Princípios S.O.L.I.D

Matheus Barbosa
matheus.barbosa@dcx.ufpb.br

Single Responsibility Principle (Princípio da responsabilidade única)

Open-Closed Principle (Princípio Aberto-Fechado)

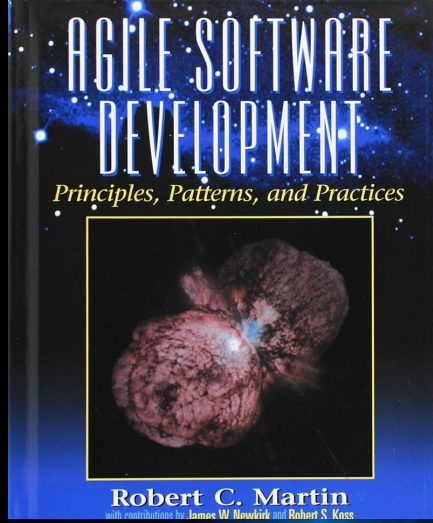
Liskov Substitution Principle (Princípio da substituição de Liskov)

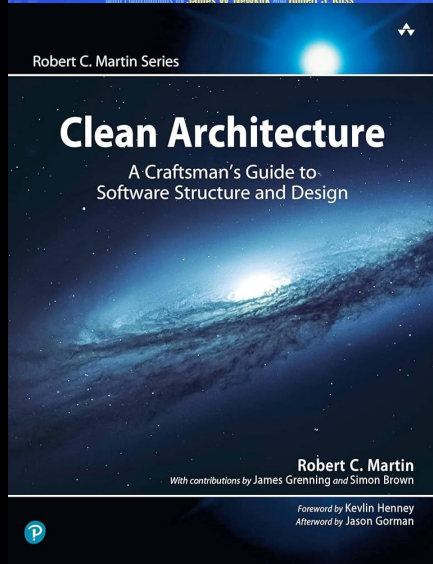
Interface Segregation Principle (Princípio da Segregação da Interface)

Dependency Inversion Principle (Princípio da inversão da dependência)

Criado por Michael Feathers.

A divulgação mais ampla veio com livros como "Agile Software Development, Principles, Patterns, and Practices" (2002) de Robert C. Martin, e depois reforçada em "Clean Architecture" (2017).





