SOLID

Rodrigo Machado

1 *S*OLID - Princípio da Responsabilidade Única (SRP)

Introduzido por Robert C. Martin, o Princípio da Responsabilidade Única promove uma abordagem sustentável e modular para o desenvolvimento de sistemas. O SRP parte do princípio de que uma classe deve ter apenas uma razão para existir, o que ajuda a segregação das responsabilidades no sistema. Ao aderir ao SRP, os programadores podem criar um código que é fácil de entender, manter e expandir.

Em essência, o SRP afirma que uma classe deve ter uma única responsabilidade bem definida e deve executá-la de maneira eficaz. Esse princípio encoraja a divisão de funcionalidades complexas em classes menores e mais específicas, cada uma responsável por uma única tarefa. Ao manter as responsabilidades separadas, alterações e atualizações em uma parte do código têm menos chance de afetar outras partes do sistema, reduzindo assim o risco de efeitos colaterais e bugs.

Exemplo: Para ilustrar, considere uma aplicação simples que gerencia informações de funcionários. Suponha que tenhamos uma classe Employee que lida tanto com os dados do funcionário (nome, cargo) quanto com o cálculo do salário:

```
0 referências
public class Employee
{
    0 referências
    public string Name { get; set; }
    2 referências
    public string Designation { get; set; }
    3 referências
    public double Salary { get; set; }

    0 referências
    public double CalculateSalary()
    {
        if (Designation.Equals("Manager"))
        {
            return Salary * 1.2;
        }
        else if (Designation.Equals("Developer"))
        {
            return Salary * 1.1;
        }
        else
        {
            return Salary;
        }
    }
}
```

Neste exemplo, a classe Employee viola o princípio da responsabilidade única por conter duas responsabilidades: gerenciar dados de funcionários e calcular salários com base no cargo. Essa mistura de responsabilidades pode resultar em dificuldades para modificar e adicionar novas funcionalidades à aplicação conforme ela cresce.

Aplicando o SRP, podemos separar as responsabilidades de gerenciamento de informações do funcionário e cálculo de salário, criando duas classes distintas:

```
1 referência
public class Employee
{
    0 referências
    public string Name { get; set; }
    2 referências
    public string Designation { get; set; }
    3 referências
    public double Salary { get; set; }
}
0 referências
public class Salary
{
    0 referências
    public double CalculateSalary(Employee employee)
    {
        if (employee.Designation.Equals("Manager"))
        {
            return employee.Salary * 1.2;
        }
        else if (employee.Designation.Equals("Developer"))
        {
            return employee.Salary * 1.1;
        }
        else
        {
            return employee.Salary;
        }
    }
}
```

Agora temos duas classes separadas:

1. Employee: Responsável pelo gerenciamento de dados dos funcionários.
2. Salary: Classe dedicada exclusivamente ao cálculo do salário com base no cargo do funcionário.

1.1 Benefícios do SRP:

- Modularidade: Cada classe que se concentra em uma única responsabilidade pode ser alterada, testada e reutilizada de forma independente, sem afetar outras partes do sistema. Isso promove um código mais modular.
- Legibilidade: O código se torna mais fácil de entender, pois cada classe é concisa e representa apenas um aspecto específico ou função.

- Manutenibilidade: Ao fazer alterações, os programadores podem focar apenas na parte específica do código relacionada à mudança, reduzindo o risco de introduzir bugs em outras partes do sistema.
- Escalabilidade: À medida que a aplicação cresce, a gestão de responsabilidades separadas facilita a adição de novas funcionalidades sem impactar as funcionalidades existentes.

Em resumo, o uso do Princípio da Responsabilidade Única melhora a qualidade e a manutenibilidade do código, resultando em um software mais coeso e focado. Criar classes com responsabilidades bem definidas permite aos programadores construir sistemas robustos e adaptáveis, que são mais fáceis de compreender e de escalar conforme as necessidades.

