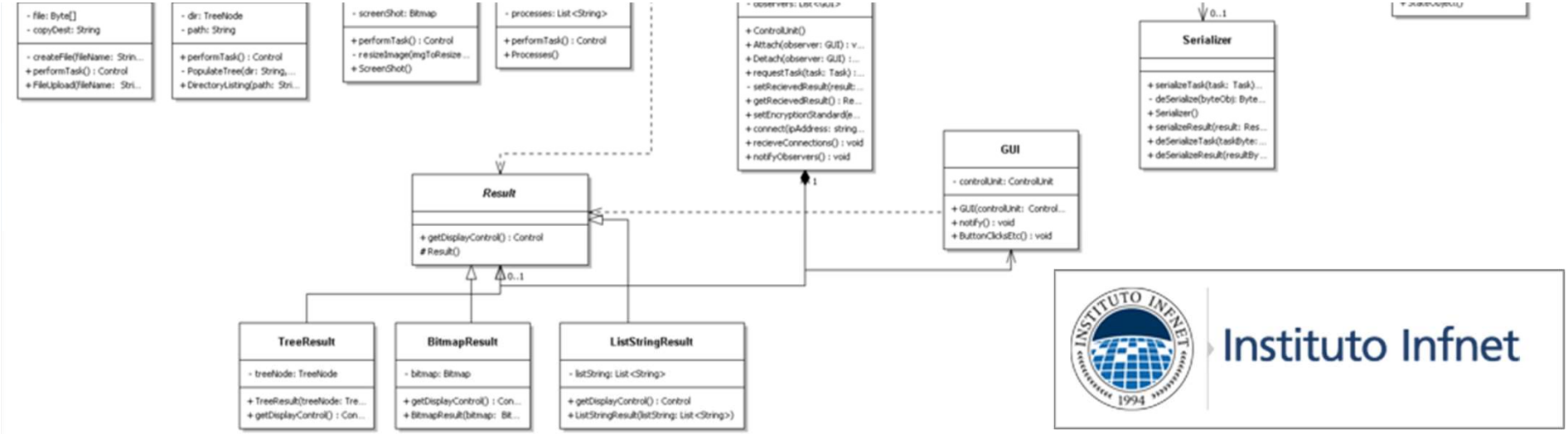


Orientação a Objetos com Java

S.O.L.I.D.

Gustavo de Miranda Gonçalves
gustavo.miranda@prof.infnet.edu.br

Orientação a Objetos (S.O.L.I.D.)

S.O.L.I.D., na verdade, é a união da primeira letra de 5 princípios importantes, identificados por Robert C. Martin e descritos em seu livro Princípios, Padrões e Práticas Ágeis em C#.

- **S**ingle Responsability Principle
- **O**pen-Close Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsability Principle

(Princípio da Responsabilidade Única)

"There should never be more than one reason for a class to change"

Single Responsability Principle

(Princípio da Responsabilidade Única)

Só deve existir um requisito que quando alterado, fará com com que uma classe mude...

Single Responsability Principle

(Princípio da Responsabilidade Única)

"Nomes são classes, verbos são suas funções e adjetivos são suas propriedades"

Single Responsability Principle

(Princípio da Responsabilidade Única)

"Nomes são classes, **verbos são suas funções** e adjetivos são suas propriedades"

Na verdade, alguns verbos deveriam ser outras classes.

