INE5404
Técnicas de Desenvolvimento de Software

Prof. Mateus Grellert

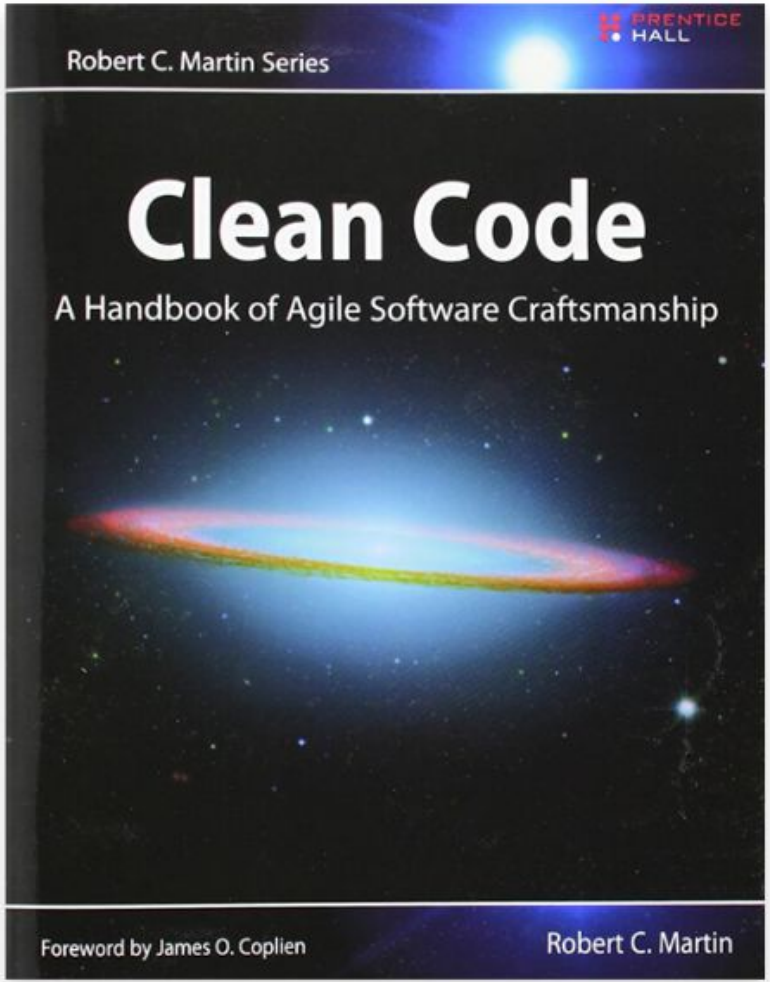
UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Fonte

Technical debt: o débito técnico ou débito de código é o custo implícito causado por escolhas fáceis (e limitadas) ao invés de uma solução melhorada que levaria mais tempo

Fonte

Escreva Código Limpo


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin


circle
Polar circle

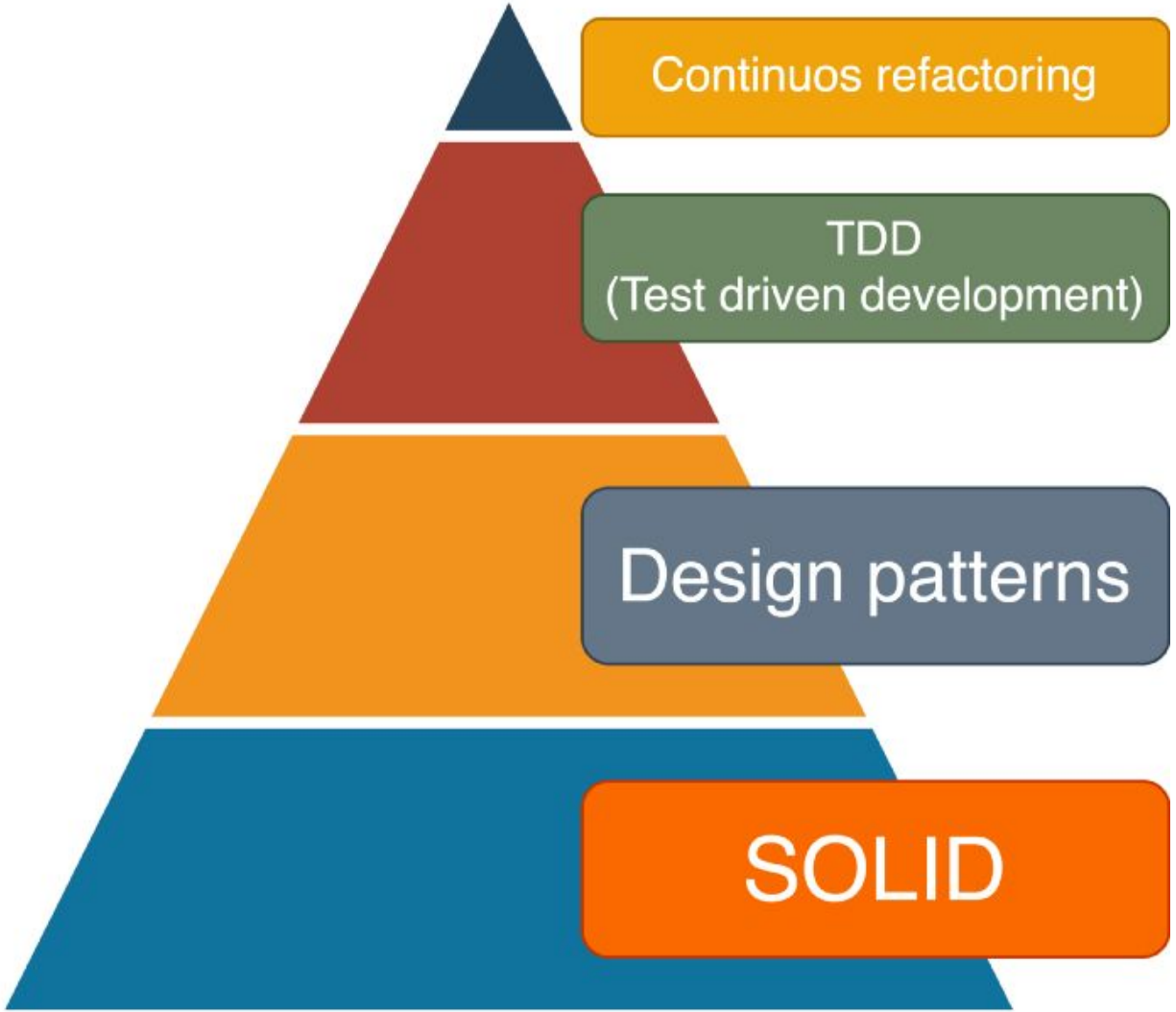
[Fonte]

Desenvolvendo Sistemas Complexos

Já sabemos:

- ➔ modelar sistemas utilizando o paradigma OO
- ➔ adicionar uma interface gráfica sistemas
- ➔ serializar os dados para reuso/compartilhamento
- ➔ Desenvolver sistemas seguindo uma arquitetura padrão com MVC

Vamos ver agora algumas técnicas de desenvolvimento de **código limpo e reusável**


Continuos refactoring
TDD
(Test driven development)
Design patterns
SOLID

Fonte

Parte 1:
S.O.L.I.D.

