Modelos, Métodos e Técnicas de Engenharia de Software

Prof.: Sônia Aparecida Santana

una

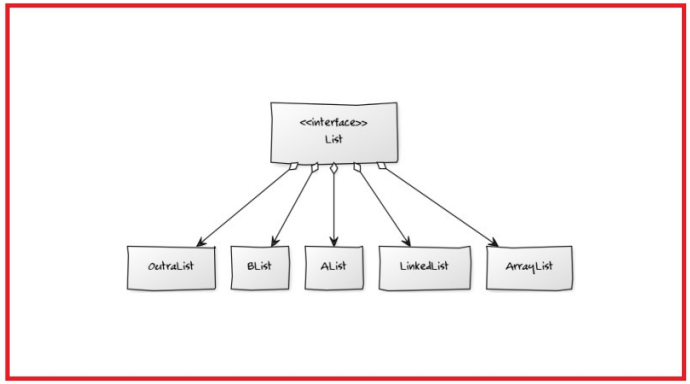
ACOPLAMENTO

Conceito

- Acoplamento é a força (strength) da conexão entre duas classes.
- Podem ser de dois tipos:
  - acoplamento aceitável
  - acoplamento ruim.

Acoplamento Aceitável

- Classe A depende de uma classe B:
  - Mas a classe B possui uma interface estáve: assinaturas dos métodos não mudam com frequência.
  - Classe A somente invoca métodos públicos de B



Acoplamento Aceitável

```
import java.util.Hashtable;

public class Estacionamento {

  private Hashtable<String,String> veiculos;

  public Estacionamento() {
    veiculos = new Hashtable<String, String>();
  }

  public void estaciona(String veiculo, String placa) {
    veiculos.put(veiculo, placa);
  }

  public static void main(String[] args) {
    Estacionamento e = new Estacionamento();
    e.estaciona("TCP-7030", "Uno");
    e.estaciona("BNF-4501", "Gol");
    e.estaciona("JKL-3481", "Corsa");
  }

}
```

Classe Estacionamento depende (está acoplada) à classe Hashtable, mas esse acoplamento é aceitável

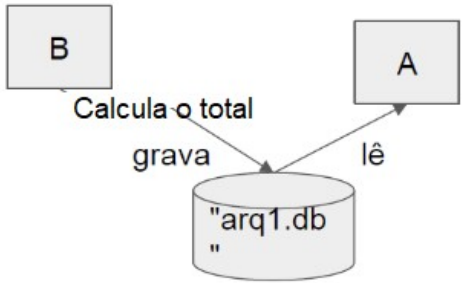
Acoplamento Ruim

- Por outro lado, existe um **acoplamento ruim** de uma classe A para uma classe B quando mudanças em B podem facilmente impactar A. Isso ocorre principalmente nas seguintes situações:
  - Quando a classe A realiza um acesso direto a um arquivo ou banco de dados da classe B.
  - Quando as classes A e B compartilham uma variável ou estrutura de dados global. Por exemplo, a classe B altera o valor de uma variável global que a classe A usa no seu código.
  - Quando a interface da classe B não é estável. Por exemplo, os métodos públicos de B são renomeados com frequência.

Acoplamento Ruim

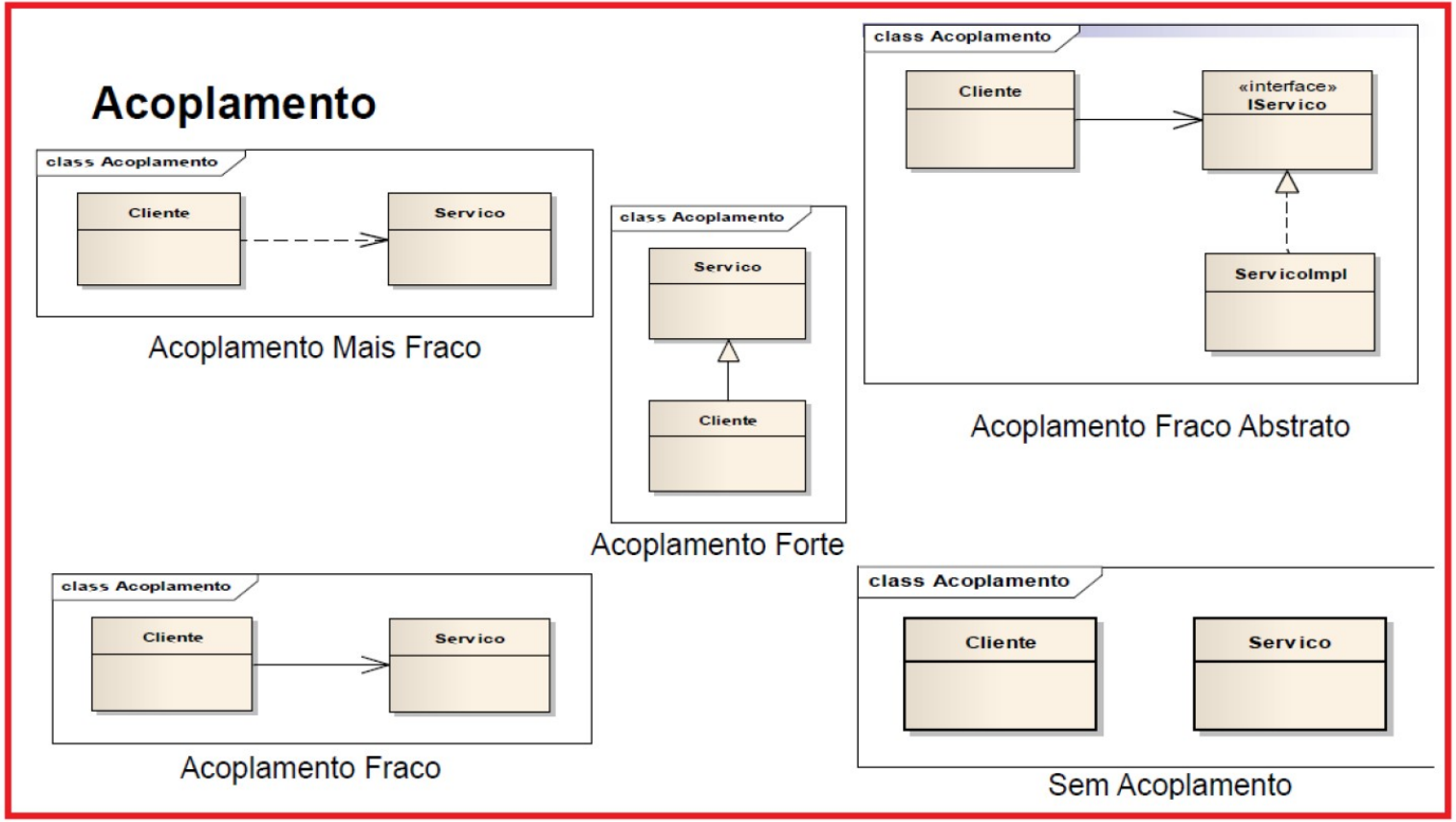
```
class A {

  private void f() {
    int total;
    ...
    File f = File.open("arq1.db");
    total = f.readInt();
    ...
  }

}
```

```
class B {
  private void g() {
    int total;
    // computa valor de total
    File f = File.open("arq1.db");
    f.writeInt(total);
    ...
    f.close();
  }
}
```