Single Responsability Principle

(Princípio da Responsabilidade Única)

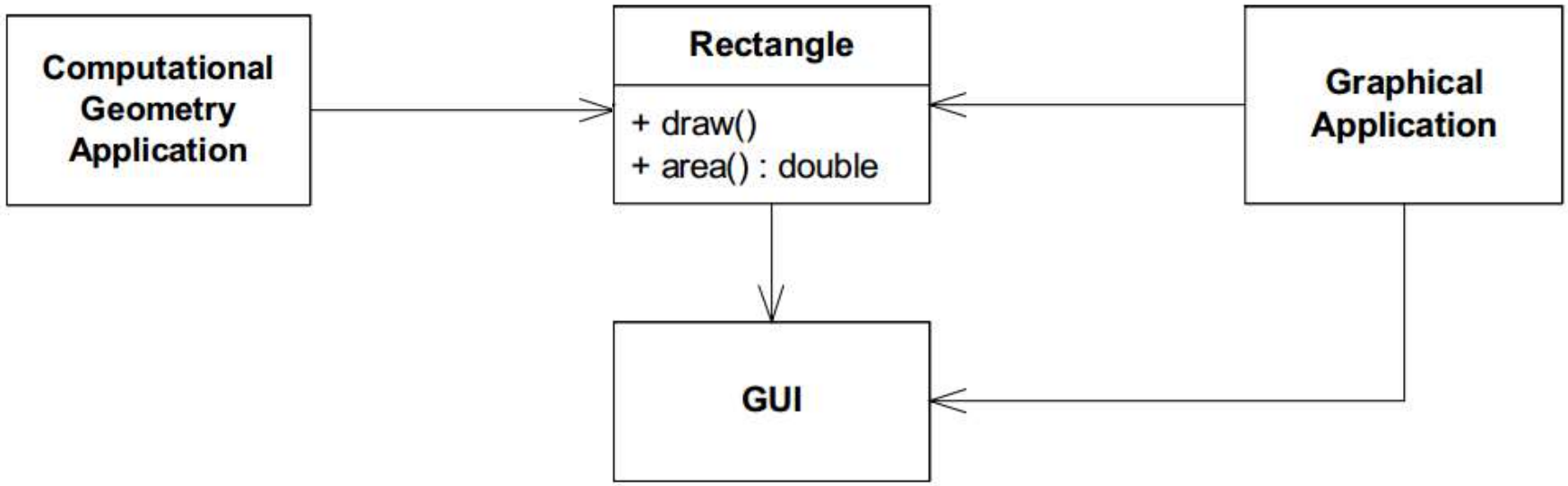
- Separação de Conceitos

  Exemplo: Retângulo (Matematico vs Grafico)

- Única Responsabilidade por Classe
- Evita Classes Monolíticas
- Maior Legibilidade do Código
- Facilidade de Manutenção