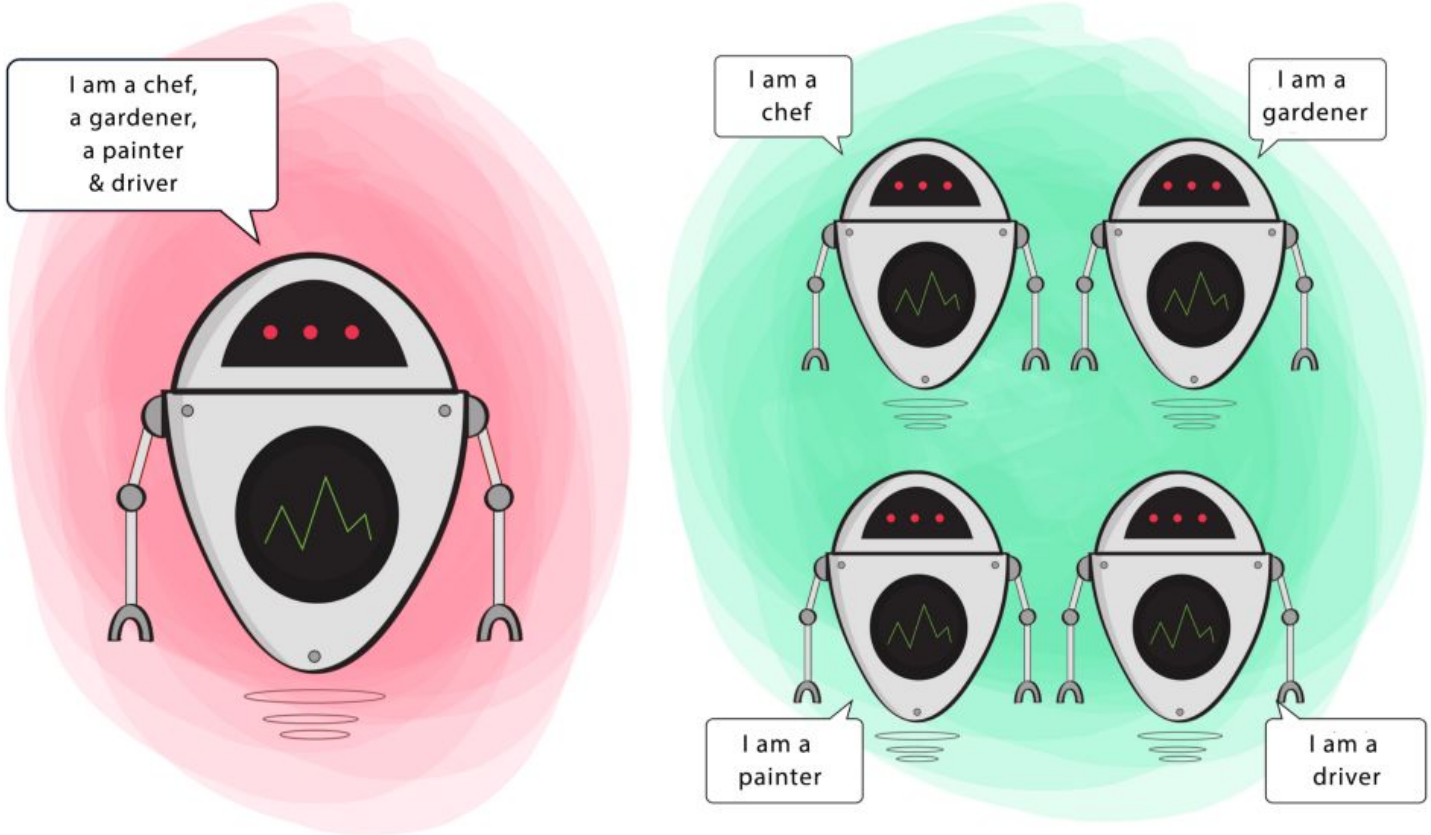
SOLID

- **S**ingle Responsibility
  - Uma classe deve ter **uma responsabilidade,** isto é, somente uma das mudanças nas especificações exigem mudanças na implementação
- **O**pen-closed
  - Classes devem estar abertas para **extensão**, mas fechadas para **modificação**
- **L**iskov Substitution
  - **Classes base devem ser substituíveis** de suas classes derivadas sem alterar a corretude do programa
- **I**nterface segregation
  - **Interfaces específicas** do cliente são melhores que uma interface genérica
- **D**ependency Inversion
  - Devemos ter **dependências em abstrações**, não em implementações concretas

Single Responsibility Principle

Single Responsibility Principle (SRP)

- Cada módulo, classe ou função deve ter responsabilidade sobre **uma única funcionalidade** do sistema
- *“Uma classe deve ter somente um motivo para mudar”*
  - **Uma única** especificação do projeto deve mudar a implementação de uma dada classe
- Princípio baseado na **coesão**

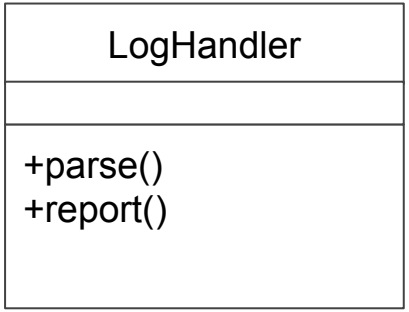


Single Responsibility

Fonte

Single Responsibility Principle (SRP)

Exemplo: um módulo do programa é responsável por compilar (parse) um relatório e imprimir o resultado na tela