2 S*O*LID - Princípio Aberto/Fechado (OCP)

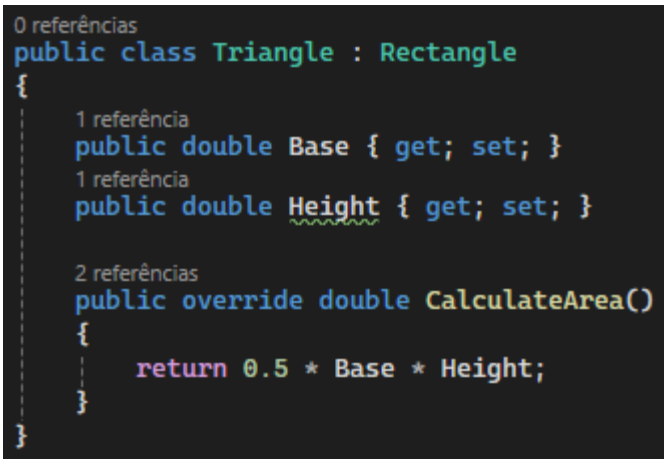
O Princípio Aberto/Fechado (OCP) foi introduzido por Bertrand Meyer e popularizado posteriormente por Robert C. Martin. Este princípio enfatiza que entidades de software (classes, módulos, funções etc.) devem ser abertas para extensões, mas fechadas para modificações. Em outras palavras, uma vez que um módulo esteja desenvolvido e funcionando corretamente, ele não deve ser alterado para adicionar novas funcionalidades ou alterar seu comportamento. Em vez disso, o módulo deve permitir extensões através de herança ou interfaces para atender novos requisitos.

O OCP encoraja os desenvolvedores a projetarem sistemas de forma que minimizem a necessidade de alterações no código existente para adicionar novas funcionalidades. Isso ajuda a reduzir o risco de introdução de bugs em funções que já funcionam corretamente e promove a estabilidade do sistema, pois o código existente permanece intacto enquanto novas funcionalidades são adicionadas.

Exemplo: Considere uma hierarquia de classes com herança para calcular a área de diferentes formas geométricas, como retângulos e círculos:

```
1 referência
public abstract class Shape
{
    2 referências
    public abstract double CalculateArea();
}

1 referência
public class Rectangle : Shape
{
    1 referência
    public double Width { get; set; }
    1 referência
    public double Height { get; set; }

    2 referências
    public override double CalculateArea()
    {
        return Width * Height;
    }
}

0 referências
public class Circle : Rectangle
{
    1 referência
    public double Radius { get; set; }

    2 referências
    public override double CalculateArea()
    {
        return Math.PI * Math.Pow(Radius, 2);
    }
}
```

Com a implementação acima, as classes Shape, Rectangle e Circle seguem o princípio OCP. Se quisermos adicionar uma nova forma, como um triângulo, podemos fazê-lo sem modificar as classes existentes. Ao definir uma classe abstrata que especifica os métodos necessários para qualquer objeto Shape, podemos criar qualquer número de novas formas que calculam a área:



Adicionando uma nova classe à hierarquia, podemos introduzir novas formas sem alterar o código existente, mantendo o OCP. Essa abordagem preserva a estabilidade do código existente e assegura que mudanças nos novos requisitos não afetem o código já implementado.

2.1 Benefícios do OCP:

- Estabilidade: O código existente permanece intacto, reduzindo o risco de bugs e erros em funcionalidades já existentes.
- Extensibilidade: Novas funcionalidades podem ser adicionadas apenas criando novas classes, em vez de modificar as classes existentes, promovendo a reutilização de código e a extensibilidade do sistema.
- Escalabilidade: O sistema se torna mais escalável e adaptável, permitindo a inclusão de novas funcionalidades sem alterar funcionalidades já existentes.

Em resumo, o OCP encoraja os programadores a projetarem classes e módulos que sejam abertos para extensão através de herança ou interfaces, enquanto são fechados para modificações. Esse princípio promove estabilidade, reusabilidade e adaptabilidade, aumentando a manutenibilidade e robustez do sistema quando novas implementações são necessárias.

3 SO*L*ID - Princípio de Substituição de Liskov (LSP)

O Princípio de Substituição de Liskov (LSP) foi introduzido por Barbara Liskov em 1987. Esse princípio estipula que objetos de uma superclasse devem ser substituíveis por objetos

de suas subclasses sem afetar o funcionamento do programa. Em outras palavras, a subclasse deve poder assumir o papel da superclasse em qualquer contexto sem modificar o comportamento desejado do programa.

O LSP é crucial para manter o comportamento correto de um sistema que utiliza herança. Ele assegura que as classes derivadas não introduzam comportamentos inesperados ou violem as expectativas dos clientes que as utilizam.