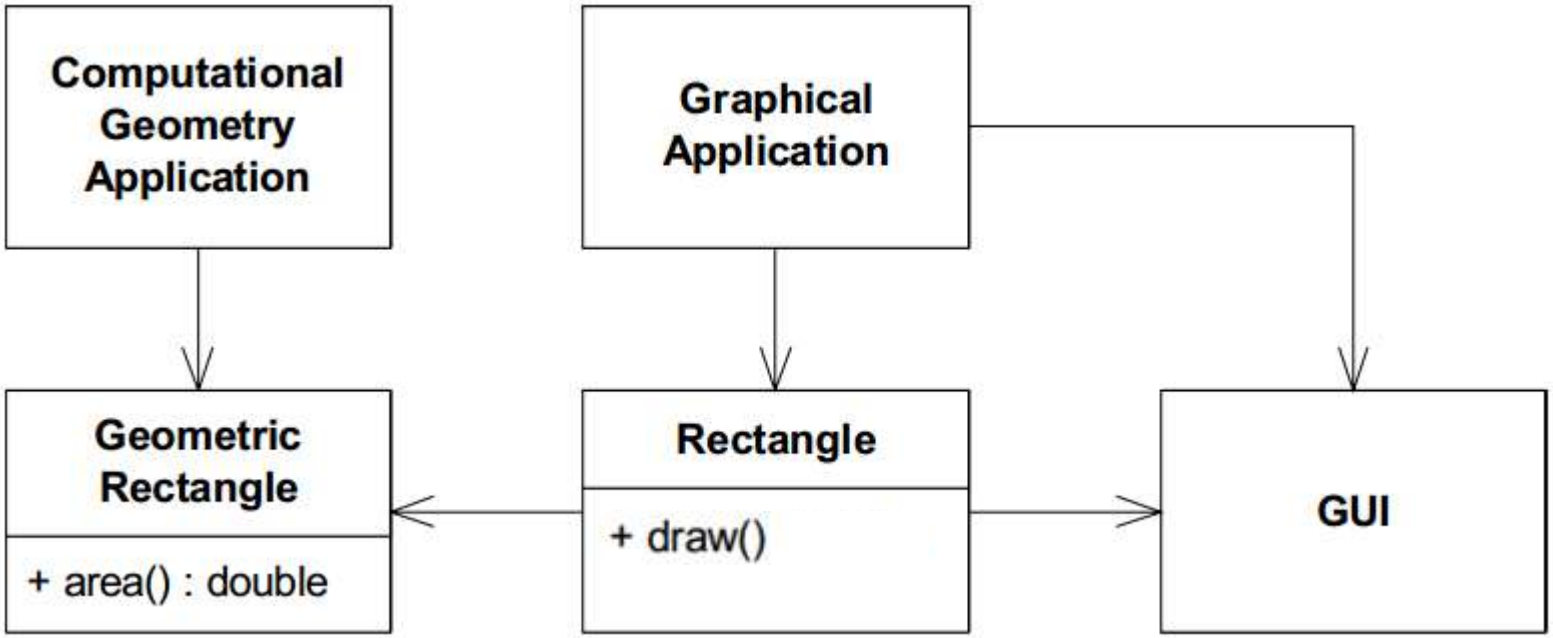
Single Responsability Principle

(Princípio da Responsabilidade Única)