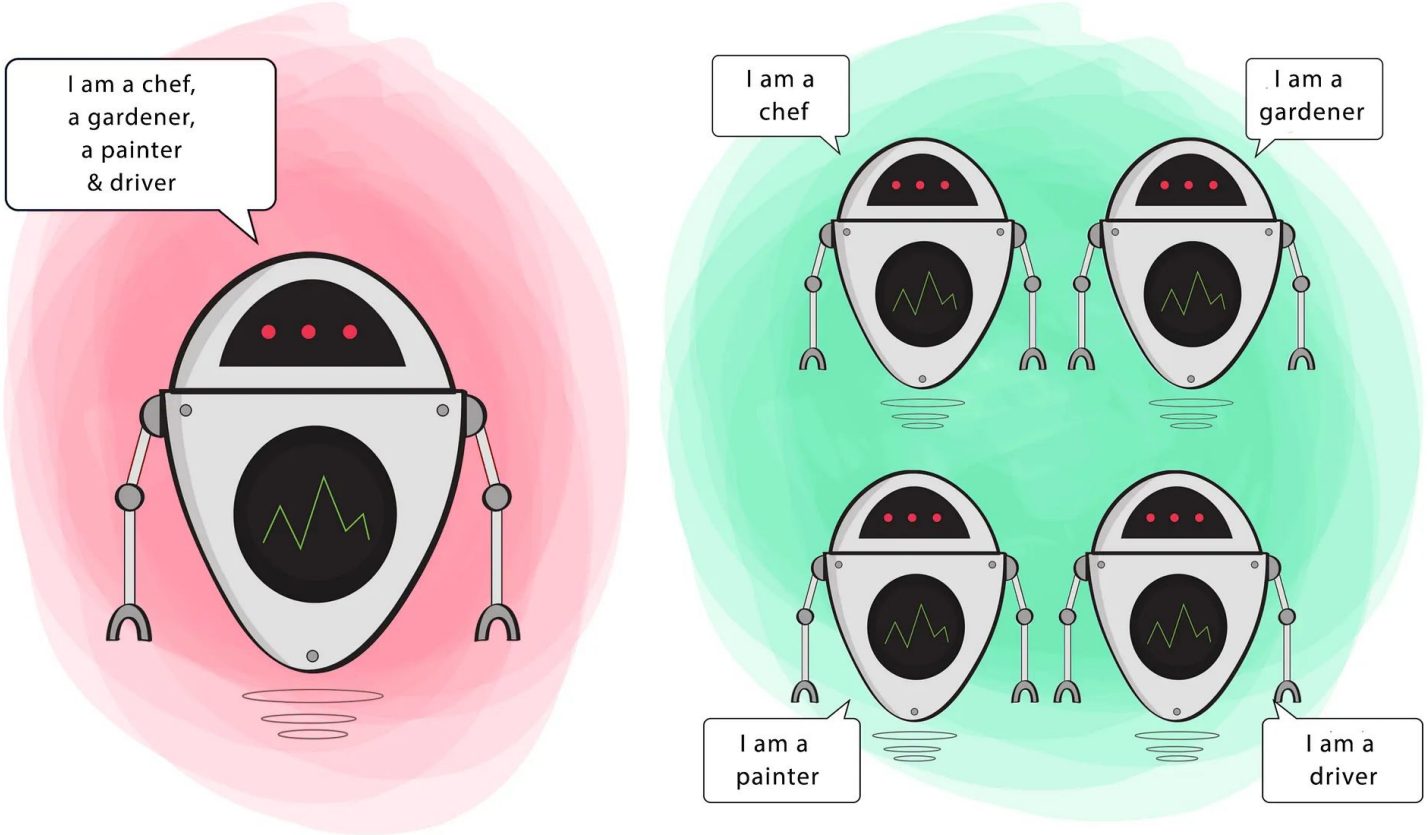
S.O.L.I.D

Single Responsibility Principle

(Princípio da responsabilidade única)

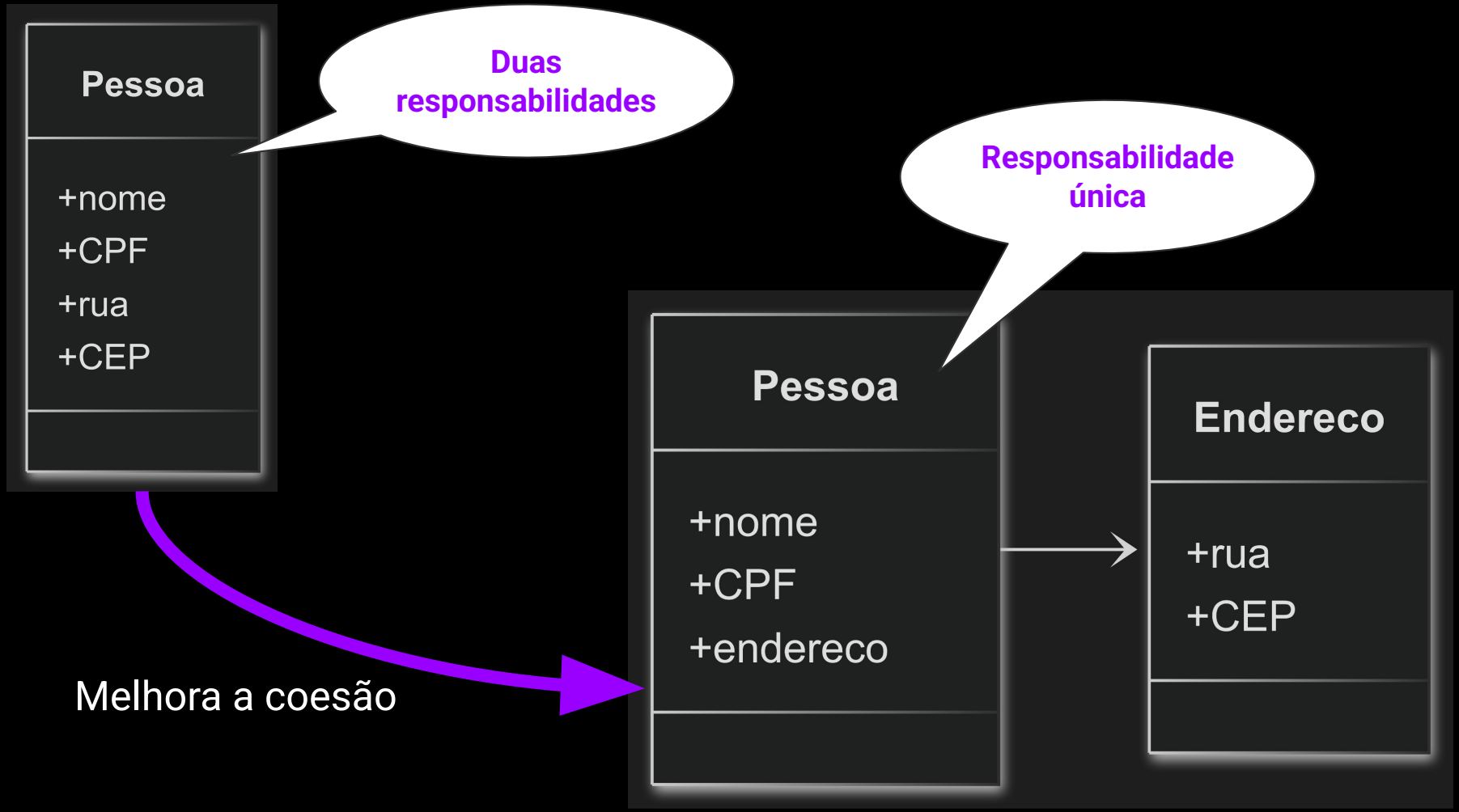
SRP (Princípio da responsabilidade única)

Uma classe deve ser responsável por uma única parte da funcionalidade do sistema.