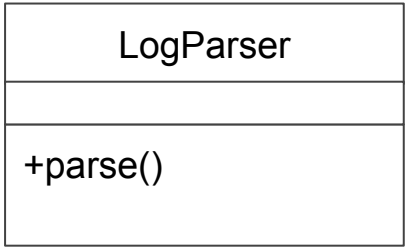


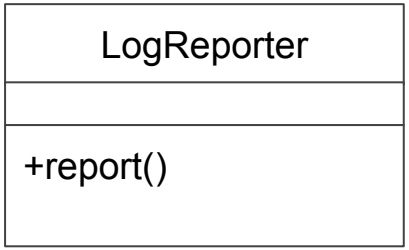
- Se mudarmos o **conteúdo** do relatório, vamos precisar mudar a implementação
- Se mudarmos a **forma** como deve ser impresso precisamos mudar a implementação

Single Responsibility Principle (SRP)

Exemplo: um módulo do programa é responsável por compilar (parse) um relatório e imprimir o resultado na tela

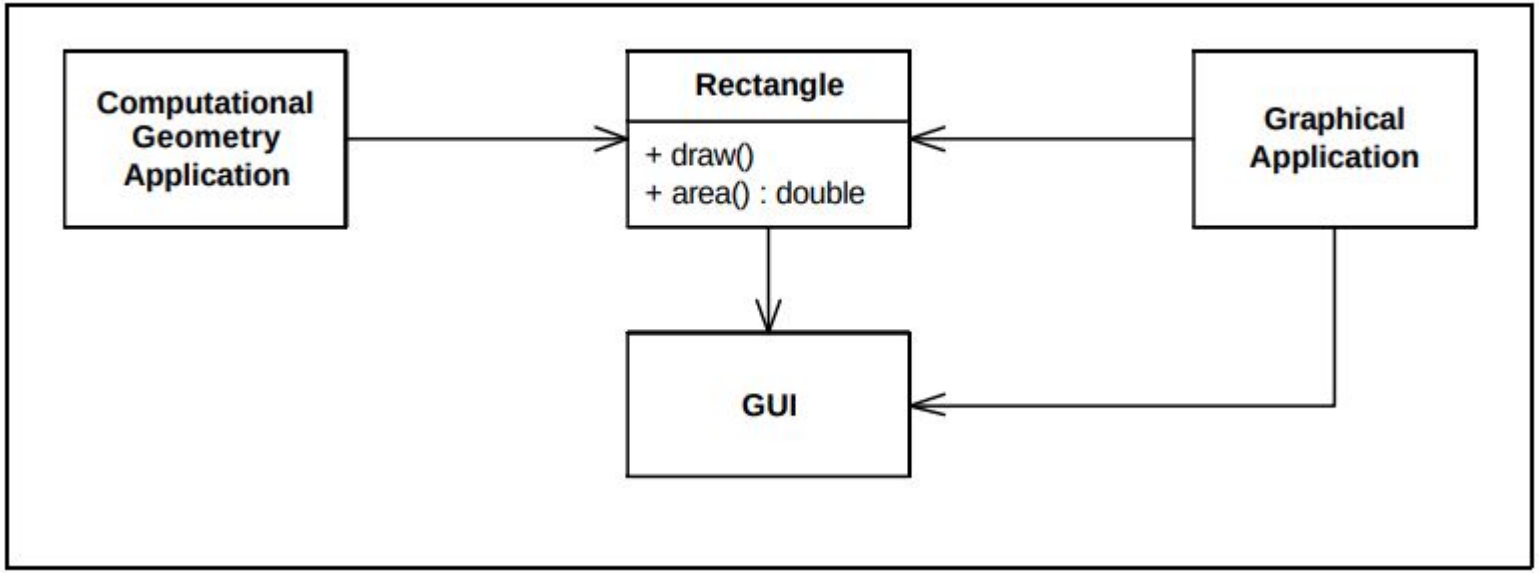
- Para resolver isso, podemos **dividir essa tarefa em duas classes**, de forma que fica mais fácil manter e modificar quando necessário





Single Responsibility Principle (SRP)

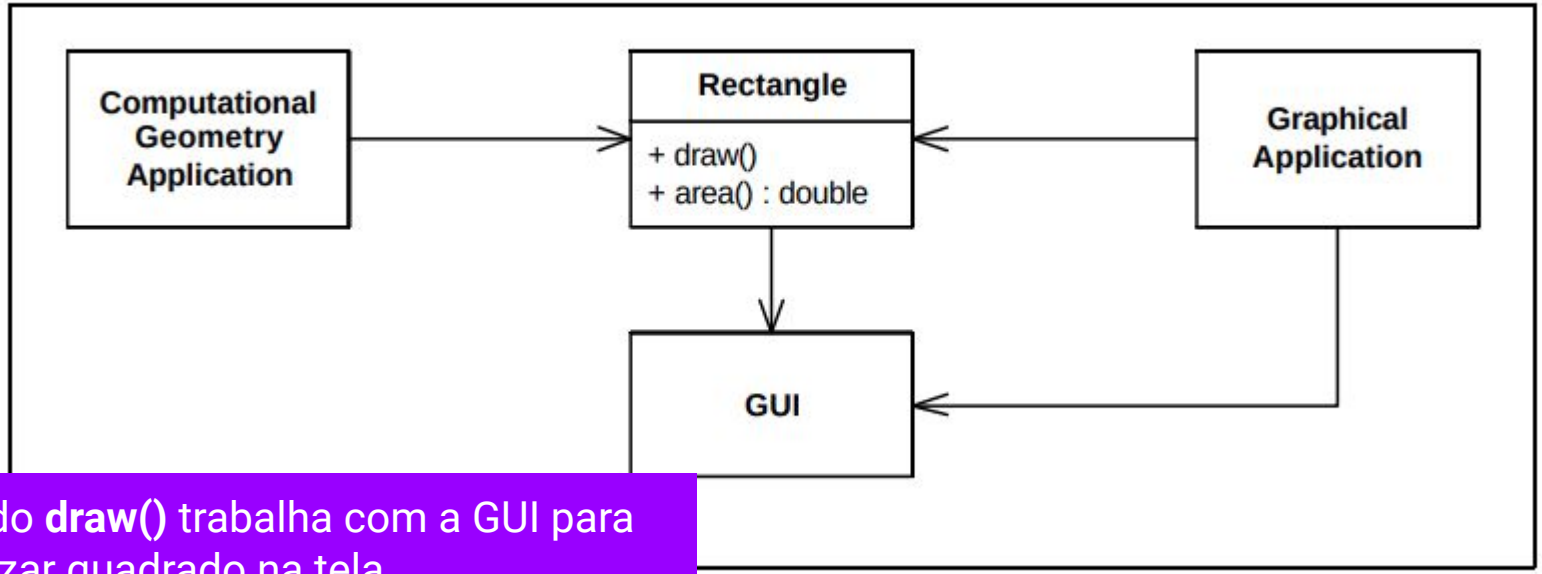
Mais um exemplo:



Single Responsibility Principle (SRP)

Mais um exemplo:

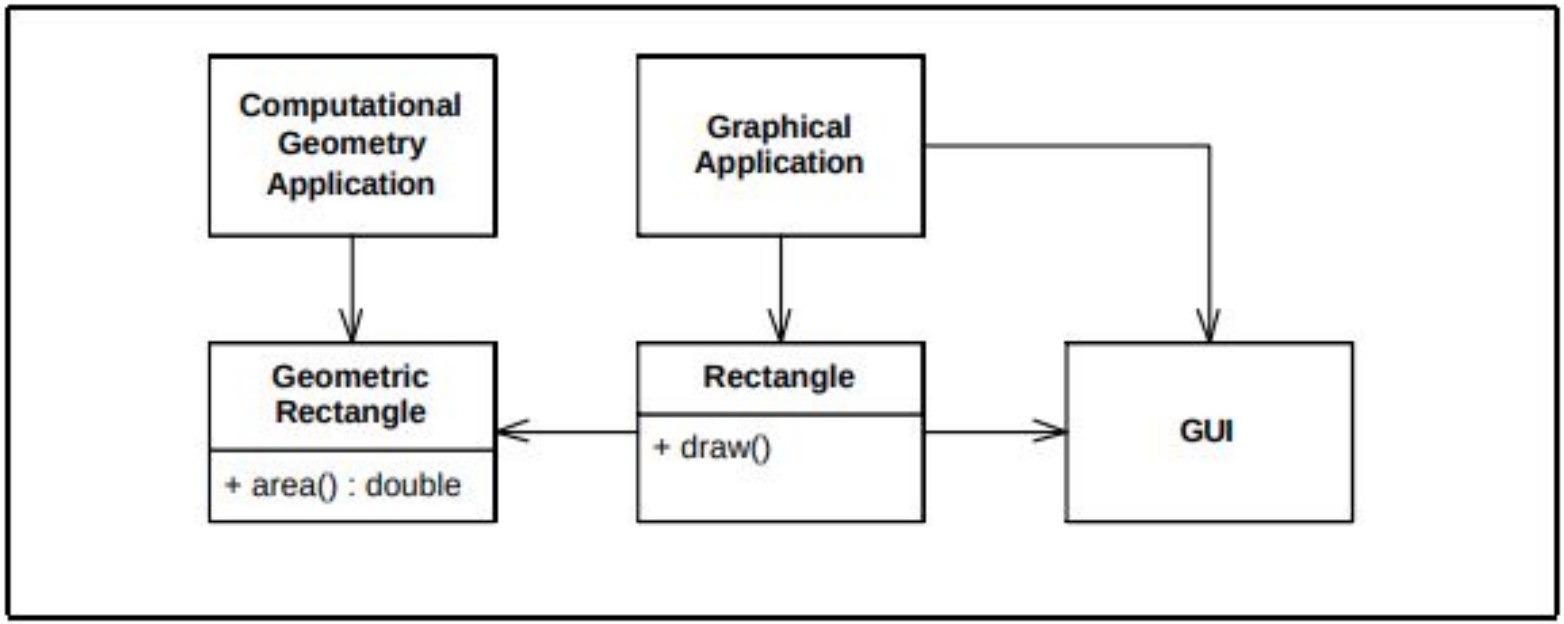
Violação de SRP



- Método **draw()** trabalha com a GUI para renderizar quadrado na tela
- Método **area()** trabalha com a Geometry App. para calcular dados geométricos

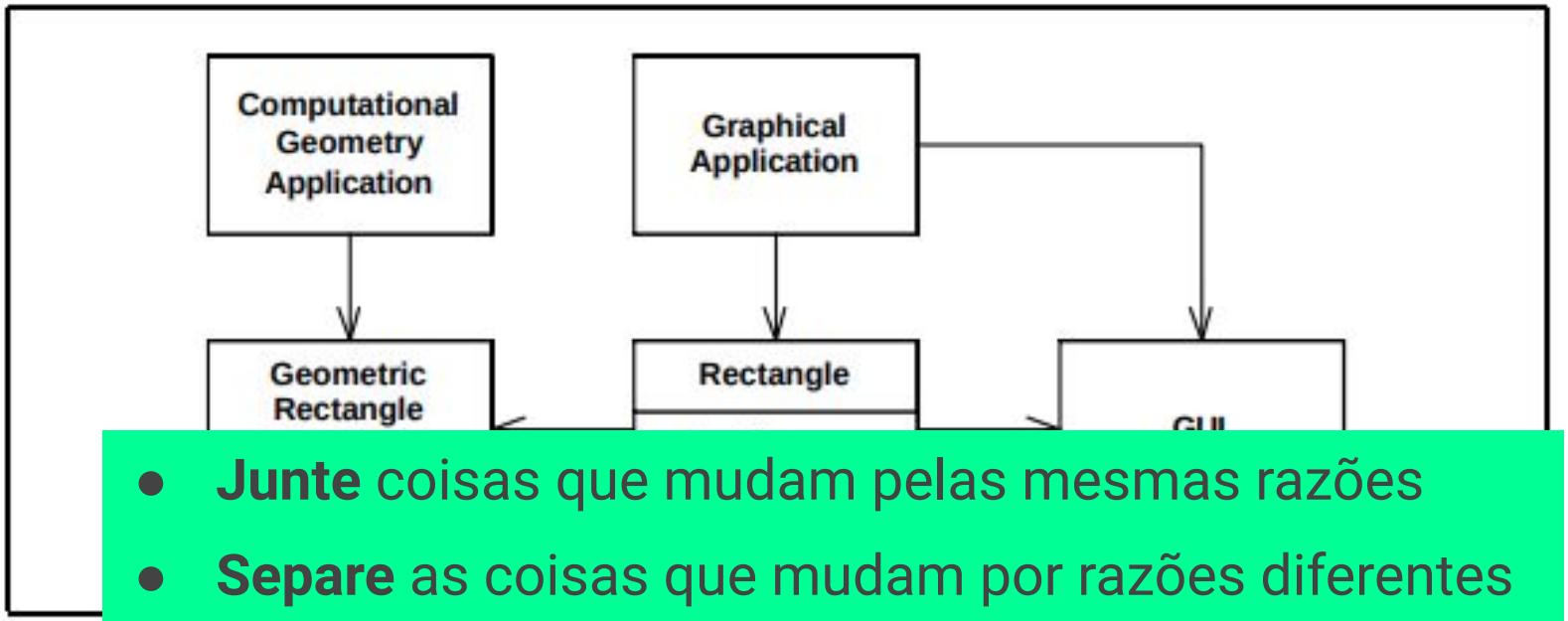
Single Responsibility Principle (SRP)