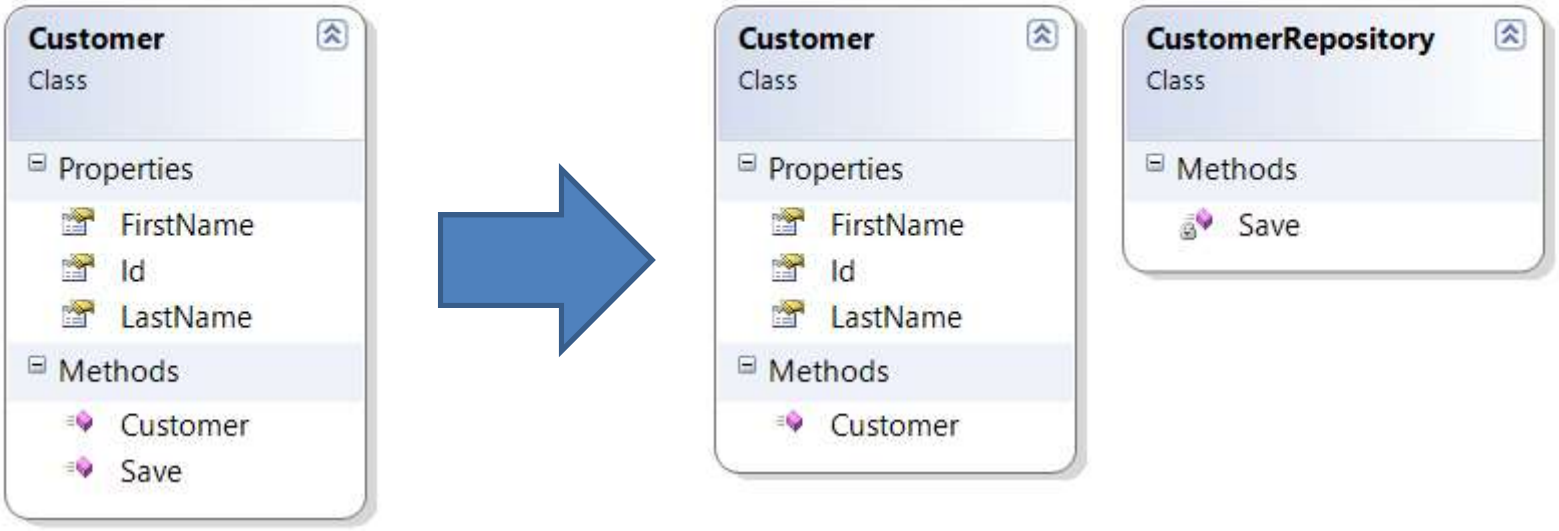


Single Responsability Principle

(Princípio da Responsabilidade Única)



Single Responsability Principle



Open-Close Principle

(Princípio do Aberto-Fechado)

"Software entities (classes, modules, functions, etc.), should be **open for extension**, but **closed for modification**"

Open-Close Principle

(Princípio do Aberto-Fechado)

- Classes devem estar abertas para extensão e fechadas para modificação.

  Adicionar novas funcionalidades e estender uma classe sem que seja necessário alterar seu comportamento interno.

Open-Close Principle

(Princípio do Aberto-Fechado)

```
public class Forma
        {
            int m_tipo;
        }
```

```
public class Quadrado : Forma
        {
            Quadrado()
            {
            }
        }
```

```
public class Circulo : Forma
        {
            Circulo()
            {
            }
        }
```

Open-Close Principle
(Princípio do Aberto-Fechado)

```
public class EditorGrafico
    {
        public void DesenharFormas(List<Forma> listaFormas)
        {
            for (int i = 0; i < listaFormas.Count; i++)
            {
                Forma formaAtual = listaFormas[i];
                if (formaAtual is Quadrado)
                    DesenharQuadrado();
                else if (formaAtual is Circulo)
                    DesenharCirculo();
            }
        }
        public void DesenharCirculo()
        {
            Console.WriteLine("Desenhar circulo");
        }

        public void DesenharQuadrado()
        {
            Console.WriteLine("Desenhar quadrado");
        }
    }
```

Open-Close Principle

(Princípio do Aberto-Fechado)

```
public abstract class Forma
    {
        public abstract void desenhar();
    }
```

```
public class Quadrado : Forma
    {
        public override void desenhar()
        { }
    }
```

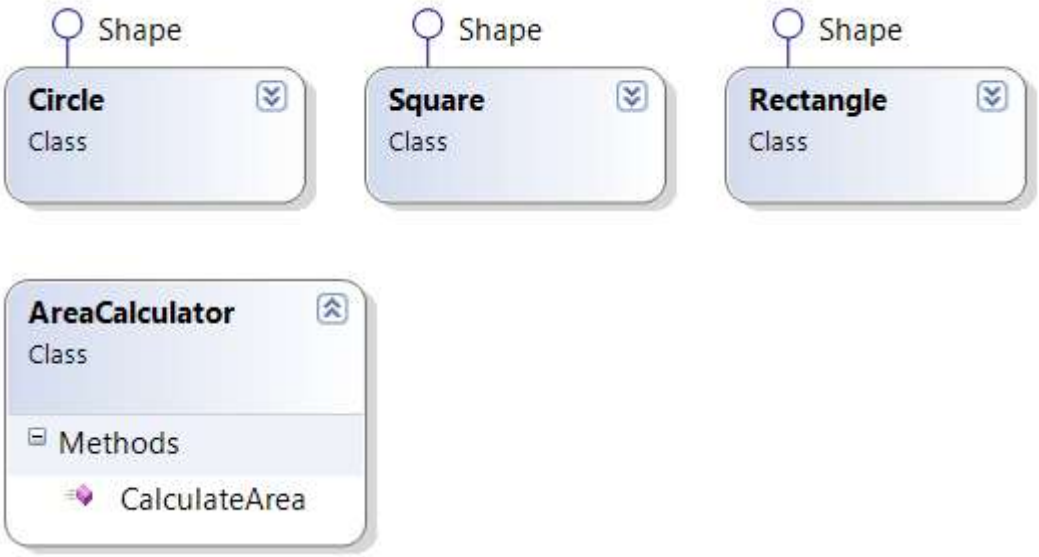
```
public class Circulo : Forma
    {
        public override void desenhar()
        { }
    }
```