- Classe B não sabe que a classe A é sua "cliente"
- Logo, B pode mudar o formato do arquivo ou mesmo deixar de salvar o dado que é lido por A

Exemplos


Acoplamento
class Acoplamento
Cliente
Servico
Acoplamento Mais Fraco
class Acoplamento
Servico
Cliente
Acoplamento Forte
class Acoplamento
Cliente
«interface»
IServico
ServicoImpl
Acoplamento Fraco Abstrato
class Acoplamento
Cliente
Servico
Acoplamento Fraco
class Acoplamento
Cliente
Servico
Sem Acoplamento

Frase muito comum

Maximize a coesão, minimize o acoplamento

Mas cuidado: minimize (ou elimine) principalmente o acoplamento ruim

Observações Importantes

- **Com uma classe possuindo forte acoplamento, temos os seguintes problemas:**
  - Mudanças a uma classe relacionada força mudanças locais à classe
  - A classe é mais difícil de entender isoladamente
  - A classe é mais difícil de ser reusada, já que depende da presença de outras classes
  - Causa o efeito cascata(ou dominó)
- Deve-se tentar **minimizar** o **acoplamento** para evitar a propagação de mudanças e para possibilitar a reutilização de classes.
  - Pedaços menos acoplados são mais fáceis de entender, testar, reusar e manter
  - Promove o paralelismo de implementação

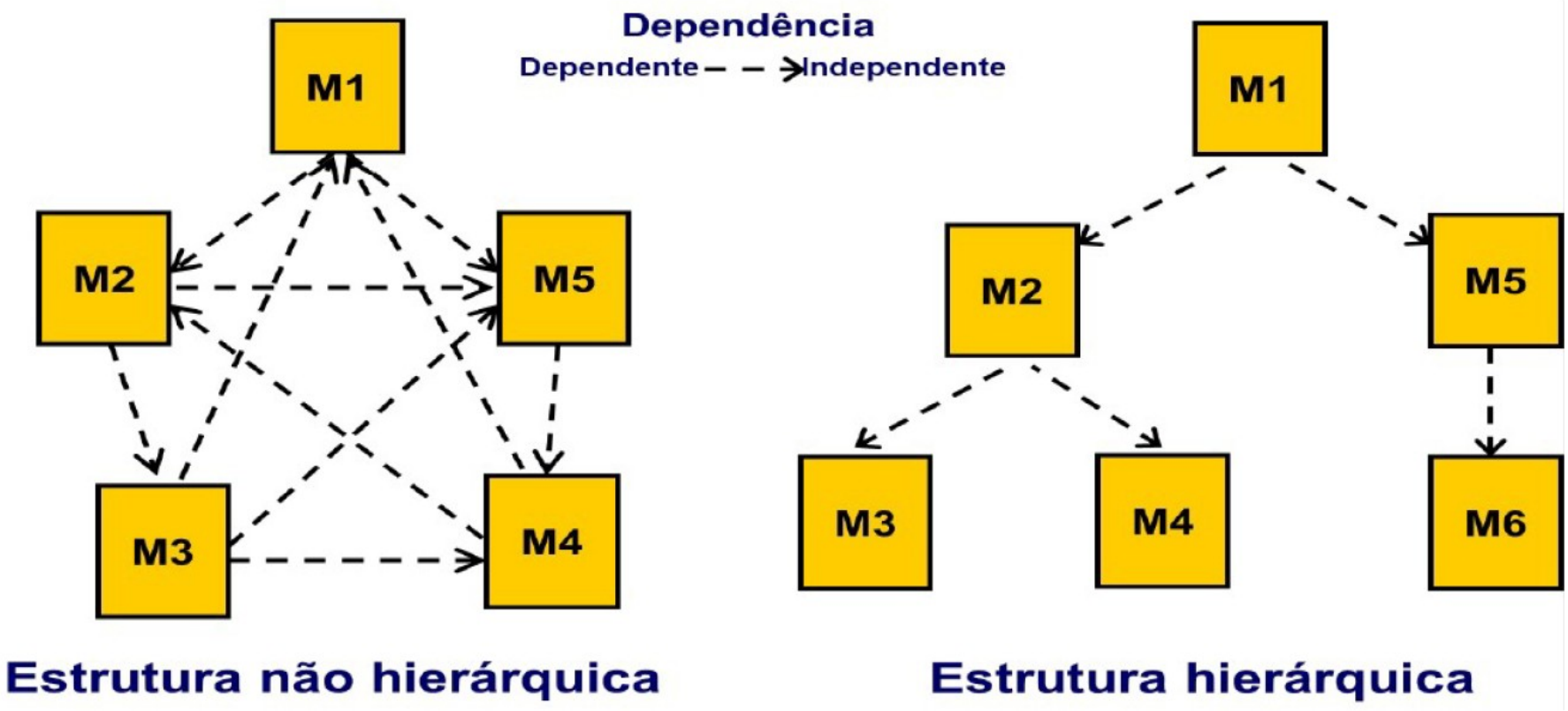
Observações Importantes

- **Minimizar o acoplamento é um dos princípios de ouro do projeto OO**
- **Acoplamento se manifesta de várias formas:**
  - X tem um atributo que referencia uma instância de Y
  - X tem uma operação que referencia uma instância de Y
    - Pode ser parâmetro, variável local, objeto retornado pela operação
  - X é uma subclasse direta ou indireta de Y
  - X implementa a interface Y
- **A herança é um tipo de acoplamento particularmente forte**
- **Não se deve minimizar o acoplamento criando algumas poucas classes monstruosas (*God classes*)**