Separando as responsabilidades:



Single Responsibility Principle (SRP)

Separando as responsabilidades:



- **Junte** coisas que mudam pelas mesmas razões
- **Separe** as coisas que mudam por razões diferentes

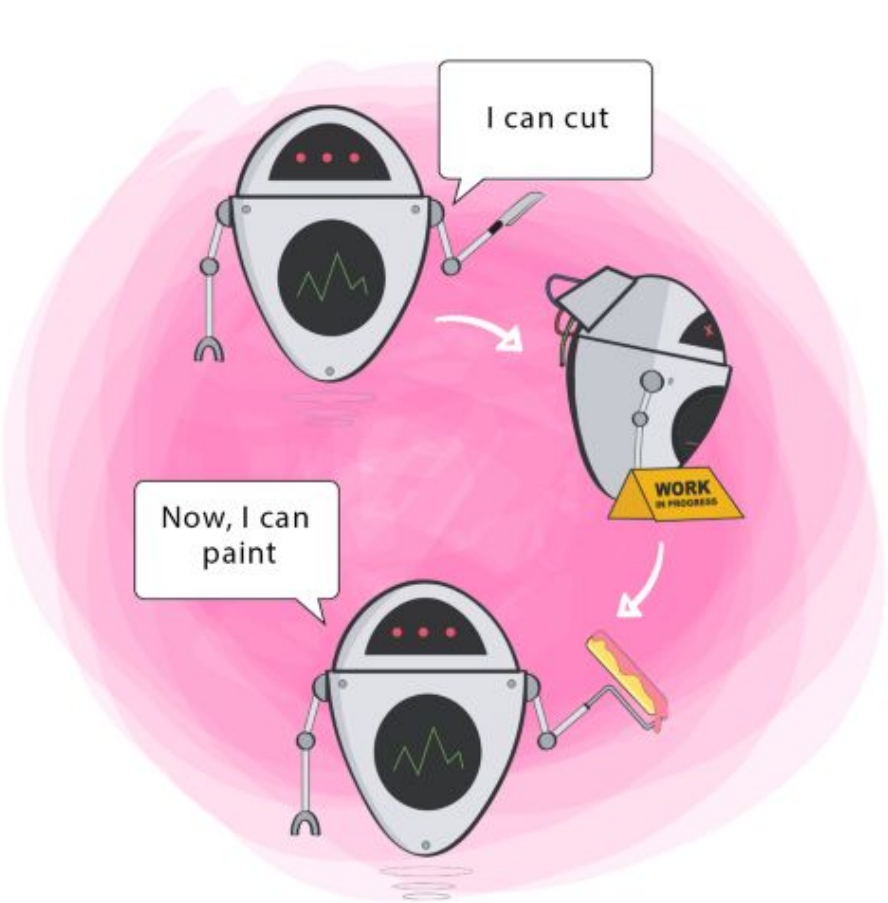
Open-closed Principle

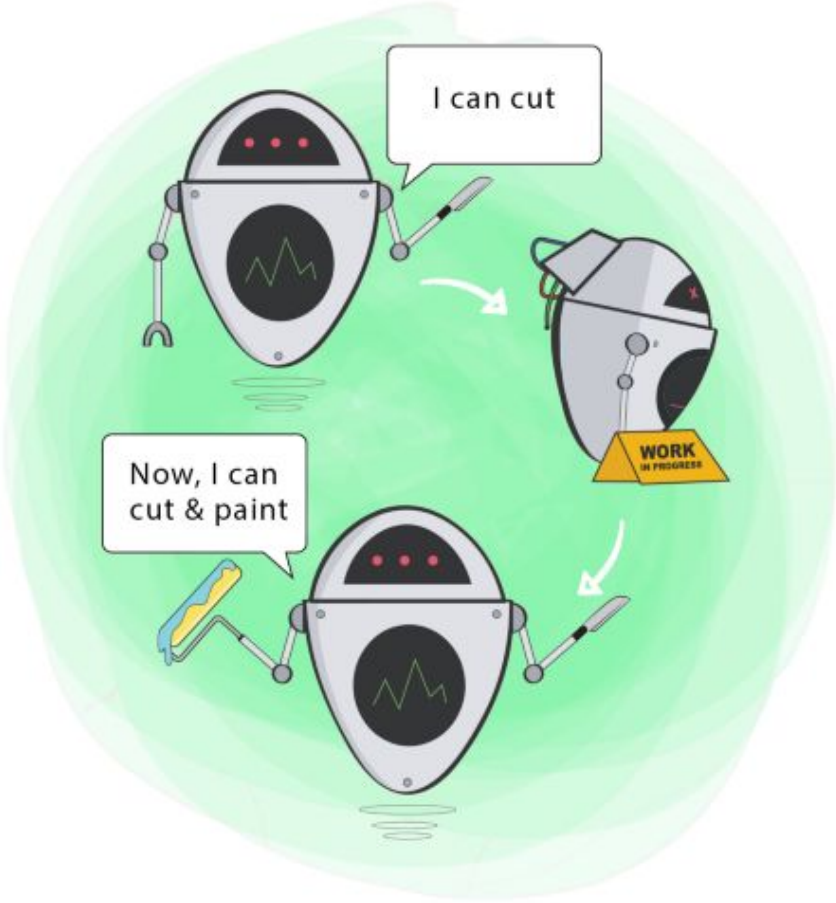
Open-closed Principle (OCP)

Devemos **estender** as funcionalidades de uma classe **sem** que para isso precisemos **modificar** sua implementação

- Um módulo estará aberto se ainda está disponível para extensão.
- Um módulo estará fechado se estiver disponível para uso por outros módulos. Assume-se que o módulo tenha uma interface bem definida e estável

Classes estão **fechadas** após compilação e portanto podem ser facilmente reusadas, mas continuam **abertas**, já que outras classes podem utilizá-la como classe pai





Open-Closed

Open-closed Principle (OCP)

Nossa tarefa é criar uma classe para calcular a área de um retângulo

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

Open-closed Principle (OCP)

Nossa tarefa é criar uma classe para calcular a área de um retângulo

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

Criamos o método que computa a área de um retângulo de forma adequada

… mas o que acontece se o cliente resolve pedir para **estender** a classe para calcular a área de um círculo?