Exemplo: Para ilustrar o LSP, consideremos o exemplo clássico envolvendo a hierarquia de classes Shape. Temos uma classe base Shape com um método virtual Area() para calcular a área de diferentes formas, e então derivamos duas classes: Rectangle e Circle.

```
2 referências
public class Shape
{
    2 referências
    public virtual double Area()
    {
        return 0;
    }
}

0 referências
public class Rectangle : Shape
{
    1 referência
    public double Width { get; set; }
    1 referência
    public double Height { get; set; }

    1 referência
    public override double Area()
    {
        return Width * Height;
    }
}

0 referências
public class Circle : Shape
{
    1 referência
    public double Radius { get; set; }
    1 referência
    public override double Area()
    {
        return Math.PI * Math.Pow(Radius, 2);
    }
}
```

Neste exemplo, tanto Rectangle quanto Circle estendem adequadamente a classe base Shape, fornecendo suas próprias implementações para o método Area(). Isso é uma aplicação do LSP, pois podemos substituir qualquer instância de Rectangle ou Circle por uma instância de Shape sem quebrar o programa.

Para exemplificar uma violação do princípio, introduzimos uma classe Square que herda de Rectangle. Um quadrado é um caso especial de retângulo onde todos os lados têm o mesmo comprimento.

```
public class Square : Rectangle
{
    2 referências
    public double Side { get; set; }

    2 referências
    public override double Area()
    {
        return Side * Side;
    }
}
```

À primeira vista, parece lógico que Square herde de Rectangle, pois um quadrado é um tipo específico de retângulo. No entanto, essa implementação viola o LSP. Considere o seguinte código:

```
public class CalculateAreaOfShape(Shape shape)
{
    double area = shape.Area();
}
```

Se passarmos uma instância de Square para a classe CalculateAreaOfShape, o cálculo da área será feito incorretamente, pois a classe Square reescreve o método Area() para tratar width e height com o mesmo valor. Isso viola o comportamento esperado.

Para seguir o princípio, devemos reformular a hierarquia utilizando composição em vez de herança. Ou seja, Square deve ser uma classe independente que também herda de Shape.

```
public class Square : Shape
{
    2 referências
    public double Side { get; set; }

    2 referências
    public override double Area()
    {
        return Side * Side;
    }
}
```

Nesta versão revisada, o quadrado é tratado como uma entidade própria, independente do retângulo. Ambas as classes seguem o modelo da classe Shape sem violar o princípio LSP. Agora, o comportamento do programa permanece consistente ao lidar com objetos de Shape, Rectangle e Square.

3.1 Benefícios do LSP:

- Código Correto: Seguir o LSP assegura que as classes derivadas se comportem como esperado e não introduzam erros.
- Extensibilidade: Permite a criação de subclasses sem afetar os comportamentos existentes das superclasses.
- Reutilização de Código: Clientes podem depender da superclasse, permitindo trabalhar com qualquer subclasse de forma consistente.

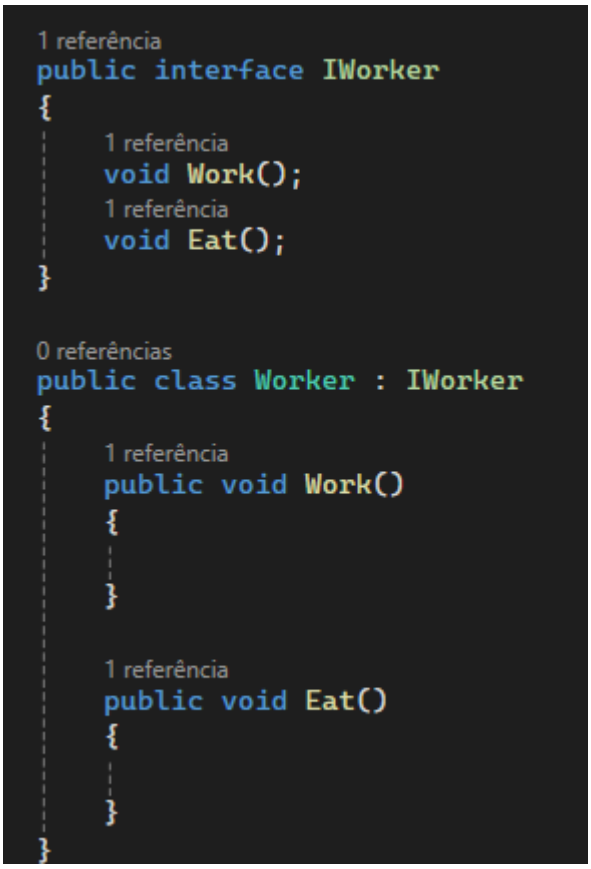
Em resumo, o LSP guia os programadores a projetarem hierarquias de classes que mantenham o comportamento esperado quando as subclasses substituem as superclasses. Respeitar este princípio permite que os programadores criem aplicações robustas, escaláveis e de fácil manutenção.