Single Responsibility

https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898


Pessoa
+nome
+CPF
+rua
+CEP
Duas responsabilidades
Melhora a coesão
Responsabilidade única
Pessoa
+nome
+CPF
+endereco
Endereco
+rua
+CEP

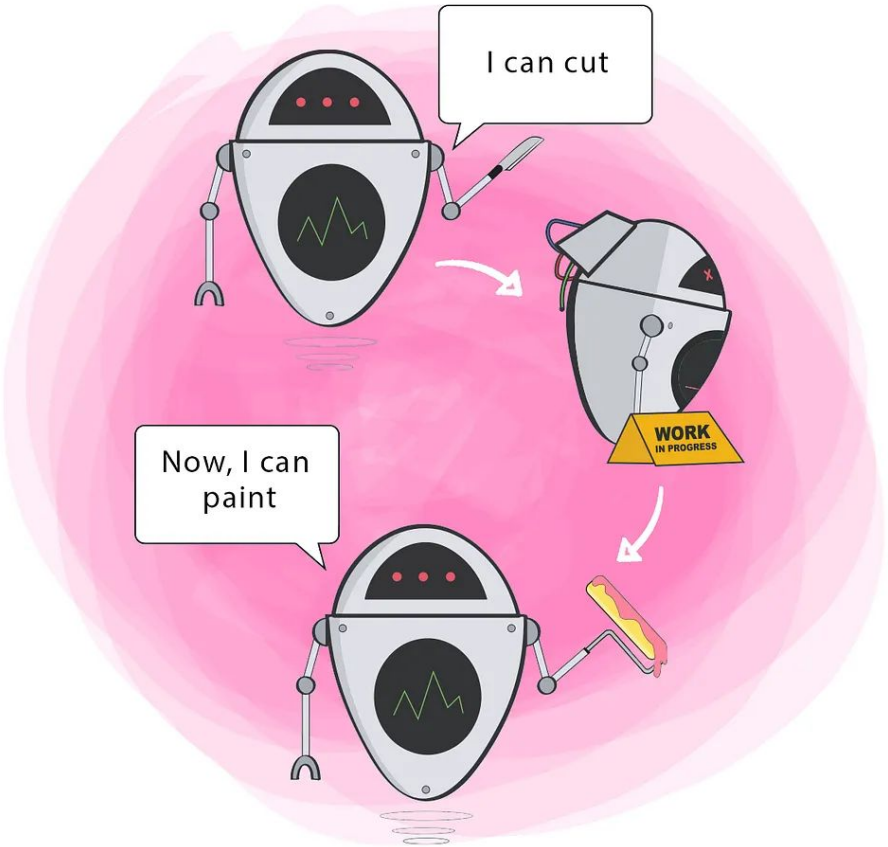
S.O.L.I.D

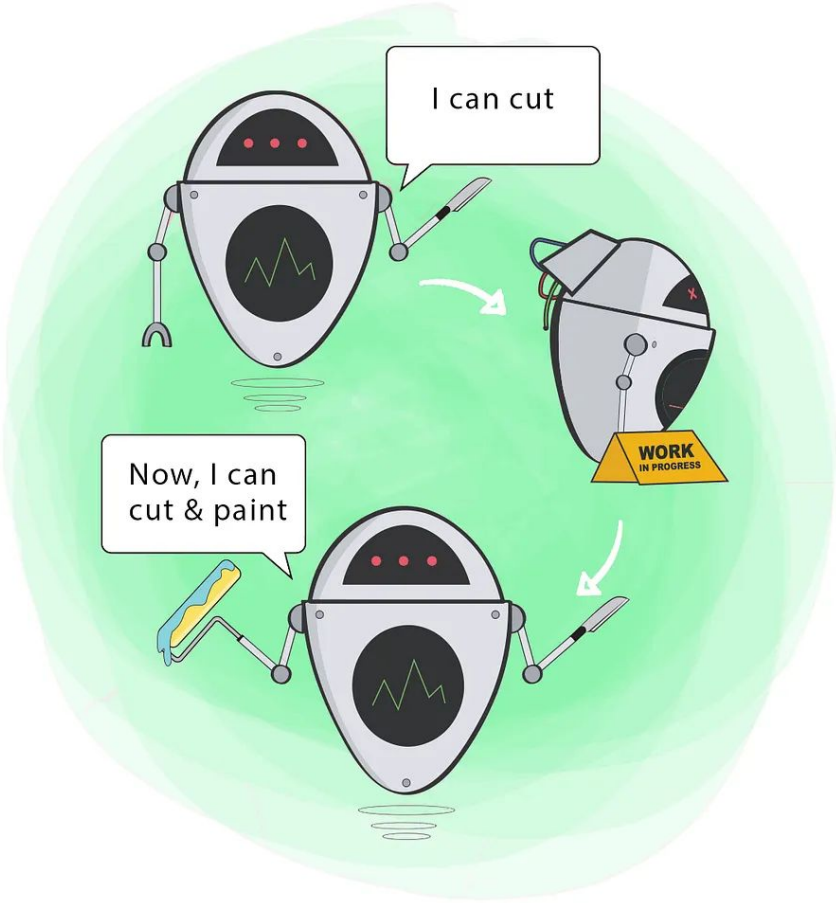
Open-Closed Principle

(Princípio Aberto-Fechado)

OCP (Princípio Aberto-Fechado)

Um módulo deve ser aberto para extensão, mas fechado para modificação.