Open-Close Principle
(Princípio do Aberto-Fechado)

```
public class EditorGraficoOC
    {
        public void DesenharForma(Forma f)
        {
            f.desenhar();
        }
    }
```

Open-Close Principle

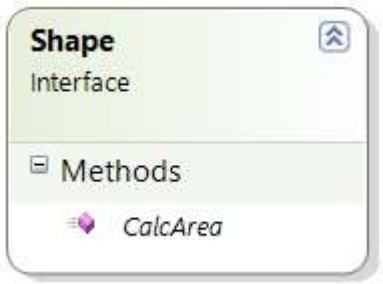
(Princípio do Aberto-Fechado)




```
public int CalculateArea(List<Shape> shapes)
{
    int totalArea = 0;

    foreach (var shape in shapes)
    {
        if (shape is Circle)
        {
            //calculate Circle Area
        }
        else if (shape is Square)
        {
            //calculate Square Area
        }
        else if (shape is Rectangle)
        {
            //calculate Rectangle Area
        }
    }
    return totalArea;
}
```

Open-Close Principle

(Princípio do Aberto-Fechado)




```
public class AreaCalculator
{
    public int CalculateArea(List<Shape> shapes)
    {
        int totalArea = 0;

        foreach (var shape in shapes)
        {
            totalArea += shape.CalcArea();
        }
        return totalArea;
    }
}
```

Liskov Substitution Principle

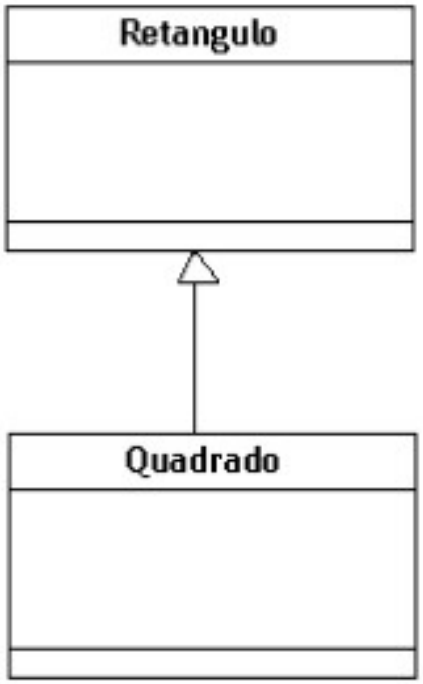
"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program"

Liskov Substitution Principle

- A classe derivada deve sempre poder substituir a classe base.

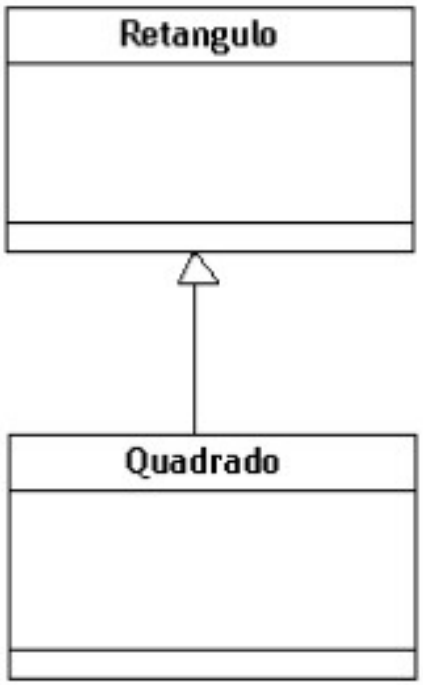
  A aplicação deve continuar funcionando sem que seja necessário realizar mais nenhuma alteração.