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Mat
h.PI;
        }
    }

    return area;
}
```

Podemos modificar o método Area() para considerar o tipo do objeto enviado como parâmetro

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Mat
h.PI;
        }
    }

    return area;
}
```

Violação de OCP

Podemos modificar o método Area() para considerar o tipo do objeto enviado como parâmetro

… mas estamos **modificando a implementação** do calculador de área.

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Mat
h.PI;
        }
    }

    return area;
}
```

Violação de OCP

Podemos modificar o método Area() para considerar o tipo do objeto enviado como parâmetro

… mas estamos **modificando a implementação** do

O que acontece se queremos novamente estender a funcionalidade para calcular a área de círculos?

Abstração é a Chave

```
public abstract class Shape
{
    public abstract double Area();
}
```

1

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

2

2

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

3

1 - Criamos uma classe **abstrata** Forma

2 - Implementamos o cálculo de área nas classes **especializadas**

3 - Implementamos o cálculo da área de forma **extensível** através de **polimorfismo**

Até a próxima!

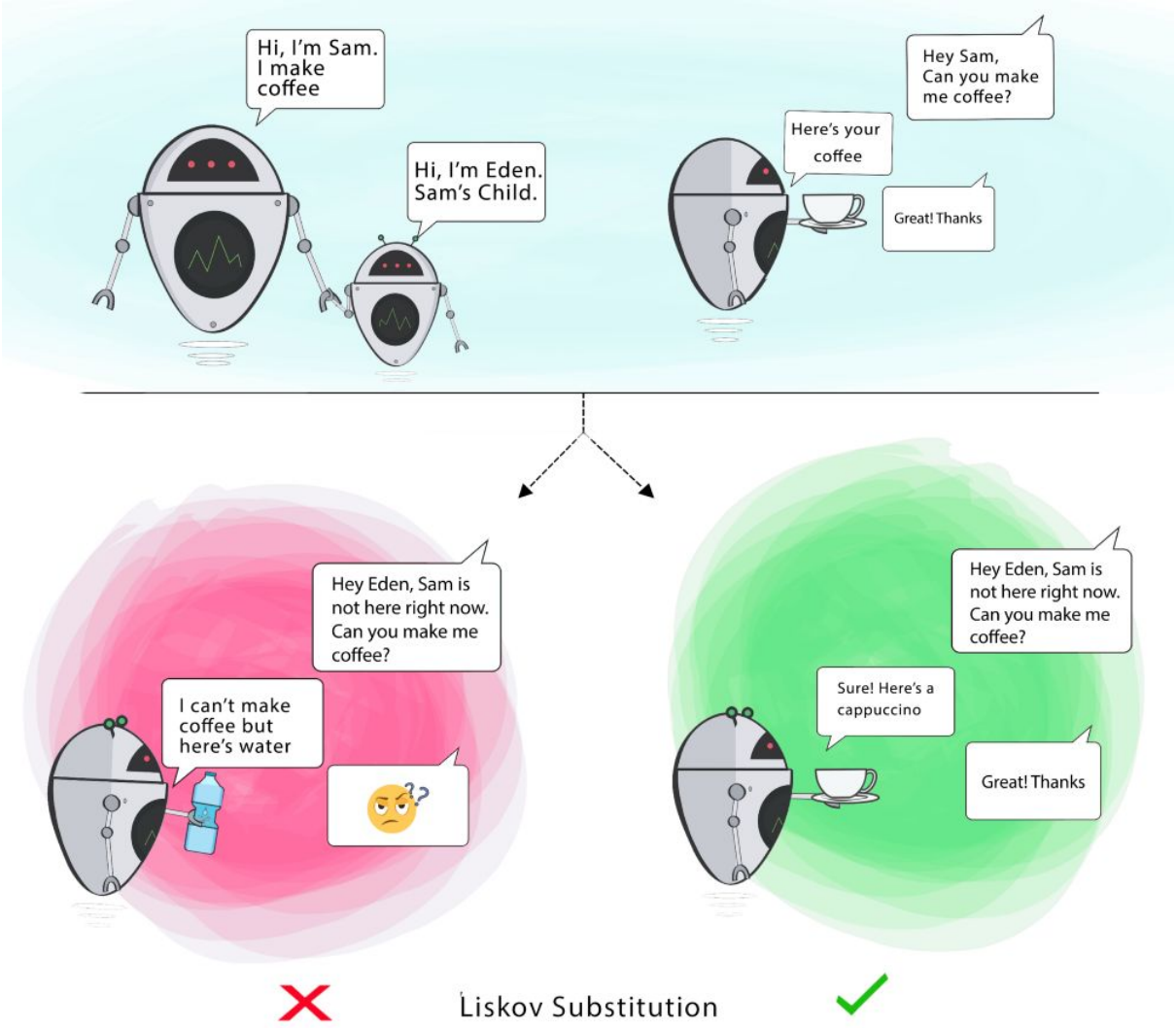
Parte 2:
(mais) S.O.L.I.D.

Liskov Substitution Principle

Liskov Substitution Principle (LSP)

Funções que usam ponteiros ou referências para uma **classe base** devem poder usar objetos de **classes derivadas sem saber disso**

“Se S é um subtipo de T, então objetos do tipo T devem poder ser substituídos por objetos do tipo S, sem que as propriedades desejáveis do programa sejam alteradas”


Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks
Liskov Substitution

[Fonte]

Liskov Substitution Principle (LSP)

```java
public class Animal {
    public void makeNoise() {
        System.out.println("I am making noise");
    }
}
```

Classe base

```java
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("bow wow");
    }
}

public class Cat extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("meow meow");
    }
}
```

Classes derivadas

Fonte

Liskov Substitution Principle (LSP)

```java
public class Animal {
    public void makeNoise() {
        System.out.println("I am making noise");
    }
}
```

Classe base

```java
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("bow wow");
    }
}

public class Cat extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("meow meow");
    }
}
```

Classes derivadas

Podemos usar tanto o método makeNoise() da classe base quando das derivadas sem gerar nenhum erro

Fonte

Liskov Substitution Principle (LSP)

```java
public class Animal {
    public void makeNoise() {
        System.out.println("I am making noise");
    }
}
```

Classe base

```java
class DumbDog extends Animal {
    @Override
    public void makeNoise() {
        throw new RuntimeException("I can't make noise");
    }
}
```

Classe derivada

Fonte

Liskov Substitution Principle (LSP)

```java
public class Animal {
    public void makeNoise() {
        System.out.println("I am making noise");
    }
}
```

Classe base

