE ÚNICA

Cada classe deve ser responsável por apenas uma responsabilidade.

“Uma classe deve ter um único motivo para mudar”.

*AS FUNÇÕES DEVEM FAZER UMA COISA. DEVEM FAZÊ-LA BEM.  
DEVEM FAZER APENAS ELA.*

# PARÂMETROS

## Parâmetros de Funções

A quantidade ideal de parâmetros para uma função é zero (nulo). Depois vem um (mônade), seguido de dois (diade). Sempre que possível devem-se evitar três parâmetros (triade). Para mais de três deve-se ter um motivo muito especial (políade) – mesmo assim não devem ser usados.



# COMENTÁRIOS MÍNIMOS E SIGNIFICATIVOS

- Os códigos mudam mas os comentários não;
- Muitas vezes os comentários mentem sobre o que o código faz;
- Podem gerar bugs se descomentados accidentalmente;
- Demonstram que o código está mal escrito e necessita de explicação;
- Podem ser substituídos por ferramentas de controle de versão ou bons nomes;

# FORMATAÇÃO CORRETA

Um código identado é mais fácil de ser lido;

Evitar o deslise da tela na horizontal e vertical o máximo possível para facilitar a leitura e manutenção;

Manter padrão dentro de cada sistema ou módulo;

# TRATAMENTO DE ERROS

“Prefira exceções a retorno de código de erro.”

Isso reduz a quantidade de código pois o código que utilizar a função que retorna código de erro deverá examinar como trata-lo em um algoritmo.

Disparar `throw` facilita a separação da lógica de erros e permite a customização de diversos tipos específicos de exceção esperada.

# TESTES DE UNIDADE

## As três leis do TDD

Hoje em dia todos sabem que o TDD nos pede para criar primeiro os testes de unidade antes do código de produção. Mas essa regra é apenas o início. Considere as três leis<sup>1</sup> abaixo:

**Primeira Lei** Não se deve escrever o código de produção até criar um teste de unidade de falhas.

**Segunda Lei** Não se deve escrever mais de um teste de unidade do que o necessário para falhar, e não compilar é falhar.

**Terceira Lei** Não se deve escrever mais códigos de produção do que o necessário para aplicar o teste de falha atual.

# BAIXO ACOPLAMENTO

Diminuir a dependência entre as classes permite que os motivos para mudança  
sejam menores facilitando a criação de classes pequenas.

# SOLID

Acrônimo para cinco princípios de design de software que visam criar sistemas mais legíveis, escaláveis e fáceis de manter.

# PRINCÍPIO DA RESPONSABILIDADE ÚNICA

Tal como Uncle Bob ensina no livro Código Limpo, uma classe ou função deve ter um nome significativo que represente o que ela faz e fazer apenas uma coisa. O nome da classe e as responsabilidades dela estão intimamente ligados. Uma classe com apenas uma responsabilidade diminui o número de pontos a serem alterados quando mudar algum requisito ou lógica.

# PRINCÍPIO DO ABERTO-FECHADO

As classes e módulos de um sistema de software devem estar abertos para extensão, mas fechados para modificação. Aberto para extensão significa que, ao receber um novo requerimento, é possível adicionar um novo comportamento. Fechado para modificação significa que, para introduzir um novo comportamento (extensão), não é necessário modificar o código existente.

# PRINCÍPIO DE SUBSTITUIÇÃO DE LISKOV

Se uma classe S é um subtipo (subclasse) de uma classe T, então os objetos da classe T podem ser substituídos por objetos da classe S sem que isso altere a corretude do programa.

```
static void MakeAnimalSound(Animal animal)
{
    Console.WriteLine(animal.EmitSound());
}

static void Main(string[] args)
{
    Animal dog = new Dog();
    Animal cat = new Cat();

    MakeAnimalSound(dog); // Deve imprimir "Latido de cachorro"
    MakeAnimalSound(cat); // Deve imprimir "Miado de gato"
}
```

# PRINCÍPIO DE SEGREGAÇÃO DE INTERFACE

Cientes não devem ser forçados a depender de interfaces que não utilizem.