Open-Closed

https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

```java
class Conta {
    double saldo;

    void creditar(double valor) {
        saldo = saldo + valor;
    }

    void debitar(double valor) {
        saldo = saldo - valor;
    }
    …
}
```

```java
class ContaEspecial extends Conta {
    double bonus;

    void creditar(double valor) {
        super.creditar(valor);
        bonus = bonus + (0.01 * valor);
    }
    …
}
```

Aberto para extensão

```java
Conta c = new Conta();
c.creditar(576.35);
```

fechado para modificação

Extensão exige modificação: sem uso de subtipos

Lidar com um novo tipo de conta exige uma nova condicional

```
class Conta {
    double saldo;
    String tipo;
    …
    void creditar(double valor) {
        if (tipo.equals("Basica")) {
            saldo = saldo + valor;
        } else {
            …
        }
    }
    …
}
```

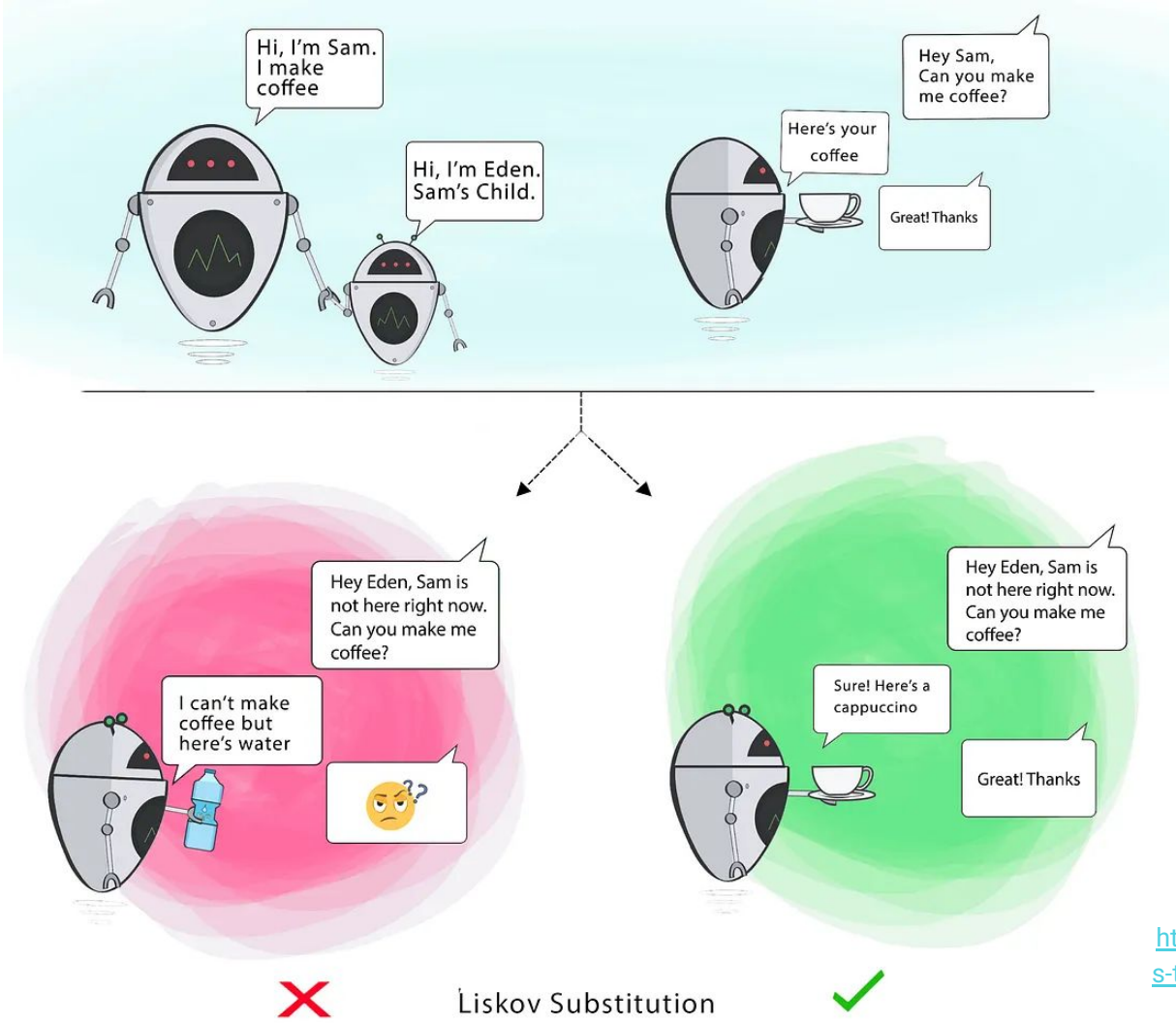
S.O.L.I.D

Liskov Substitution Principle

(Princípio da substituição de Liskov)