Observações Importantes

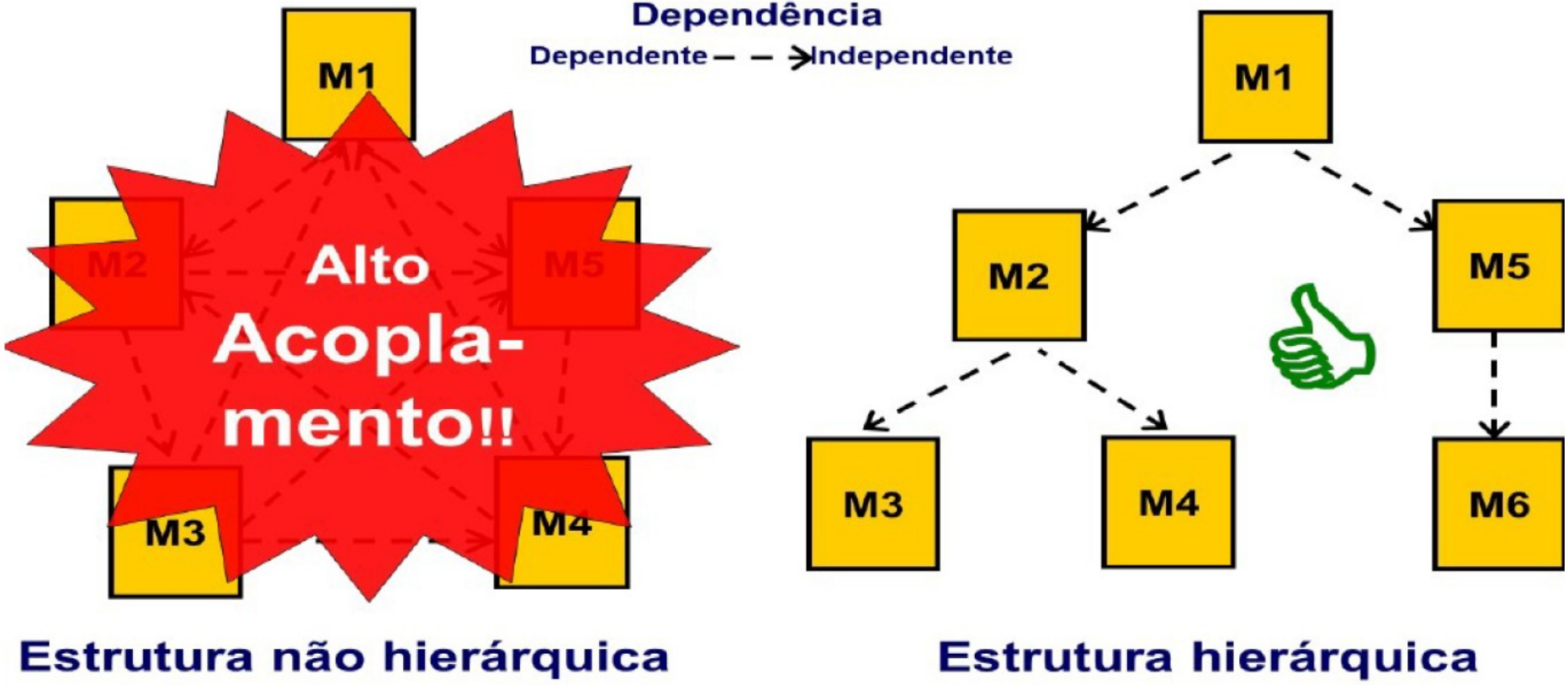
- Se o único conhecimento que uma classe A tem sobre uma classe B é o que a classe B expôs através de sua interface, então as classes A e B têm um **baixo nível** de acoplamento.
- Se a classe A sabe sobre partes da classe B que não são interfaces de B, então as classes A e B têm um **alto nível** de acoplamento.
- Sendo a classe A conectada a classe B o que aconteceria com a classe A caso seja necessário uma **alteração** na classe B ?
- Significa que:
  - Se B for modificado, A terá que ser modificado ou ao menos verificado
  - Pode ocorrer uma modificação no sistema que obrigue modificações conjuntas em A e B