Liskov Substitution Principle




```
class Retangulo
    {
        protected int m_largura;
        protected int m_altura;

        public virtual void setLargura(int largura)
        { m_largura = largura; }

        public virtual void setAltura(int altura)
        { m_altura = altura; }

        public int getLargura()
        { return m_largura; }

        public int getAltura()
        { return m_altura; }

        public int getArea()
        { return m_largura * m_altura; }
    }
```

Liskov Substitution Principle




```
class Quadrado : Retangulo
    {
        public override void setLargura(int largura)
        {
            m_largura = largura;
            m_altura = largura;
        }

        public override void setAltura(int altura)
        {
            m_largura = altura;
            m_altura = altura;
        }
    }
```

Liskov Substitution Principle

```
private static Retangulo GetNovoRetangulo()
{
    //um factory
    return new Quadrado();
}

static void Main(string[] args)
{
    //vamos criar um novo retangulo
    Retangulo r = GetNovoRetangulo();

    //definindo a largura e altura do retangulo
    r.setLargura(5);
    r.setAltura(10);
    // o usuário sabe que r é um retângulo
    // e assume que ele pode definir largura e altura
    // como para a classe base(Retangulo)

    Console.WriteLine(r.getArea());
    Console.ReadKey();
    // O valor retornado é 100 e não 50 como era esperado
}
```

Interface Segregation Principle

(Princípio da Segregação de Interface)

"Clients should not be forced to depend upon interfaces that they do not use"

Interface Segregation Principle

(Princípio da Segregação de Interface)

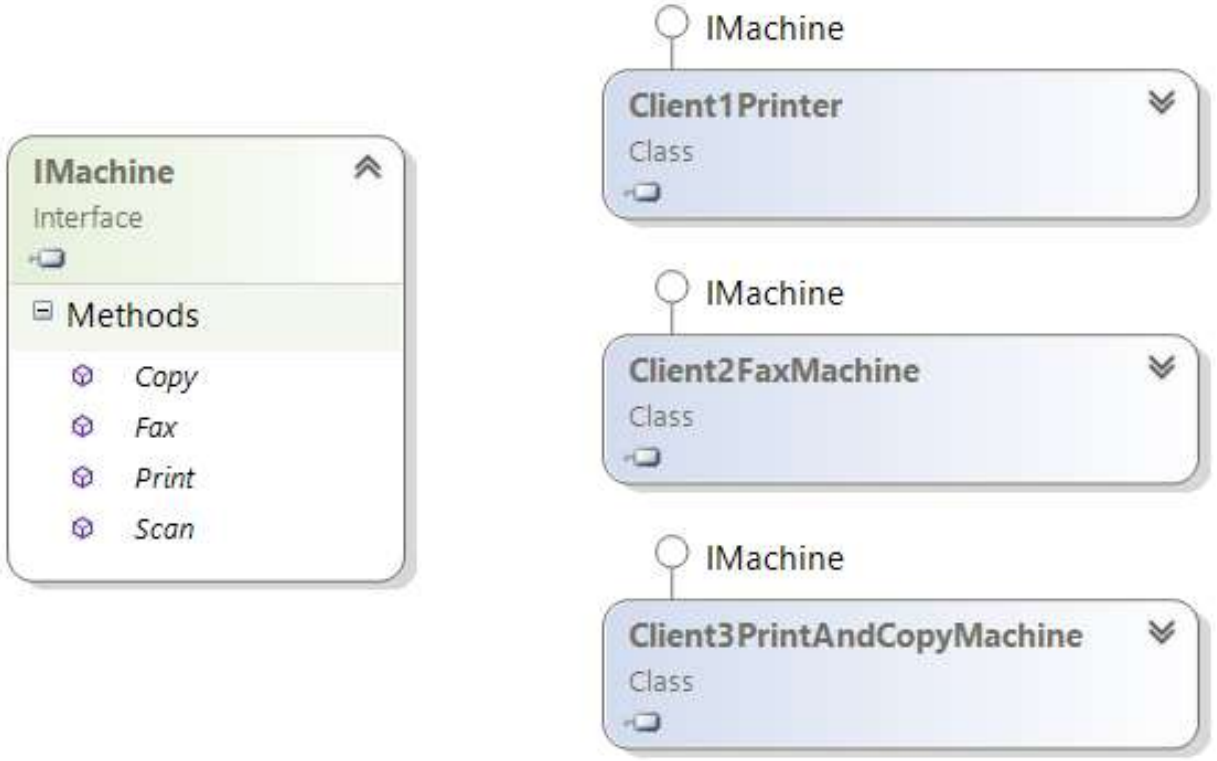
- Sugere que devemos dividir os métodos de uma classe em grupos, de acordo com sua responsabilidade, e estabelecer interfaces para esses grupos.