```java
class DumbDog extends Animal {
    @Override
    public void makeNoise() {
        throw new RuntimeException("I can't make noise");
    }
}
```

Classe derivada

Violação de LSP

Se chamarmos makeNoise com um objeto da classe derivada criaremos uma exceção!

Fonte

LSB é habilitador para OCP

Interface Segregation Principle

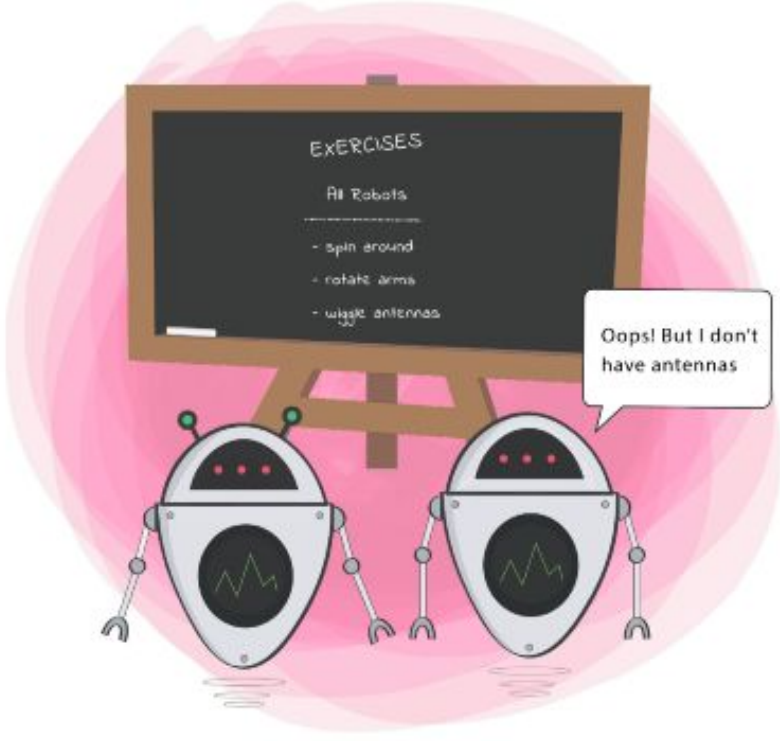
Interface Segregation Principle (ISP)

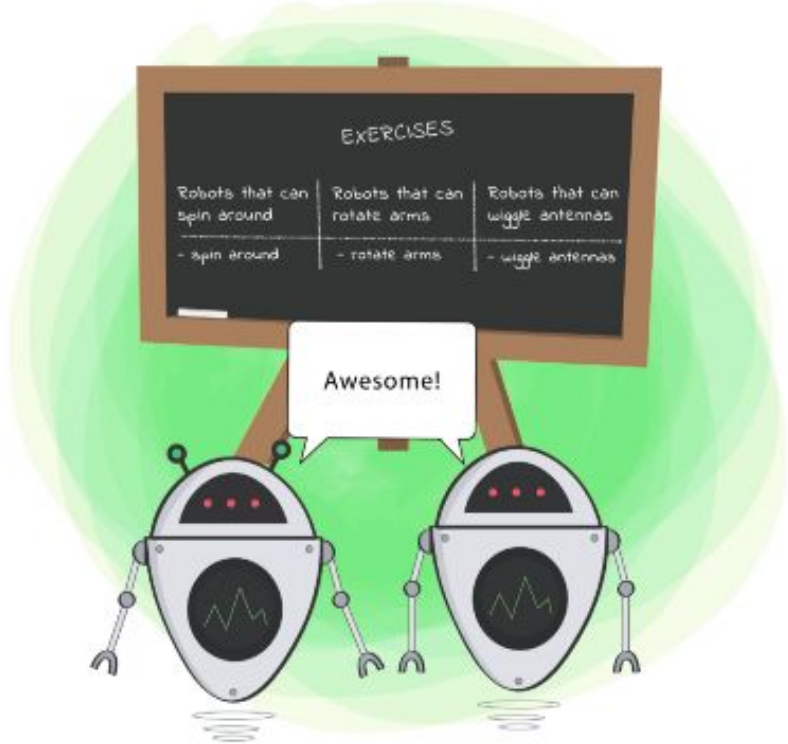
Nenhum cliente deve ser forçado a depender de **métodos que ele não usa**

Decomposição de interfaces complexas em menores (role interfaces)

Normalmente, interfaces complexas são sinônimos de **baixa coesão**

Reduz a criação de **dependências desnecessárias**





Interface Segregation

Interface Segregation Principle (ISP)

```
1   public abstract class Employee {
2
3           private String name;
4
5           public String getName() {
6               return name;
7           }
8
9           public void setName(String name) {
10              this.name = name;
11          }
12
13          public abstract double getSalary();
14
15          public abstract double getCommission();
16  }
```

Interface Segregation Principle (ISP)

```java
public class Seller extends Employee{

	private double salary;
	private int totalSales;

	public Seller(double salary, int totalSales) {
		this.salary = salary;
		this.totalSales = totalSales;
	}

	@Override
	public double getSalary() {
		return this.salary + this.getCommission();
	}

	@Override
	public double getCommission() {
		return this.totalSales * 0.2;
	}
}
```

```java
public class Developer extends Employee{

	private double salary;

	public Developer(double salary) {
		this.salary = salary;
	}

	@Override
	public double getSalary() {
		return this.salary;
	}

	@Override
	public double getCommission() {
		return 0d;
	}
}
```

Interface Segregation Principle (ISP)

Violação de ISP

```java
public class Seller extends Employee{

      private double salary;
      private int totalSales;

      public Seller(double salary, int totalSales) {
            this.salary = salary;
            this.totalSales = totalSales;
      }

      @Override
      public double getSalary() {
            return this.salary + this.getCommission();
      }

      @Override
      public double getCommission() {
            return this.totalSales * 0.2;
      }
}
```

```java
public class Developer extends Employee{

      private d

      public De
            this
      }

      @Override
      public double getSalary() {
            retu this.salary;
      }

      @Override
      public double getCommission() {
            return 0d;
      }
}
```

A Classe Developer teve que implementar a função dummy **getComission()** só para satisfazer a interface!

Solução

```java
public interface Conventional {
    public double getSalary();
}
```

```java
public interface Commissionable {
    public double getCommission();
}
```

```java
public abstract class Employee implements Conventional{

    private String name;
    private double salary;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public double getSalary() {
        return this.salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

Solução

```java
public interface Conventional {
    public double getSalary();
}
```

```java
public interface Commissionable {
    public double getCommission();
}
```