Modularização e Componentização



Modularização e Componentização

Modularização e Componentização



PRINCÍPIOS DE PROJETO

Técnicas de Elicitação de Requisitos

Requisitos Funcionais
Requisitos Não Funcionais
Regras de Negócios

Documento de Especificação de Requisitos

Princípios (SOLID) e padrões de projeto
Modelos Arquiteturais

Modelagem de Negócio | Engenharia de Requisitos | Análise e Projeto

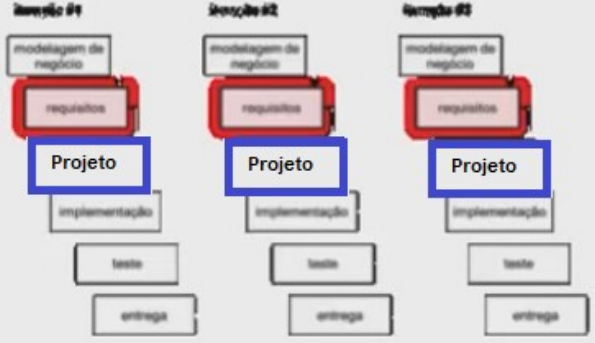
Implementação | Teste | Implantação

Ambiente | Gerência de Configuração e Mudança | Gerência de Projetos

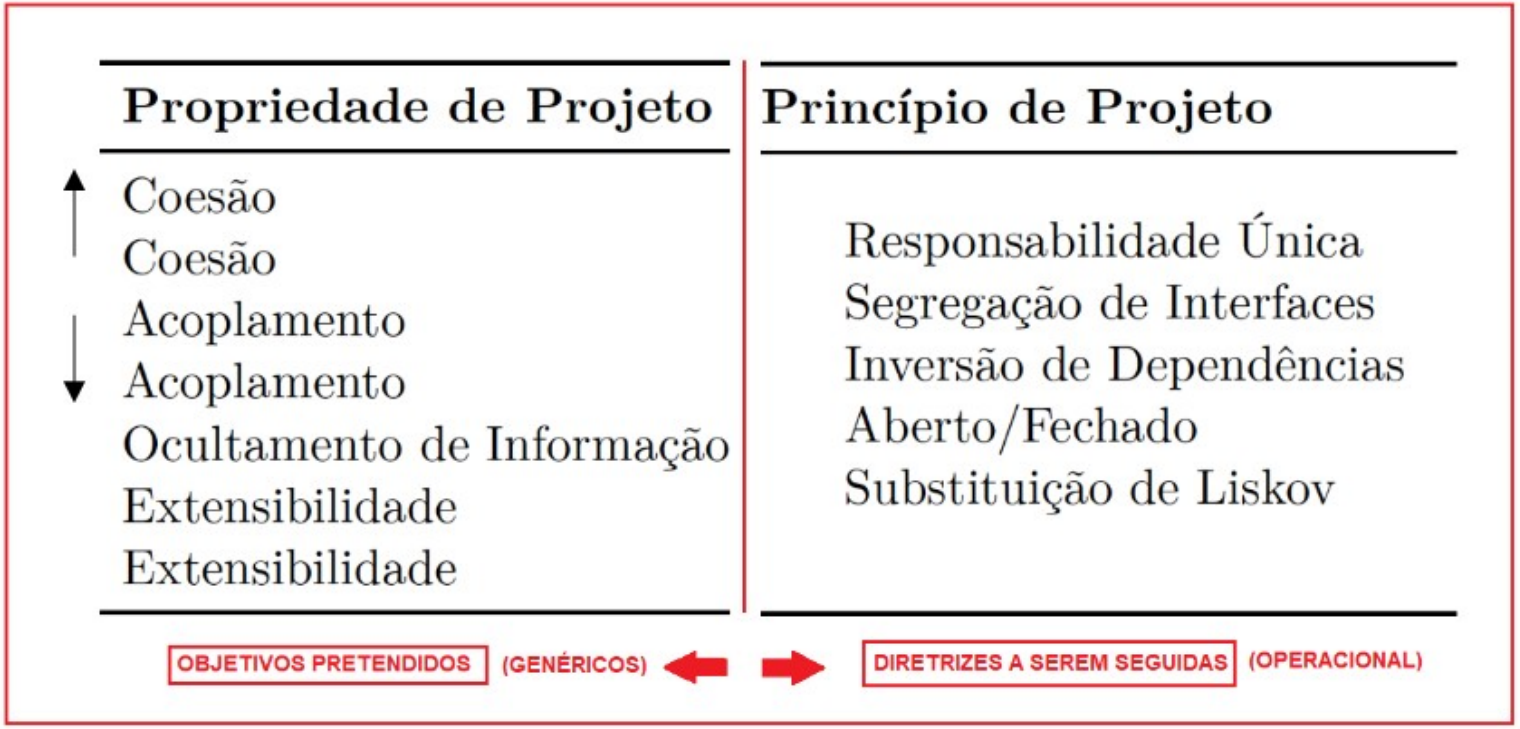
Sequencial

Iterativo e Incremental

Estratégia de Desenvolvimento



Introdução


Propriedade de Projeto
Princípio de Projeto
Coesão
Coesão
Acoplamento
Acoplamento
Ocultamento de Informação
Extensibilidade
Extensibilidade
Responsabilidade Única
Segregação de Interfaces
Inversão de Dependências
Aberto/Fechado
Substituição de Liskov
OBJETIVOS PRETENDIDOS
(GENÉRICOS)
DIRETRIZES A SEREM SEGUIDAS
(OPERACIONAL)

Princípios de Projeto

- Princípios de projeto são recomendações mais concretas que desenvolvedores de software devem seguir para atender às propriedades de projeto vistas anteriormente.
- Assim, propriedades de projeto podem ser vistas como recomendações ainda genéricas (ou táticas), enquanto que os princípios estão em um nível operacional.

Princípios de Projeto: SOLID

Single Responsability Principle	Princípio da Responsabilidade Única
Open-Closed Principle	Princípio Aberto-Fechado
Liskov Substitution Principle	Princípio da Substituição de Liskov
Interface Segretation Principle	Princípio de Segregação de Interface
Dependency Inversion Principle	Princípio de Inversão de Dependência

Vantagens do SOLID

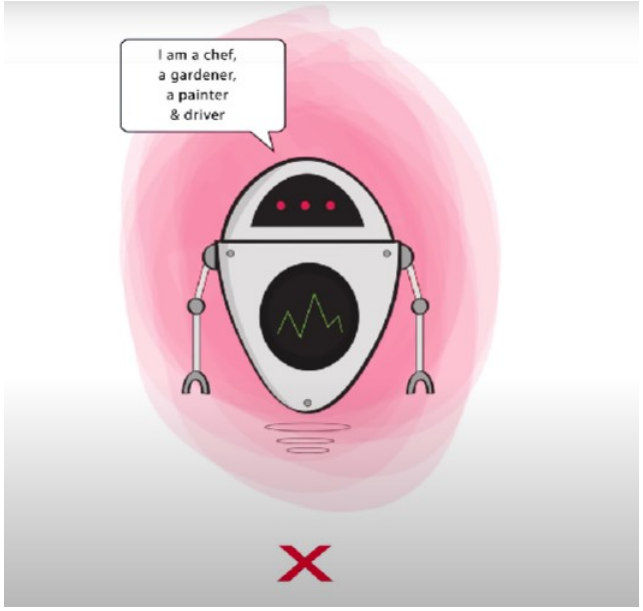
- O uso dos princípios SOLID no design orientado a objetos torna o código mais fácil de entender, gerenciar, manter e alterar.
- O princípio SOLID ajuda a reduzir o acoplamento forte, ou seja: um grupo de classes é altamente dependente um do outro. Um código é considerado como um bom quando ele tem classes fracamente acopladas. Classes fracamente acopladas minimizam as alterações em seu código, ajudam a tornar o código mais reutilizável, sustentável, flexível e estável.
- O SOLID fornece princípios de design sólidos aplicáveis a programas e ambientes modernos.

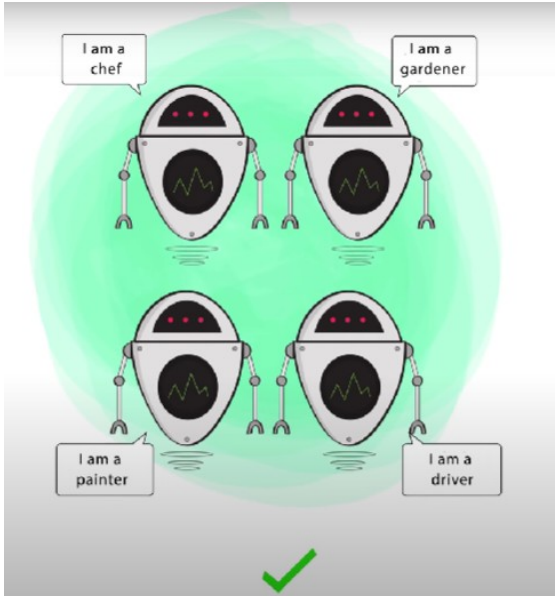
Princípio da Responsabilidade Única

Princípio da Responsabilidade Única

- Cada classe, componente, função, etc. tem que ter uma e somente uma razão para mudar. A não obediência a esse princípio traz problemas tanto para a reutilização e manutenção quanto à testabilidade

Princípio da Responsabilidade Única





- **REAPROVEITAMENTO DE CÓDIGO**
- **REFATORAÇÃO**
- **AUTOMATIZAÇÃO DE TESTES**
- **GERAR MENOS BUGS**

Em código....

```
class Disciplina {

  void calculaIndiceDesistencia() {
    indice = "calcula índice de desistência"
    System.out.println(indice);
  }

}
```