4 SOL*I*D - Princípio de Segregação de Interfaces (ISP)

O Princípio de Segregação de Interfaces (ISP) foi criado por Robert C. Martin para enfatizar que os clientes não devem depender de interfaces que não utilizam. ISP promove a ideia de criar interfaces pequenas e coesas, focadas nas necessidades específicas dos clientes, ao invés de interfaces grandes, monolíticas, que englobam todos os comportamentos possíveis. Esse princípio visa reduzir o acoplamento entre classes e melhorar a manutenabilidade e flexibilidade dos sistemas.

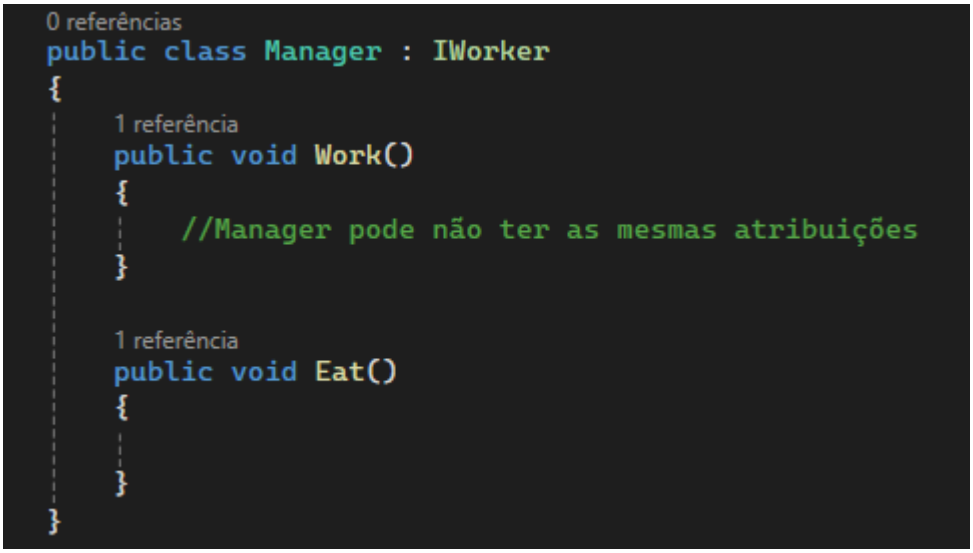
O ISP estipula que nem todos os clientes precisam ou utilizam todos os métodos expostos em uma única interface. Isso evita sobrecarga e responsabilidades desnecessárias, o que resulta em sistemas mais fáceis de manter e entender. Além disso, alterações em interfaces grandes podem afetar todas as classes que as implementam, introduzindo mudanças desnecessárias em áreas não relacionadas do sistema.

Exemplo: Considere um exemplo envolvendo a interface IWorker, que representa diferentes papéis em uma organização. Inicialmente, temos uma classe Worker simples que implementa a interface IWorker:



Neste exemplo, a classe Worker implementa corretamente a interface IWorker, que inclui os métodos Work() e Eat(). No entanto, e se tivermos outro tipo de trabalhador, como um Manager, que não apenas realiza trabalho operacional como os trabalhadores regulares, mas também possui responsabilidades adicionais?

Isso pode levar a uma violação do ISP se continuarmos utilizando a mesma interface IWorker. Por exemplo, o Manager não precisa implementar o método Work(), pois suas responsabilidades podem estar mais voltadas para a gestão das atividades.



Aqui, a classe Manager é obrigada a implementar o método Work(), mesmo que isso não faça sentido para ela. Essa situação pode resultar em confusão no código e dependências desnecessárias.

Para aplicar o ISP corretamente, podemos dividir a interface IWorker em interfaces menores e mais específicas:

```
1 referência
public interface IWorker
{
    1 referência
    void Work();
}
2 referências
public interface IEater
{
    2 referências
    void Eat();
}
0 referências
public class Worker : IWorker, IEater
{
    1 referência
    public void Work()
    {

    }

    1 referência
    public void Eat()
    {

    }
}
0 referências
public class Manager : IEater
{
    1 referência
    public void Eat()
    {

    }
}
```

Ao criar interfaces separadas (IWorker e IEater), cada uma representando de forma coesa um conjunto específico de comportamentos, evitamos impor métodos desnecessários a classes que não precisam deles. Agora, a classe Manager pode implementar apenas a interface IEater sem a necessidade de implementar a interface IWorker ou o método Work().