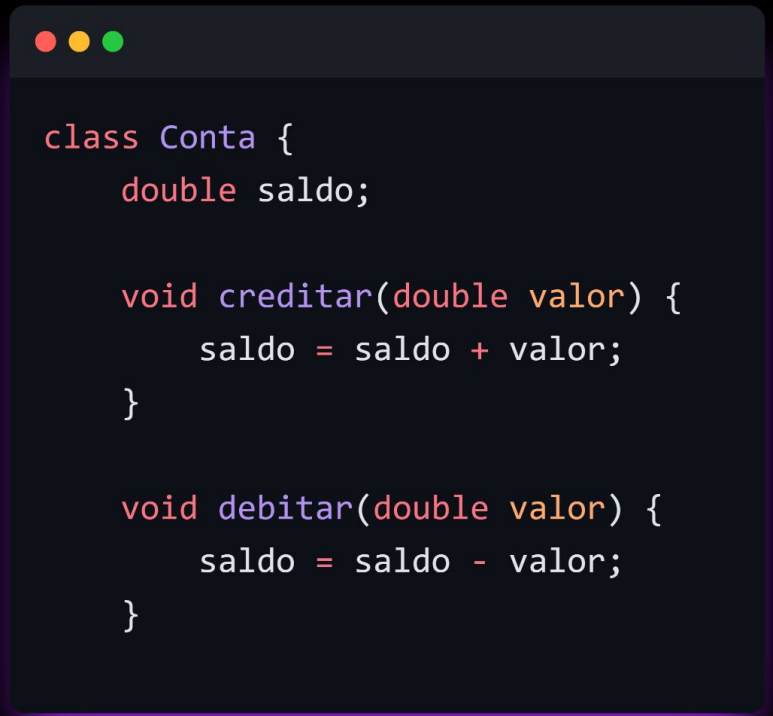
LSP (Princípio da substituição de Liskov)

Objetos de subclasses devem se comportar como os de superclasses se usados através dos métodos da superclasse

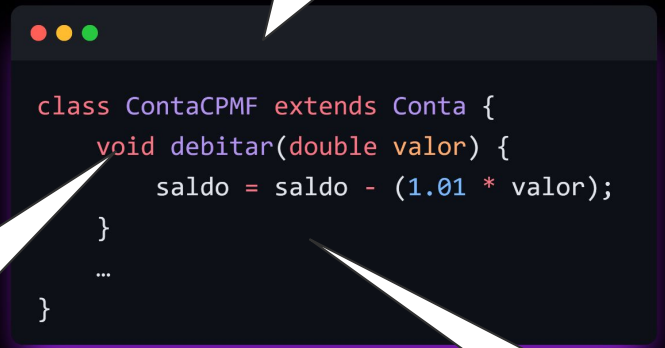


Liskov Substitution

https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898



clientes podem observar que objetos desta classe se comportam de forma diferente dos objetos de sua superclasse

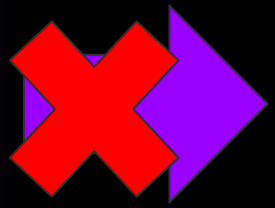


chame `debitar`, depois imprima o saldo e você pode observar a diferença

Quebra LSP

Evita-se qualquer raciocínio equivocado baseado em métodos abstratos

```
double m(Conta a) {
    a.creditar(v);
    …
    a.debitar(v);
    return a.getSaldo();
}
```

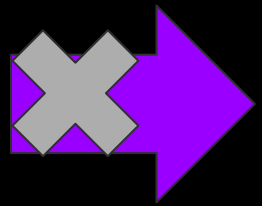


```
double m(Conta a) {
    …
    return a.getSaldo();
}
```

a declaração de método abstrato especifica que não podemos esperar um comportamento de método específico

Quebrar o LSP reduz a extensibilidade

```
double m(Conta a) {
    a.creditar(v);
    …
    a.debitar(v);
    return a.getSaldo();
}
```



```
double m(Conta a) {
    …
    return a.getSaldo();
}
```

Seguro, modificação que preserva o comportamento, baseada apenas no código de `Conta`, se o LSP for seguido

Possivelmente inseguro, modificação que não preserva o comportamento, se o LSP não for seguido (demanda análise de todas as subclasses de Conta)