Exemplo: A classe ilustra uma violação do Princípio da Responsabilidade Única. O método calculaIndiceDesistencia da classe Disciplina possui duas responsabilidades:

- calcular o índice de desistência de uma disciplina
- imprimi-lo no console do sistema.

```
class Console {

  void imprimeIndiceDesistencia(Disciplina disciplina) {
    double indice = disciplina.calculaIndiceDesistencia();
    System.out.println(indice);
  }

}

class Disciplina {

  double calculaIndiceDesistencia() {
    double indice = "calcula índice de desistência"
    return indice;
  }

}
```

Exercício: Responsabilidade Única

```java
public class Employee {

    private String fullName;
    private String dateOfJoining;
    private String annualSalaryPackage;

    // standard getters and setters methods

    // business logic
    public long calculateEmployeeSalary(Employee emp) {...}
    public long calculateEmployeeLeaves(Employee emp) {...}
    public long calculateTaxOnSalary(Employee emp) {...}

    // data persistence logic
    public Employee saveEmployee(Employee emp) {...}
    public Employee updateEmployee(Employee emp) {...}
}
```

Princípio Aberto-Fechado

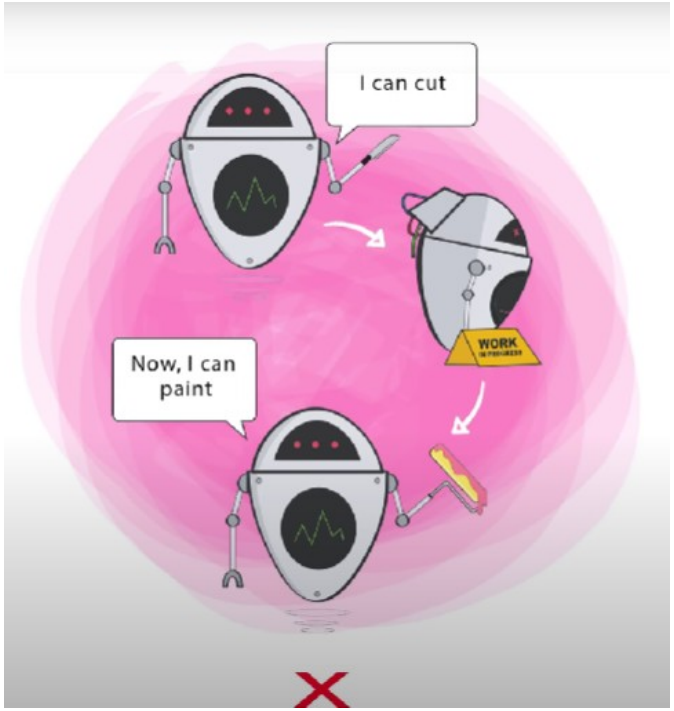
Proposto por Bertrand Meyer

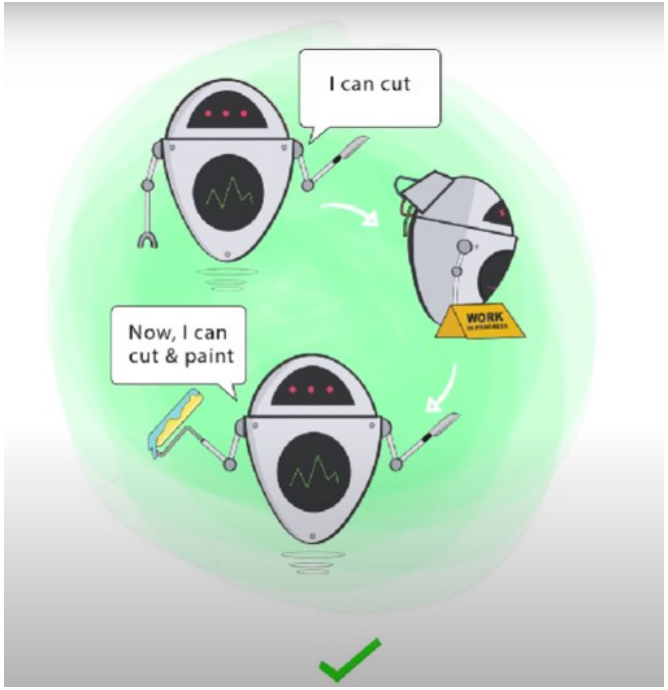
- Ideia: uma classe deve estar **fechada** para modificações, mas **aberta** para extensões

Princípio Aberto-Fechado

- “A classe deve ser Aberta para Extensão, mas Fechada para Modificação”.
- O Princípio Aberto-Fechado declara que as entidades do software devem ser projetadas de maneira a permitir que novos comportamentos sejam adicionados sem a necessidade de modificar o código existente. Isso significa que, ao estender as funcionalidades do sistema, devemos ser capazes de fazer isso através de adição de novas classes ou módulos, em vez de alterar o código existente.

Princípio Aberto-Fechado





Em Código

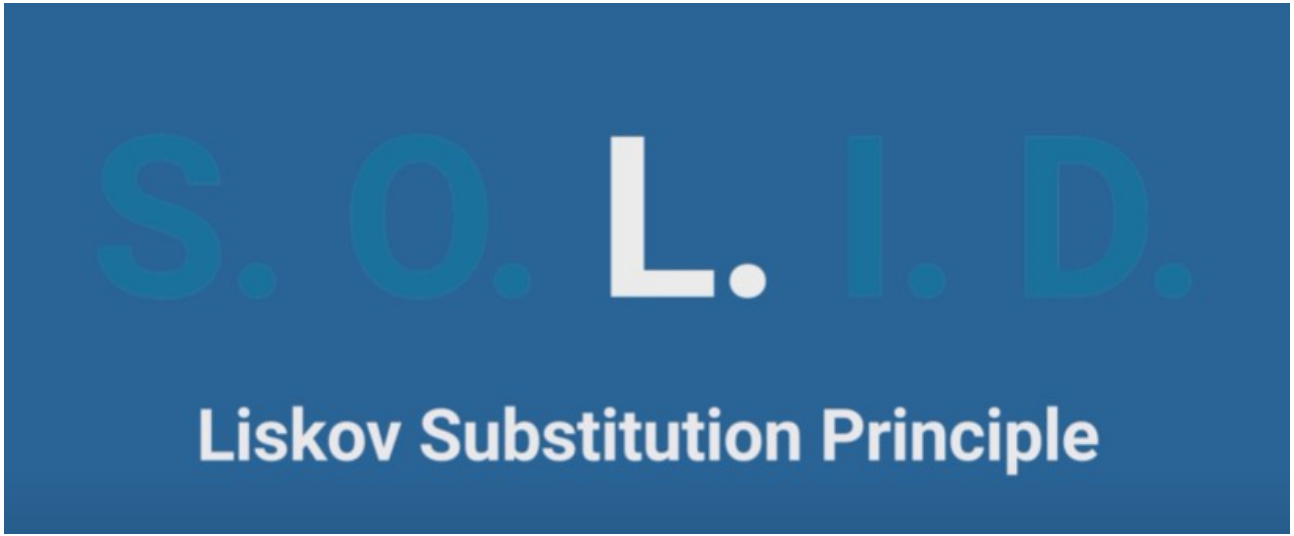
```
class Shape {
    private String type;

    public void draw() {
        if (type.equals("circle")) {
            drawCircle();
        } else if (type.equals("rectangle")) {
            drawRectangle();
        } else if (type.equals("triangle")) {
            drawTriangle();
        }
    }

    private void drawCircle() {
        // Desenhar um círculo
    }

    private void drawRectangle() {
        // Desenhar um retângulo
    }

    private void drawTriangle() {
        // Desenhar um triângulo
    }
}
```