Diz que as interfaces devem ser específicas (ou pequenas) para que as classes possam implementar somente os comportamentos necessários.

Cientes não devem ser afetados por mudanças na interface.

# EXEMPLO DE INTERFACE RUIM

```
2 references
public interface IPessoa
{
    2 references
    void Trabalhar();
    2 references
    void Comer();
    2 references
    void Dormir();
    2 references
    void Estudar();
}
```

C#

C#

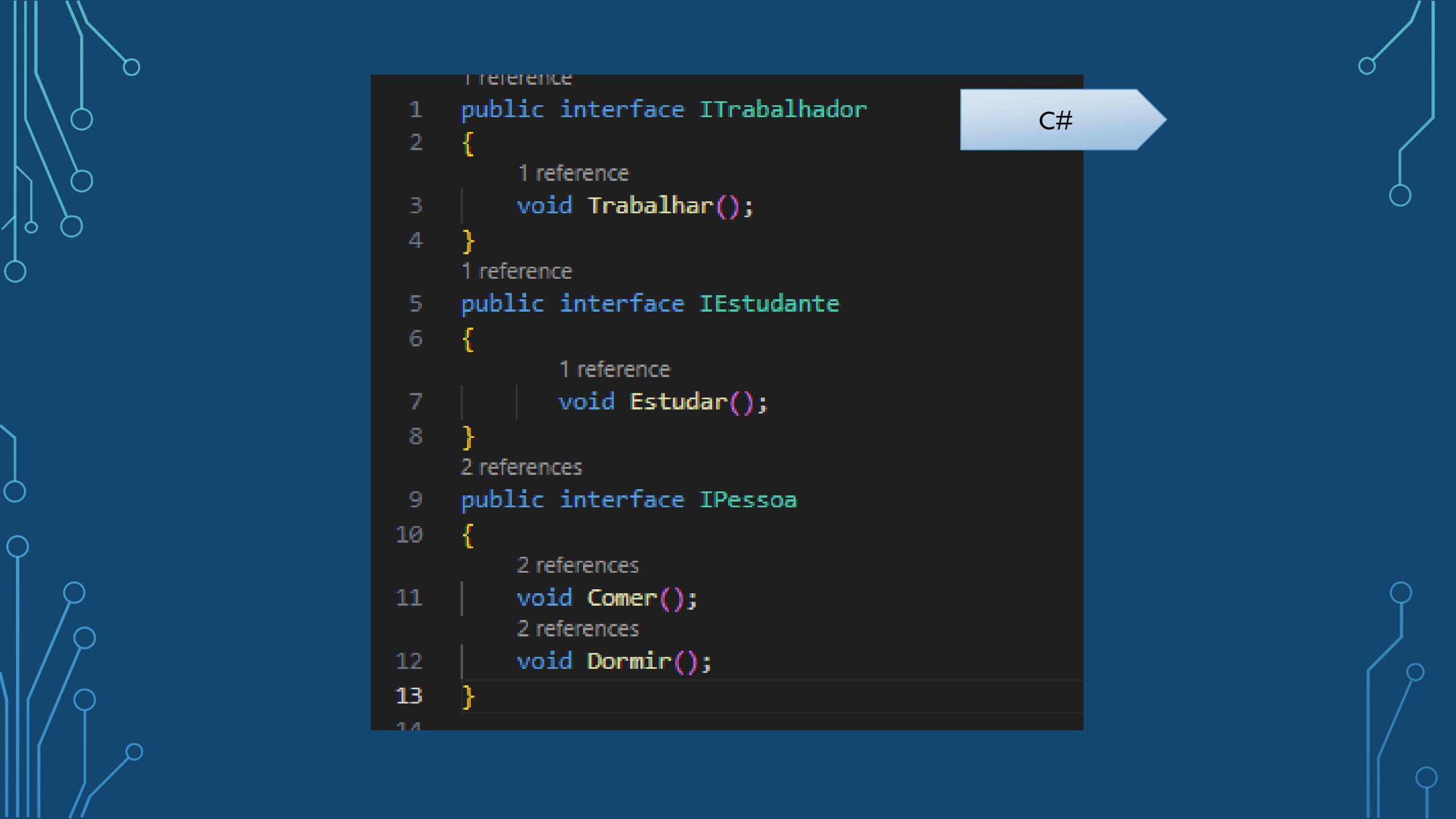


C#

```
0 references
1 public class Estudante : IPessoa
2 {
3     1 reference
4     public void Trabalhar()
5     {
6         // Implementação do trabalho de um estudante (pode ser vazio, já que estudantes podem não trabalhar)
7     }
8
9     1 reference
10    public void Comer()
11    {
12        // Implementação da pausa para comer
13    }
14
15    1 reference
16    public void Dormir()
17    {
18        // Implementação da pausa para dormir
19    }
20
21    1 reference
22    public void Estudar()
23    {
24        // Implementação da pausa para estudar
25    }
26}
```