Respeitando o LSP

Movendo comportamento incompatível da superclasse para subclasses irmãs

```
abstract class Conta {
    double saldo;

    void creditar(double v) {
        saldo = saldo + v;
    }

    void debitar(double v);
    …
```

```
class ContaPadrão extends Conta {
    void debitar(double v) {
        saldo = saldo - v;
    }
    …
}
```

```
class ContaCPMF extends Conta {
    void debitar(double v) {
        saldo = saldo - (1.01 * v);
    }
    …
}
```

LSP não impede subclasses adicionem métodos, campos e comportamentos extras.

```java
class Conta {
    double saldo;

    void creditar(double valor) {
        saldo = saldo + valor;
    }

    void debitar(double valor) {
        saldo = saldo - valor;
    }
    …
}
```

```java
class ContaEspecial extends Conta {
    double bonus;

    void creditar(double valor) {
        super.creditar(valor);
        bonus = bonus + (0.01 * valor);
    }
    …
}
```

Chame `creditar`, imprima o saldo e você não perceberá a diferença

os clientes só podem observar que os objetos desta classe se comportam de maneira diferente dos objetos de sua superclasse ao receber bônus

Satisfaz LSP

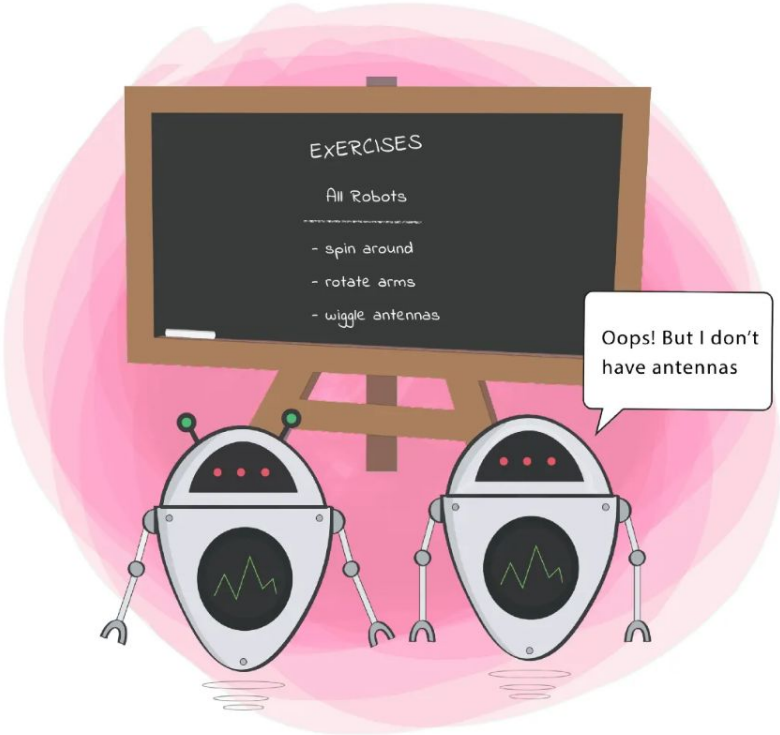
S.O.L.I.D

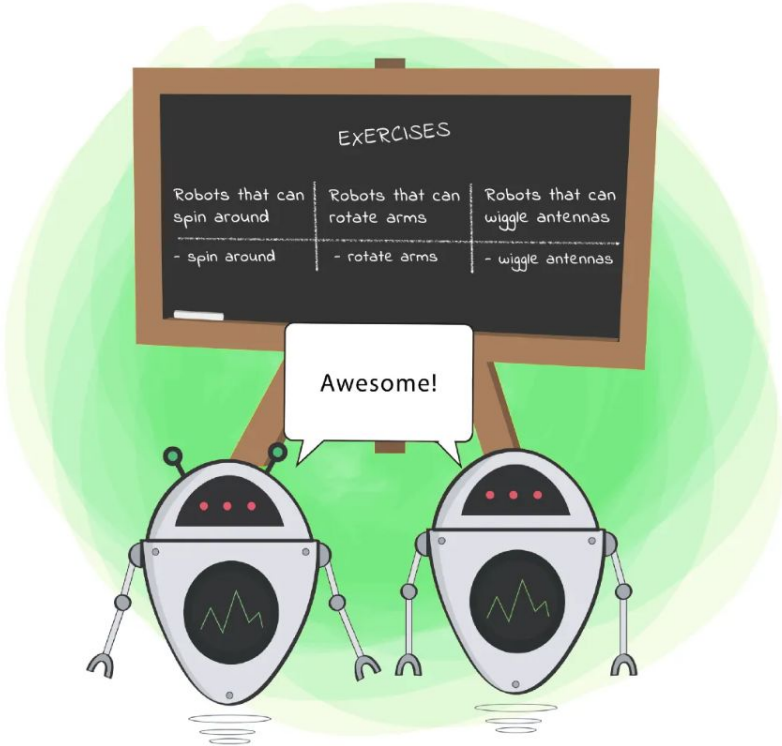
Interface Segregation Principle

(Princípio da Segregação da Interface)

ISP (Princípio da Segregação da Interface)

Os clientes não devem ser forçados a depender de métodos que não usam.