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        // Desenhar um círculo
    }
}

class Rectangle implements Shape {
    public void draw() {
        // Desenhar um retângulo
    }
}

class Triangle implements Shape {
    public void draw() {
        // Desenhar um triângulo
    }
}
```

Exercício: Aberto-Fechado

```
public class EmployeeSalary {

    public Long calculateSalary(Employee emp) {
        Long salary = null;

        if (emp.getType().equals("PERMANENT")) {

          salary = (totalWorkingDay * basicPay) + getCompanyBenefits() + getBonus();
        } else if (emp.getType().equals("CONTRACT")) {

          salary = (totalWorkingDay * basicPay);
        }
        return salary;
    }
}
```

Princípio da Substituição de Liskov

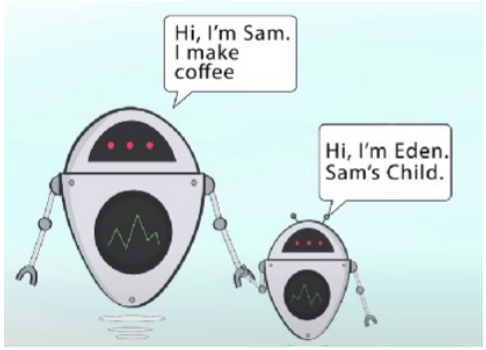


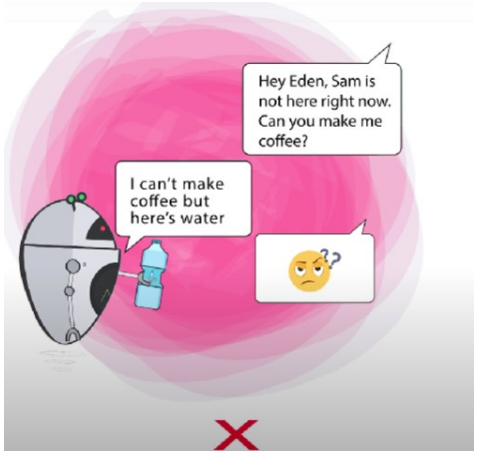
O Princípio de Substituição de Liskov explicita regras para redefinição de métodos de classes base em classes filhas.

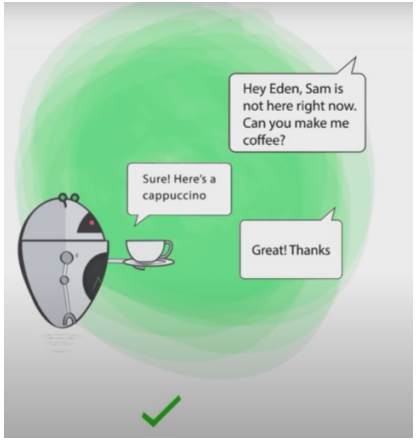
Princípio da Substituição de Liskov

- “Objetos devem ser substituíveis por instâncias de seus subtipos (filhos), sem alterar a funcionalidade do programa.”

Princípio da Substituição de Liskov







Esquematicamente...

```
public abstract class Passaro {

    public abstract void bicar();

    public abstract void voar();

}
```

```
public class BemTeVi extends Passaro {

    @Override
    public void bicar() {
        System.out.println("Sim eu consigo bicar😎!");
    }

    @Override
    public void voar() {
        System.out.println("Sim eu consigo voar🕊!");
    }

}
```

```
public class Pinguim extends Passaro {

    @Override
    public void voar() {
        throw new UnsupportedOperationException("eu não consigo voar⛔!");
    }

    @Override
    public void bicar() {
        System.out.println("Sim eu consigo bicar😎!");
    }

}
```

Em Código...

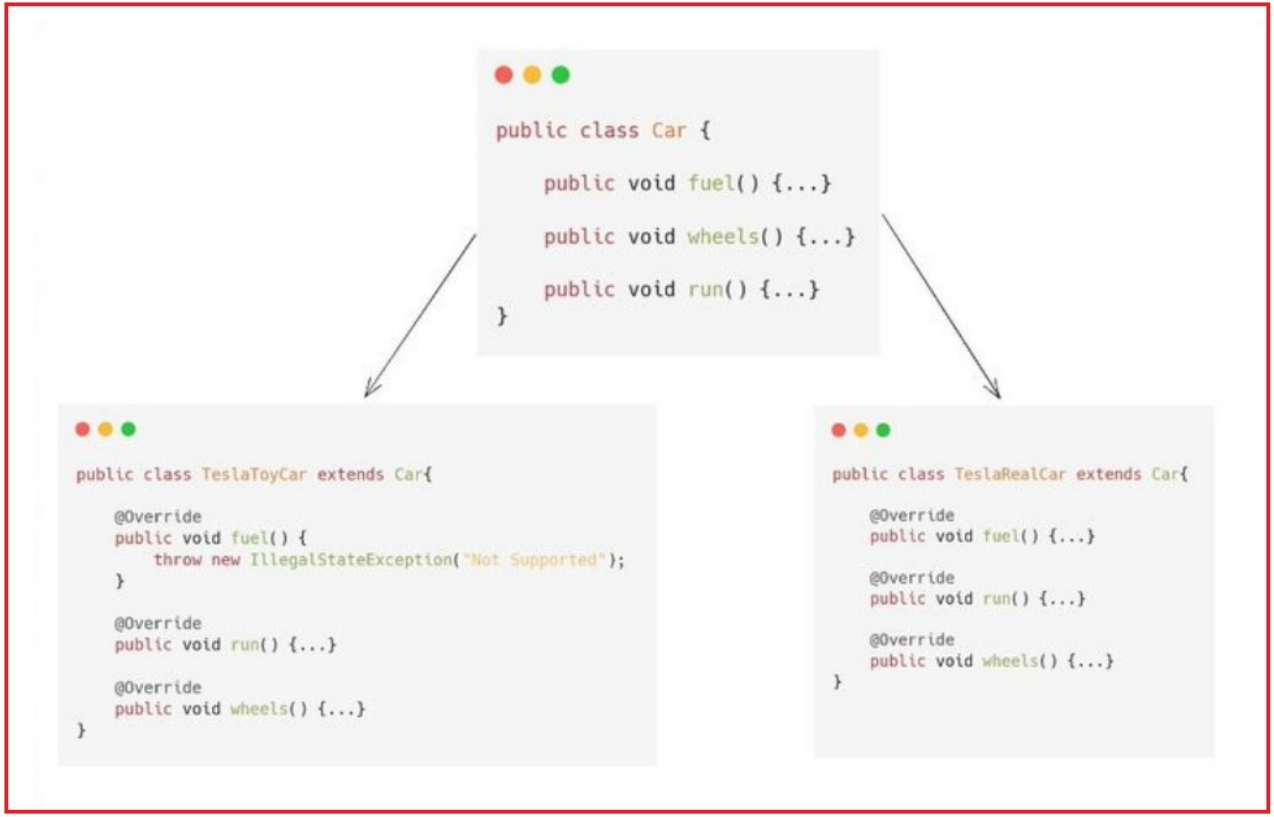
```java
class Retangulo {
    protected int largura;
    protected int altura;