  Evita que o cliente tenha que implementar uma interface com muitos métodos.

- Ajuda a garantir que uma classe implemente apenas um conjunto específico de métodos.

Interface Segregation Principle

(Princípio da Segregação de Interface)



Dependency Inversion Principle

(Princípio da Inversão de Dependência)

"A. High-level modules should not depend on low-level modules. Both should depend on abstractions."

B. Abstractions should not depend on details. Details should depend on abstractions.

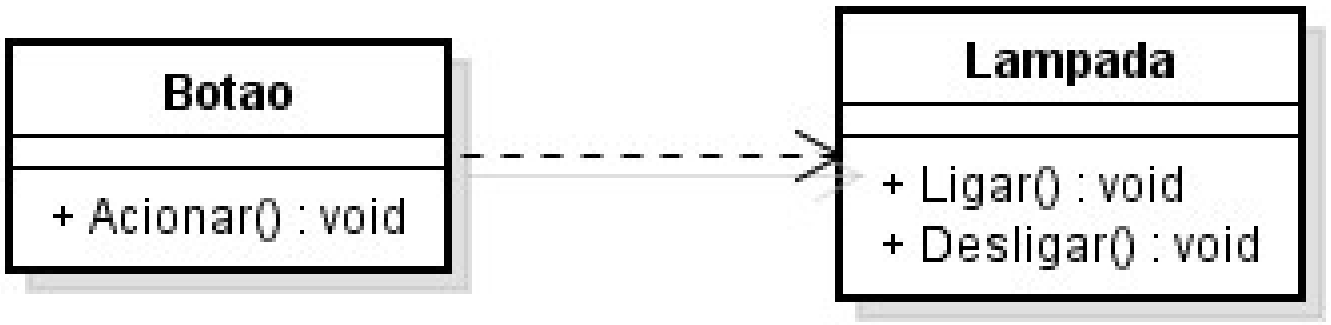
Dependency Inversion Principle

(Princípio da Inversão de Dependência)

- Sugere que devemos isolar as classes de implementações concretas, devemos fazer com que as dependências de uma classe sejam abstratas ou interfaces.

- Esse princípio nos ajuda a garantir que uma classe depende sempre de uma abstração de um conceito e não da implementação dele, já que este tem maiores chances de mudar.