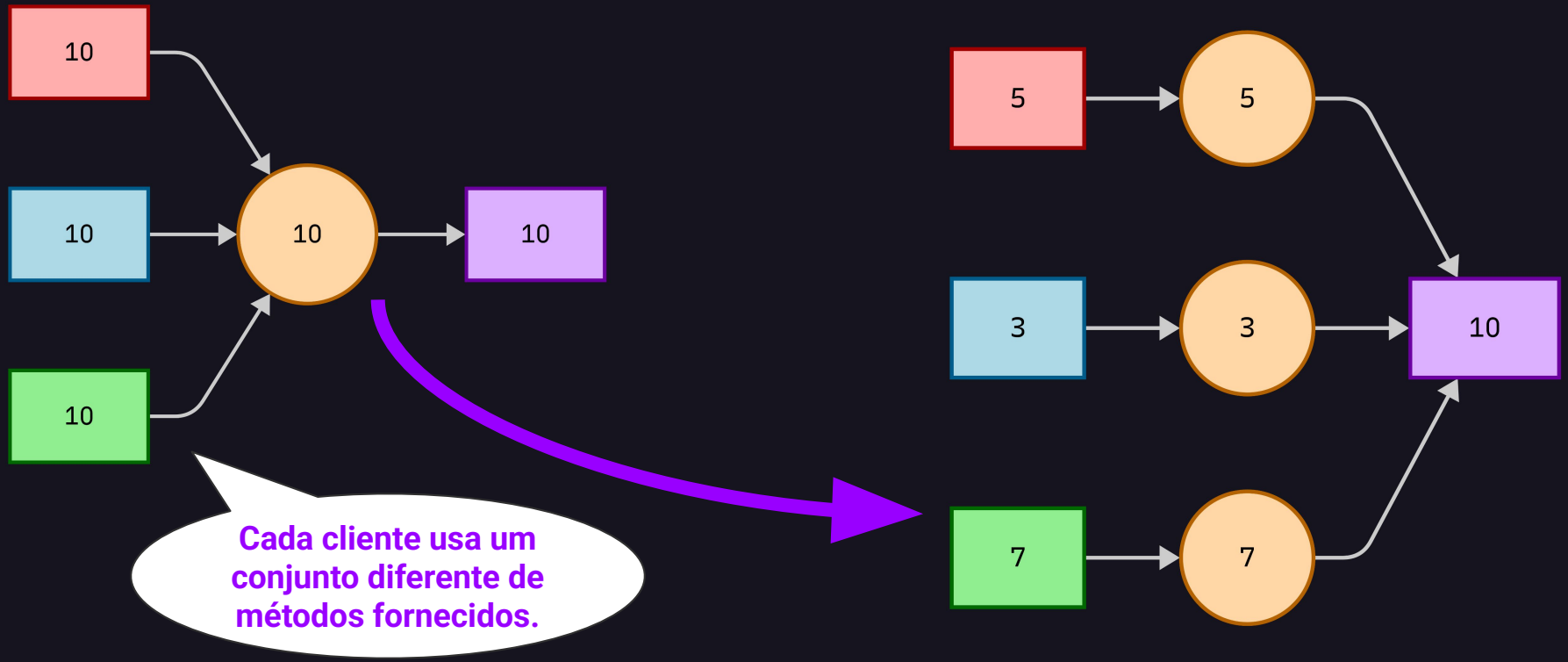




Interface Segregation

https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

Interfaces diferentes para diferentes demandas do clientes



Violação do ISP: O Risco de Interfaces Genéricas

```
interface Aves {
    bicar();
    voar();
}
```

```
class Gaviao implements Aves{
    bicar();
    voar();
}

class Pinguim implements Aves{
    bicar();
    voar(); ❌
}
```

Aplicando a ISP: A Vantagem de Interfaces Específicas

```
interface Aves {
    bicar();
}
```

```
interface AvesQueVoam extends Aves {
    voar();
}
```

```
class Gaviao implements AvesQueVoam{
    bicar();
    voar();
}

class Pinguim implements Aves{
    bicar();
}
```