Decompomos em duas interfaces: convencional e por comissão

```java
public abstract class Employee implements Conventional{

    private String name;
    private double salary;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public double getSalary() {
        return this.salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

Classe abstrata **Employee** implementa a interface **Conventional** (comum a ambos os casos)

```java
package io.github.mariazevedo88.solid.isp;

public class Seller extends Employee implements Commissionable{

	private double salary;
	private int totalSales;

	public Seller(double salary, int totalSales) {
		this.salary = salary;
		this.totalSales = totalSales;
	}

	@Override
	public double getSalary() {
		return this.salary + this.getCommission();
	}

	@Override
	public double getCommission() {
		return this.totalSales * 0.2;
	}

	@Override
	public String toString() {
		return "Seller [salary=" + salary + ", totalSales=" + totalSales + "]";
	}
}
```

Classe **Seller** implementa a **segunda** interface que inclui o método **getComission()**

Dependency Inversion Principle

Dependency Inversion Principle (DIP)

Dependência deve ocorrer entre abstrações, não em implementações (versões concretas)

“Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações”

“Abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações”

```
1   public class Interruptor
2   {
3     private Ventilador _ventilador;
4
5     public void Acionar()
6     {
7       if(!_ventilador.Ligado)
8         _ventilador.Ligar();
9       else
10        _ventilador.Desligar();
11    }
12  }
13
14  public class Ventilador
15  {
16    public bool Ligado {get; set; }
17
18    public void Ligar() { ... }
19
20    public void Desligar() { ... }
21  }
```

Fonte

```
1   public class Interruptor
2   {
3     private Ventilador _ventilador;
4
5     public void Acionar()
6     {
7       if(!_ventilador.Ligado)
8         _ventilador.Ligar();
9       else
10        _ventilador.Desligar();
11    }
12  }
13
14  public class Ventilador
15  {
16    public bool Ligado {get; set; }
17
18    public void Ligar() { ... }
19
20    public void Desligar() { ... }
21  }
```

Fonte

Violação de DIP!

- Interruptor depende de uma classe concreta
- Não faz mais sentido que interruptores consiga acionar vários dispositivos?

Solução

```
interface IDispositivo
{
  bool Ligado { get; set; }
  void Acionar();
  void Ligar();
  void Desligar();
}

public class Ventilador : IDispositivo
{
  public bool Ligado { get; set; }

  public void Acionar ()
  {
    if (!this.Ligado)
      this.Ligar();
    else
      this.Desligar();
  }

  public void Ligar() { ... }

  public void Desligar() { ... }
}
```

```
public class Lampada : IDispositivo
{
  public bool Ligado { get; set; }

  public void Acionar ()
  {
    if (!this.Ligado)
      this.Ligar();
    else
      this.Desligar();
  }

  public void Ligar() { ... }

  public void Desligar() { ... }
}

public class Interruptor
{
  private readonly IDispositivo _dispositivo;

  public void AcionarDispositivo()
  {
    _dispositivo.Acionar();
  }
}
```

Solução

```
1  interface IDispositivo
2  {
3    bool Ligado { get; set; }
4    void Acionar();
5    void Ligar();
6    void Desligar();
7  }
8
9  public class Ventilador : IDispositivo
10 {
11   public bool Ligado { get; set; }
12
13   public void Acionar ()
14   {
15     if (!this.Ligado)
16       this.Ligar();
17     else
18       this.Desligar();
19   }
20
21   public void Ligar() { ... }
22
23   public void Desligar() { ... }
24 }
```

1

```
25
26 public class Lampada : IDispositivo
27 {
28   public bool Ligado { get; set; }
29
30   public void Acionar ()
31   {
32     if (!this.Ligado)
33       this.Ligar();
34     else
35       this.Desligar();
36   }
37
38   public void Ligar() { ...
39
40   public void Desligar() { .
41 }
42
43 public class Interruptor
44 {
45   private readonly IDispositivo _dispositivo;
46
47   public void AcionarDispositivo()
48   {
49     _dispositivo.Acionar();
50   }
51 }
```

1 - Criamos a interface **IDispositivo** para englobar todos os dispositivos

Solução

```
1  interface IDispositivo
2  {
3    bool Ligado { get; set; }
4    void Acionar();
5    void Ligar();
6    void Desligar();
7  }
8
9  public class Ventilador : IDispositivo
10 {
11   public bool Ligado { get; set; }
12
13   public void Acionar ()
14   {
15     if (!this.Ligado)
16       this.Ligar();
17     else
18       this.Desligar();
19   }
20
21   public void Ligar() { ... }
22
23   public void Desligar() { ... }
24 }
```

```
25
26 public class Lampada : IDispositivo
27 {
28   public bool Ligado { get; set; }
29
30   public void Acionar ()
31   {
32     if (!this.Ligado)
33       this.Ligar();
34     else
35       this.Desligar();
36   }
37
38   public void Ligar() { ...
39
40   public void Desligar() {
41 }
42
43 public class Interruptor
44 {
45   private readonly IDispositivo _dispositivo;
46
47   public void AcionarDispositivo()
48   {
49     _dispositivo.Acionar();
50   }
51 }
```

1 - Criamos a interface **IDispositivo** para englobar todos os dispositivos

2 - Especializamos os dispositivos implementando o método **Acionar**() em ambos

Solução

```
1  interface IDispositivo
2  {
3    bool Ligado { get; set; }
4    void Acionar();
5    void Ligar();
6    void Desligar();
7  }
8
9  public class Ventilador : IDispositivo
10 {
11   public bool Ligado { get; set; }
12
13   public void Acionar ()
14   {
15     if (!this.Ligado)
16       this.Ligar();
17     else
18       this.Desligar();
19   }
20
21   public void Ligar() { ... }
22
23   public void Desligar() { ... }
24 }
```

```
25
26 public class Lampada : IDispositivo
27 {
28   public bool Ligado { get; set; }
29
30   public void Acionar ()
31   {
32     if (!this.Ligado)
33       this.Ligar();
34     else
35       this.Desligar();
36   }
37
38   public void Ligar() { ...
39
40   public void Desligar() {
41 }
42
43 public class Interruptor
44 {
45   private readonly IDispositivo _dispositivo;
46
47   public void AcionarDispositivo()
48   {
49     _dispositivo.Acionar();
50   }
51 }
```

1 - Criamos a interface **IDispositivo** para englobar todos os dispositivos

2 - Especializamos os dispositivos implementando o método **Acionar**() em ambos

3 - Nosso Interruptor agora pode depender da **abstração**, não de classes concretas!

Até a próxima!