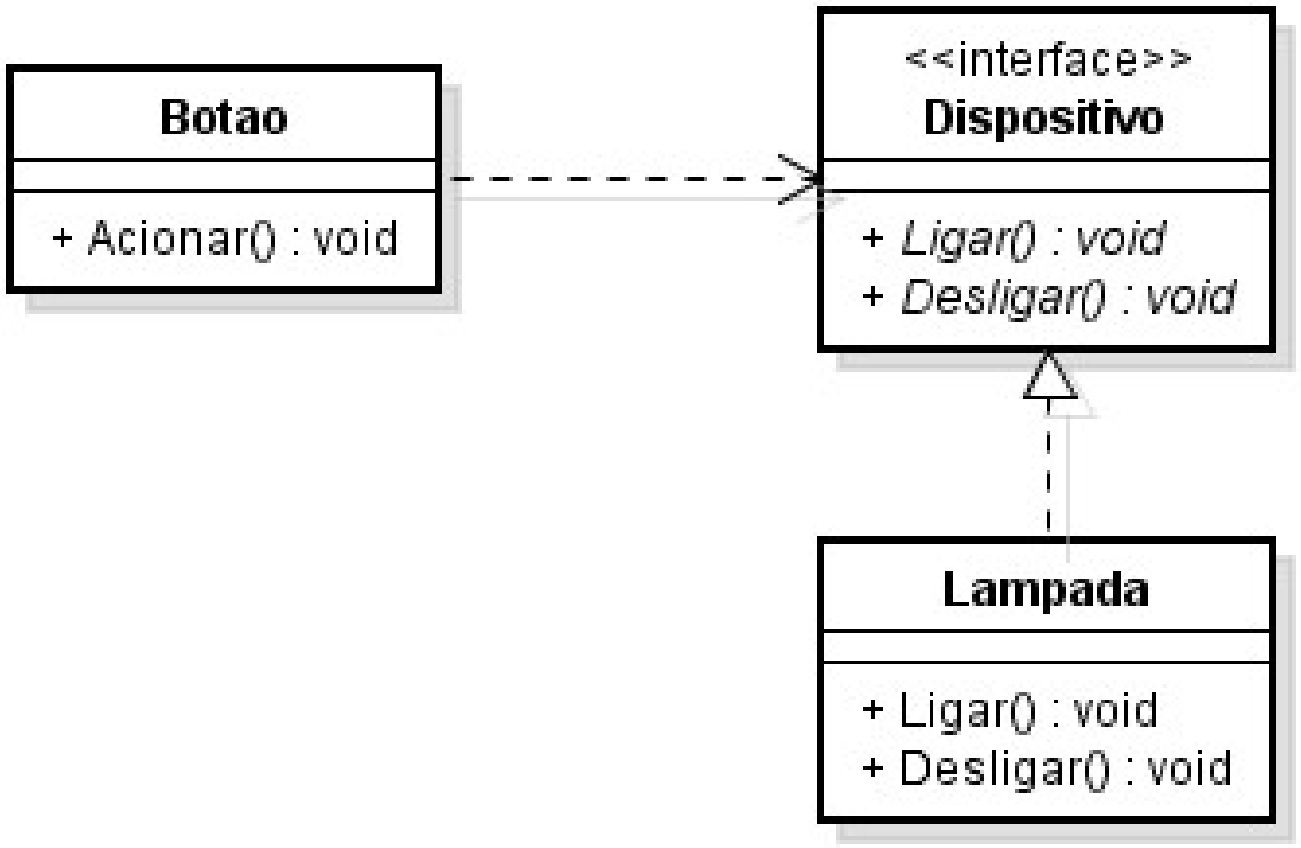
Dependency Inversion Principle



Dependency Inversion Principle

```
public  class  Botao
{
  private  Lampada lampada;


  public  void  Acionar()
  {
    if  (condicao)
      lampada.Ligar();
  }
}
```

Dependency Inversion Principle



Dependency Inversion Principle

```
public class Botao
{
  private IDispositivo dispositivo;

  public void Acionar()
  {
    if (condicao)
      dispositivo.Ligar();
  }
}
```

Dependency Inversion Principle

```
public interface IDispositivo
{
  void Ligar();
  void Desligar();
}
```

```
public class Lampada : IDispositivo
{
  public void Ligar()
  {
    // ligar lampada
  }
  public void Desligar()
  {
    // desligar lampada
  }
}
```