    public void setLargura(int largura) {
        this.largura = largura;
    }

    public void setAltura(int altura) {
        this.altura = altura;
    }

    public int calcularArea() {
        return largura * altura;
    }
}
```

```java
class Quadrado extends Retangulo {
    public void setLado(int lado) {
        this.largura = lado;
        this.altura = lado;
    }
}
```

```java
abstract class Forma {
    abstract int calcularArea();
}

class Retangulo extends Forma {
    protected int largura;
    protected int altura;

    public void setLargura(int largura) {
        this.largura = largura;
    }

    public void setAltura(int altura) {
        this.altura = altura;
    }

    @Override
    int calcularArea() {
        return largura * altura;
    }
}
```

```java
class Quadrado extends Forma {
    protected int lado;

    public void setLado(int lado) {
        this.lado = lado;
    }

    @Override
    int calcularArea() {
        return lado * lado;
    }
}
```

Exercício: Substituição de Liskov



```java
public class Car {

    public void fuel() {...}

    public void wheels() {...}

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

```java
public class TeslaToyCar extends Car{

    @Override
    public void fuel() {
        throw new IllegalStateException("Not Supported");
    }

    @Override
    public void run() {...}

    @Override
    public void wheels() {...}
}
```

```java
public class TeslaRealCar extends Car{

    @Override
    public void fuel() {...}

    @Override
    public void run() {...}

    @Override
    public void wheels() {...}
}
```

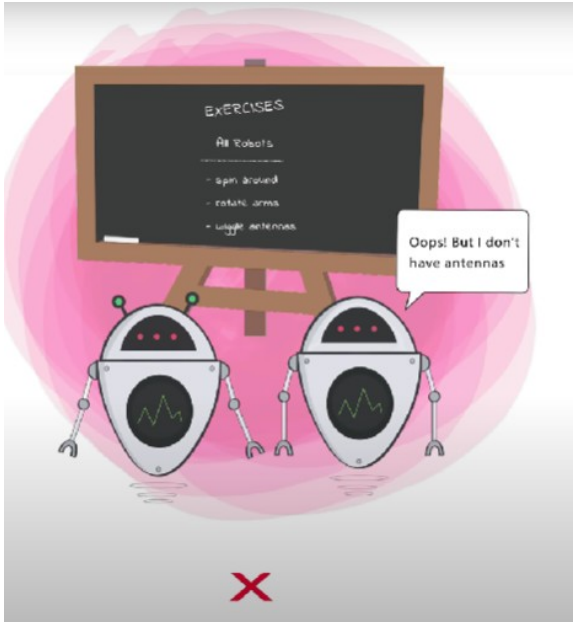
Princípio da Segregação da Interface

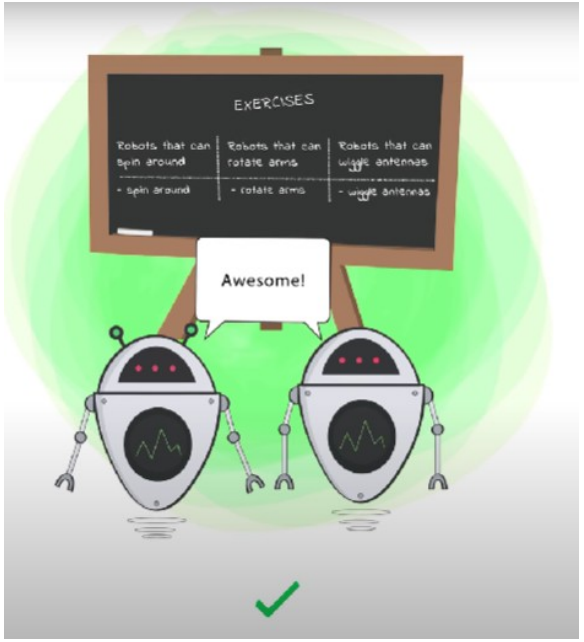
- Interfaces devem ser pequenas, coesas e específicas para cada tipo de cliente
- Caso particular do princípio anterior, mas voltado para interfaces

Princípio da Segregação da Interface

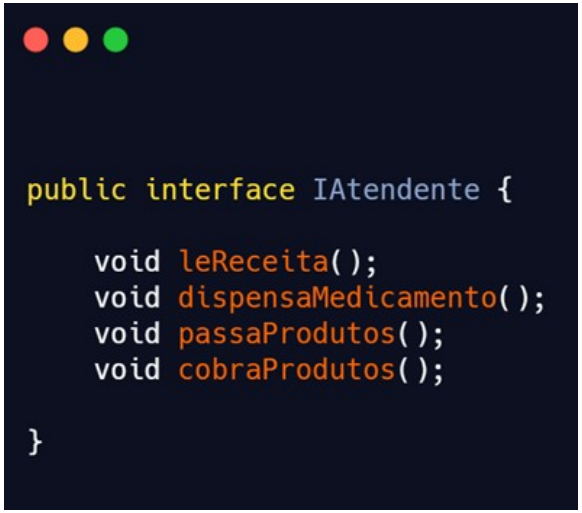
- Interface só deve ter métodos que são aplicáveis a todas as classes filhas.
- Se uma interface contém um método aplicável a algumas classes filhas, precisamos forçar o resto a fornecer implementação fictícia. Mova esses métodos para uma nova interface.