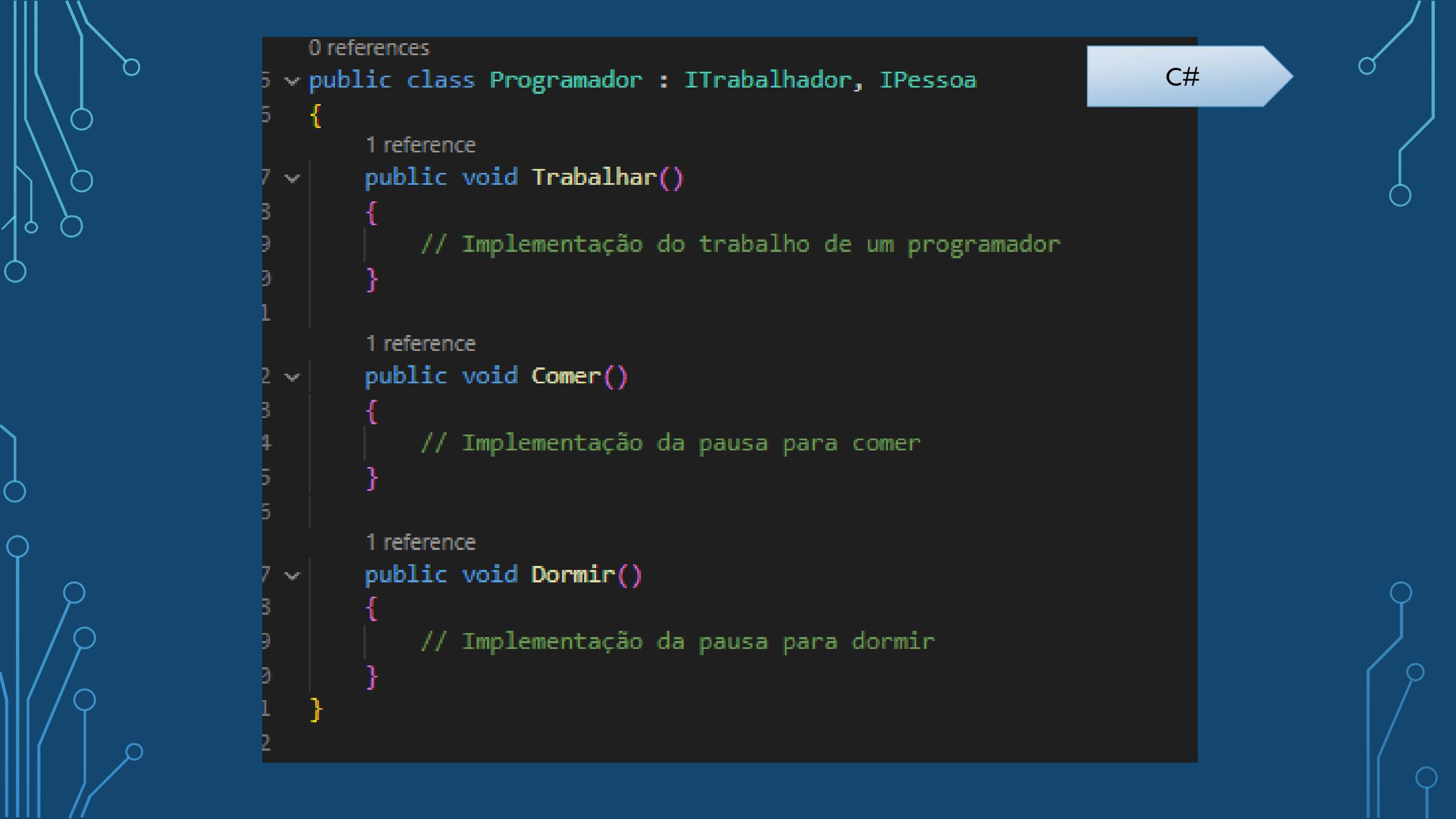


# SEGREGANDO AS INTERFACES



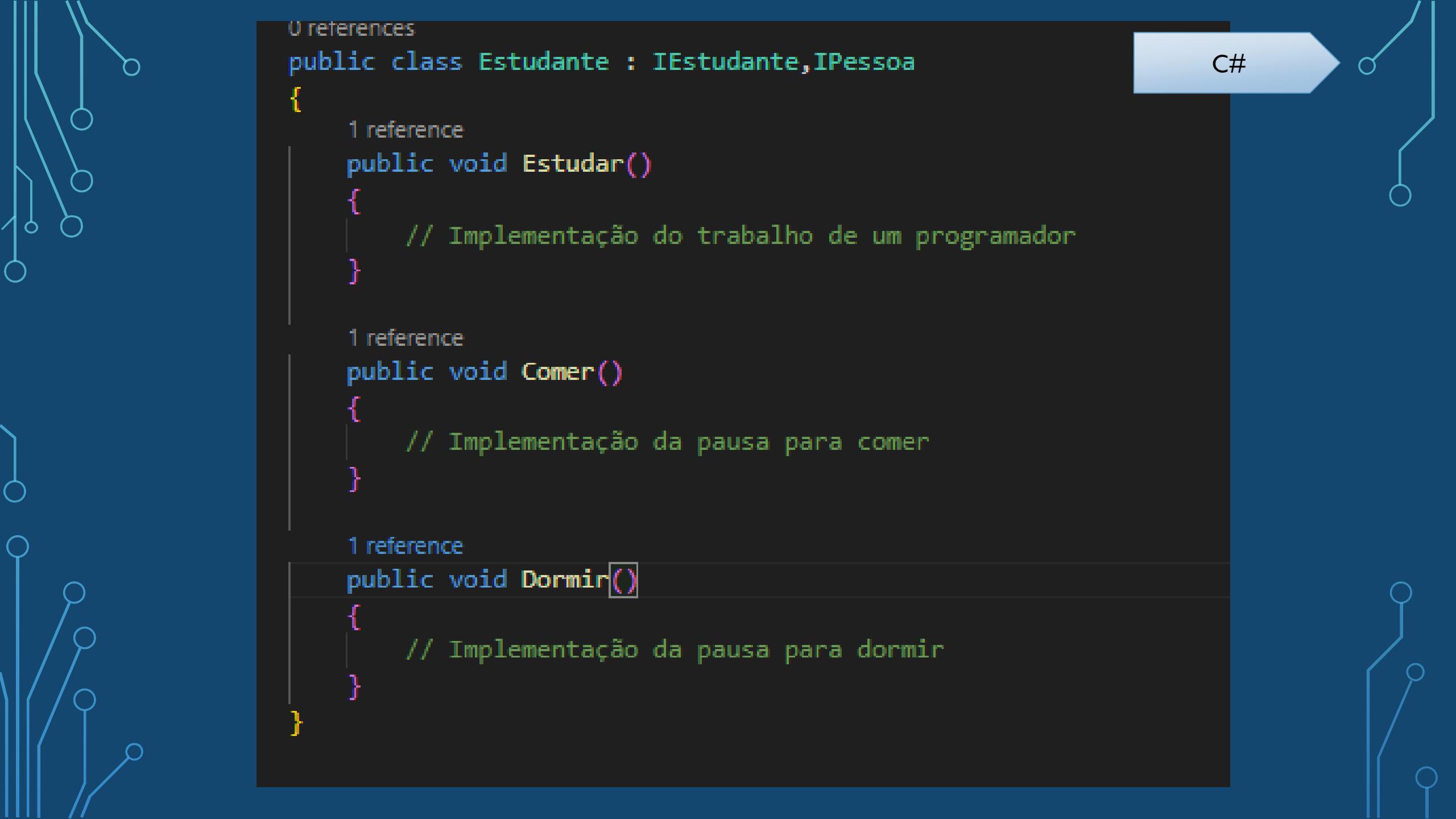
C#

```
1  public interface ITrabalhador
2  {
3      void Trabalhar();
4  }
5  public interface IEstudante
6  {
7      void Estudar();
8  }
9  public interface IPessoa
10 {
11     void Comer();
12     void Dormir();
13 }
```