S.O.L.I.D

Dependency Inversion Principle

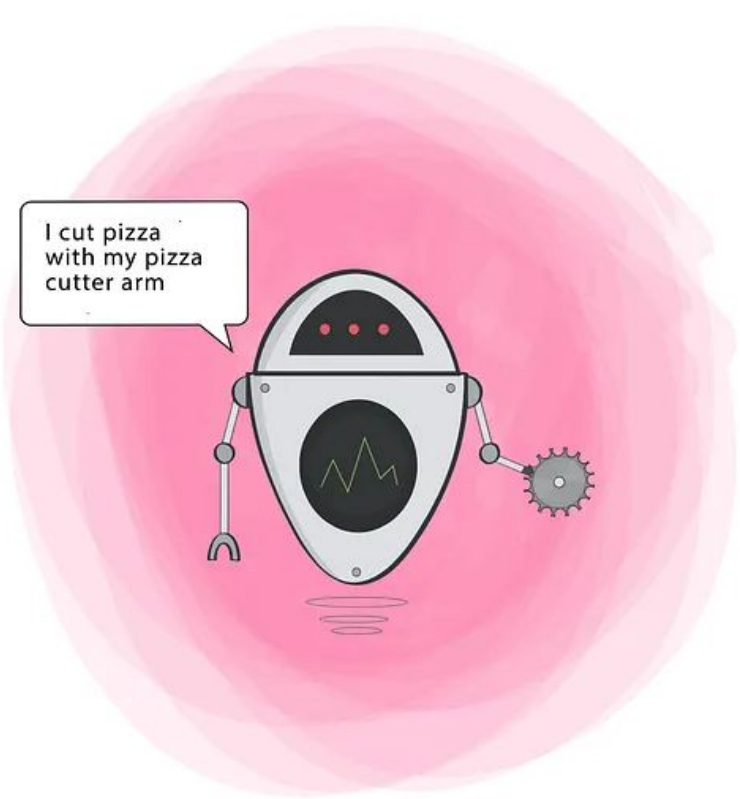
(Princípio da inversão da dependência)

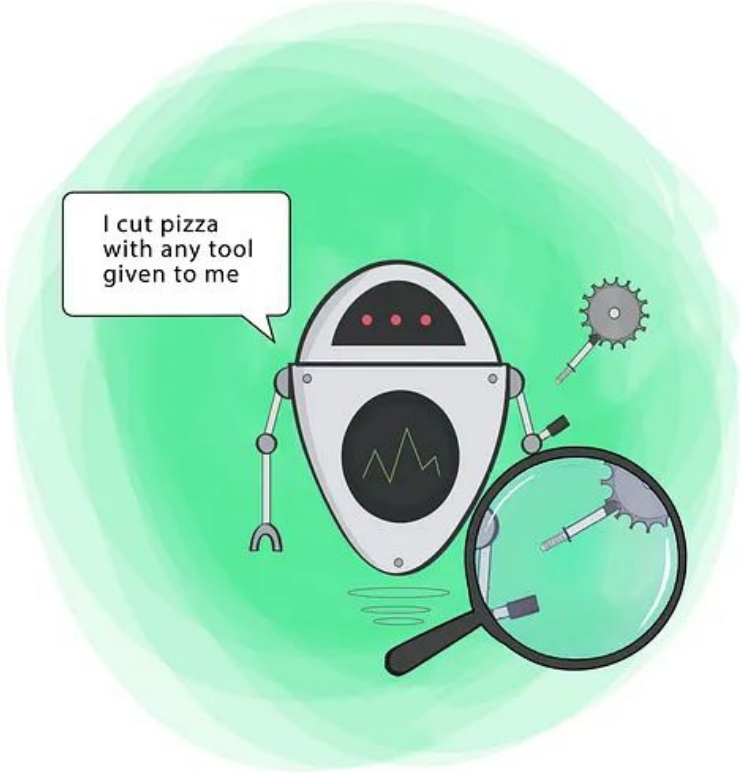
DIP (Princípio da inversão da dependência)

Um módulo não deve depender de detalhes de implementação de outro módulo diretamente. Ambos devem depender da abstração.

DIP (Princípio da inversão da dependência)

Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.





Dependency Inversion

https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

O efeito resultante de utilizar rigorosamente OCP e LSP pode ser generalizado como DIP

Robert C. Martin

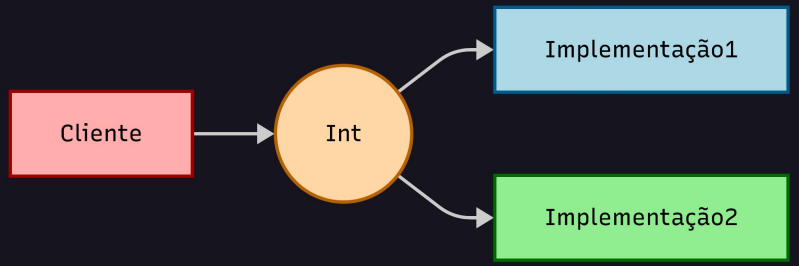
Robert C. Martin. Dependency Inversion Principle, 1996. Disponível em: <http://www.objectmentor.com/resources/articles/dip.pdf>.

Injeção de dependência: dependências devem ser injetadas em um objeto dependente

```
class Cliente {
    Int int;

    Cliente(){
        int = new Implementacao1();
    }
}
```

dependência criada internamente




```
c = new Cliente();
c.m();
c.n();
```

Dependência criada externamente

```
c = new Cliente();
c.setInt(new Implementacao1())
c.m();
c.n();
```

Alterando dependência em tempo de execução

```
class Cliente {
    Int int;

    void setInt(Int i){
        this.int = i;
    }
}
```

Trabalhando com dependência alternativa

```
c = new Cliente();
c.setInt(new Implementacao1());
c.m();
...
c = new Cliente();
c.setInt(new Implementacao2());
c.m();
```