4.1 Benefícios do ISP:

- Redução de Acoplamento: Interfaces pequenas reduzem as dependências entre classes, tornando o código mais claro, fácil de manter e menos propenso a efeitos colaterais.
- Flexibilidade: Clientes podem implementar interfaces específicas para suas necessidades, permitindo adaptação e extensão futura do código de forma mais eficiente.
- Legibilidade de Código: Interfaces coesas tornam o código mais legível e compreensível.

Em conclusão, o ISP encoraja os desenvolvedores a criar interfaces coesas e específicas para cada necessidade, evitando interfaces grandes e monolíticas. Isso torna o código mais fácil de manter, mais flexível, legível e robusto, promovendo uma maior escalabilidade do sistema.

5 SOLID - Princípio da Inversão de Dependência (DIP)

O Princípio da Inversão de Dependência (DIP), introduzido por Robert C. Martin, aborda problemas relacionados ao acoplamento forte e promove flexibilidade e manutenibilidade em sistemas. DIP estabelece que módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações. Além disso, abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações. Em resumo, DIP encoraja o uso de interfaces ou abstrações para desacoplar componentes, facilitando substituições e extensões sem afetar o sistema como um todo.

Exemplo: Vamos ilustrar o Princípio da Inversão de Dependência com um exemplo de uma simples classe Logger que escreve mensagens de log para um local específico, como o console ou um arquivo de texto.

```
1 referência
public class Logger
{
    private ConsoleLogger _consoleLogger;
    private FileLogger _fileLogger;

    0 referências
    public Logger()
    {
        _consoleLogger = new ConsoleLogger();
        _fileLogger = new FileLogger();
    }

    0 referências
    public void LogToConsole(string mensagem)
    {
        _consoleLogger.Log(mensagem);
    }

    0 referências
    public void LogToFile(string mensagem)
    {
        _fileLogger.Log(mensagem);
    }
}
```

Neste exemplo, a classe Logger depende diretamente das implementações concretas ConsoleLogger e FileLogger. Isso cria um acoplamento rígido, onde alterações na classe Logger estão intimamente ligadas às alterações nas classes concretas de log.

Aplicando o DIP, podemos introduzir uma interface ILogger e fazer com que a classe Logger dependa de abstrações em vez de implementações concretas:

```
public interface ILogger
{
    4 referências
    void Log(string message);
}
0 referências
internal class ConsoleLogger : ILogger
{
    3 referências
    public void Log(string message) { }

}
0 referências
internal class FileLogger : ILogger
{
    3 referências
    public void Log(string message) { }
}
1 referência
public class Logger
{
    private ILogger _consoleLogger;
    private ILogger _fileLogger;
    0 referências
    public Logger(ILogger consoleLogger, ILogger fileLogger)
    {
        _consoleLogger = consoleLogger;
        _fileLogger = fileLogger;
    }
    0 referências
    public void LogToConsole(string mensagem)
    {
        _consoleLogger.Log(mensagem);
    }
    0 referências
    public void LogToFile(string mensagem)
    {
        _fileLogger.Log(mensagem);
    }
}
```

Ao injetar a interface ILogger no construtor da classe Logger, removemos o acoplamento com os loggers concretos (ConsoleLogger e FileLogger). Agora, a classe Logger depende de abstrações (interfaces), tornando o sistema mais flexível e independente de implementações específicas.

5.1 Benefícios do DIP:

- Flexibilidade: Dependendo de abstrações, o sistema se torna mais flexível e adaptável a mudanças.
- Testabilidade: DIP facilita a substituição de dependências durante testes, permitindo testes unitários e implementação de mocks de forma mais simples.
- Escalabilidade: O sistema se torna mais escalável, já que novas implementações podem ser adicionadas sem modificar o código existente.
- Redução de acoplamento: Reduzir acoplamentos minimiza o risco de alterações em cascata e a introdução de bugs ao modificar o código.

Em conclusão, adotar o Princípio da Inversão de Dependência promove a redução do acoplamento entre componentes, fazendo com que dependam de abstrações em vez de implementações concretas. Isso resulta em sistemas mais fáceis de manter, testar e estender, facilitando a adaptação a novos requisitos e melhorando a robustez do código base.