C#

```
0 references
5 ✓ public class Programador : ITrabalhador, IPessoa
5 {
    1 reference
7 ✓     public void Trabalhar()
3 {
9     // Implementação do trabalho de um programador
3 }
1
1 reference
2 ✓     public void Comer()
3 {
4     // Implementação da pausa para comer
5 }
5
5
1 reference
7 ✓     public void Dormir()
3 {
9     // Implementação da pausa para dormir
3 }
1
2 }
```



C#

0 references

```
public class Estudante : IEstudante, IPessoa
{
    1 reference
    public void Estudar()
    {
        // Implementação do trabalho de um programador
    }

    1 reference
    public void Comer()
    {
        // Implementação da pausa para comer
    }

    1 reference
    public void Dormir()
    {
        // Implementação da pausa para dormir
    }
}
```



# INVERSÃO DE CONTROLE (IOC – INVERSION OF CONTROL)

# INVERSÃO DE CONTROLE

É um princípio de design de software que envolve a inversão de controle do fluxo de operações em um sistema.



# INJEÇÃO DE DEPENDÊNCIAS

# O QUE SÃO DEPENDÊNCIAS?

Dependências são os objetos de outras classes que uma classe ou módulo precisa para funcionar.

# O QUE SIGNIFICA INJEÇÃO?

Injetar dependências significa fornecer essas dependências a uma classe ou módulo a partir do exterior, em vez de permitir que a classe crie ou gerencie suas próprias dependências.

# BIBLIOGRAFIA

GONÇALVES, B. P.; LAZARO MENDES, O. . PRINCÍPIO DA INVERSÃO DE DEPENDÊNCIA NA QUALIDADE DE SOFTWARE: Aplicação da injeção de dependência no desenvolvimento de software. Revista Interface Tecnológica, [S. I.], v. 19, n. 1, p. 34–46, 2022. DOI: 10.31510/infa.v19i1.1362. Disponível em: <https://revista.fatectq.edu.br/interfacetecnologica/article/view/1362>. Acesso em: 10 out. 2023.

BETTS, ET. AL. Dependency injection with unity. Washington: Microsoft, 2013.

Utilização dos princípios SOLID na aplicação de Padrões de Projeto, Dev Media, 2023. Disponível em <<https://www.devmaster.com.br/utilizacao-dos-principios-solid-na-aplicacao-de-padroes-de-projeto/25369>> Acesso em: 10 out. 2023.

ANDRADE, Elizabeth Alves. Programação Estruturada. 2017.

Documentação do Jestjs. Disponível em<<https://jestjs.io/pt-BR/>> Acesso em 10/10/23.

MARTIN, Robert C. Código Limpo: Habilidades Práticas do Agile Software. [S. I.]: Alta