Princípio da Segregação da Interface





Princípio da Segregação da Interface



```
public interface IAtendente {

    void leReceita();
    void dispensaMedicamento();
    void passaProdutos();
    void cobraProdutos();

}
```

Classe AtendenteCaixa

Classe Balconista

Quando você implementasse essa interface na classe AtendenteCaixa, você teria que implementar todos os métodos, inclusive os métodos que o atendente de caixa não executa. Para evitar isso, basta criar uma interface que atenda separadamente as funções de cada atendente.

Princípio da Segregação da Interface

Atendente de Farmácia

- Passa no caixa
- Cobra Produtos
- Lê receita
- Dispensa medicamento

```
interface Funcionario {

   double getSalario();

   double getFGTS();// apenas funcionários CLT

   int getSIAPE();// apenas funcionários públicos

   ...
}
```

Interface genérica: trata de funcionários CLT e de funcionários públicos

O que "getSIAPE" retorna para funcionários CLT?

Agora versão que atende **segregação de interfaces**

```
interface Funcionario {
  double getSalario();
  ...
}

interface FuncionarioCLT extends Funcionario {
  double getFGTS();
  ...
}

interface FuncionarioPublico extends Funcionario {
  int getSIAPE();
  ...
}
```

```
interface Funcionario {
  double getSalario();
  ...
}
```

Comum para todos funcionários

```
interface FuncionarioCLT extends Funcionario {
  double getFGTS();
  ...
}

interface FuncionarioPublico extends Funcionario {
  int getSIAPE();
  ...
}
```

```
interface Funcionario {
   double getSalario();
   ...
}
```

```
interface FuncionarioCLT extends Funcionario {
   double getFGTS();
   ...
}
```

Específica para funcionários CLT

```
interface FuncionarioPublico extends Funcionario {
   int getSIAPE();
   ...
}
```

```
interface Funcionario {
  double getSalario();
  ...
}

interface FuncionarioCLT extends Funcionario {
  double getFGTS();
  ...
}

interface FuncionarioPublico extends Funcionario {
  int getSIAPE();
  ...
}
```

Específica para funcionários públicos

Princípio da Segregação da Interface

- Outro Exemplo:

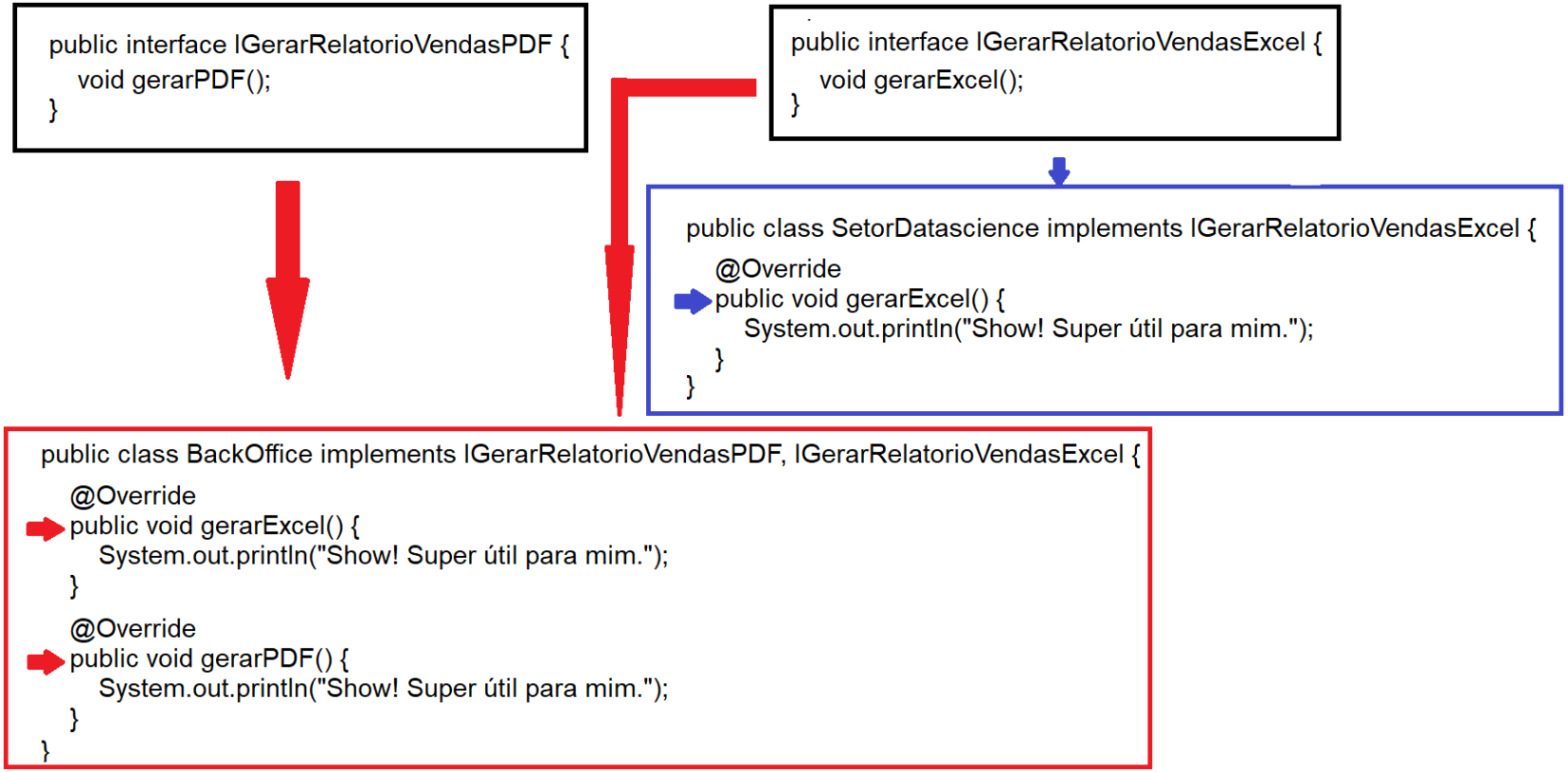
```
public interface IGerarRelatorioVendas {

    void gerarExcel();

    void gerarPDF();

}
```

```
public class BackOffice implements IGerarRelatorioVendas {

    @Override
    public void gerarExcel() {
        System.out.println("Show! Super útil para mim");
    }

    @Override
    public void gerarPDF() {
        System.out.println("Show! Super útil para mim");
    }

}
```

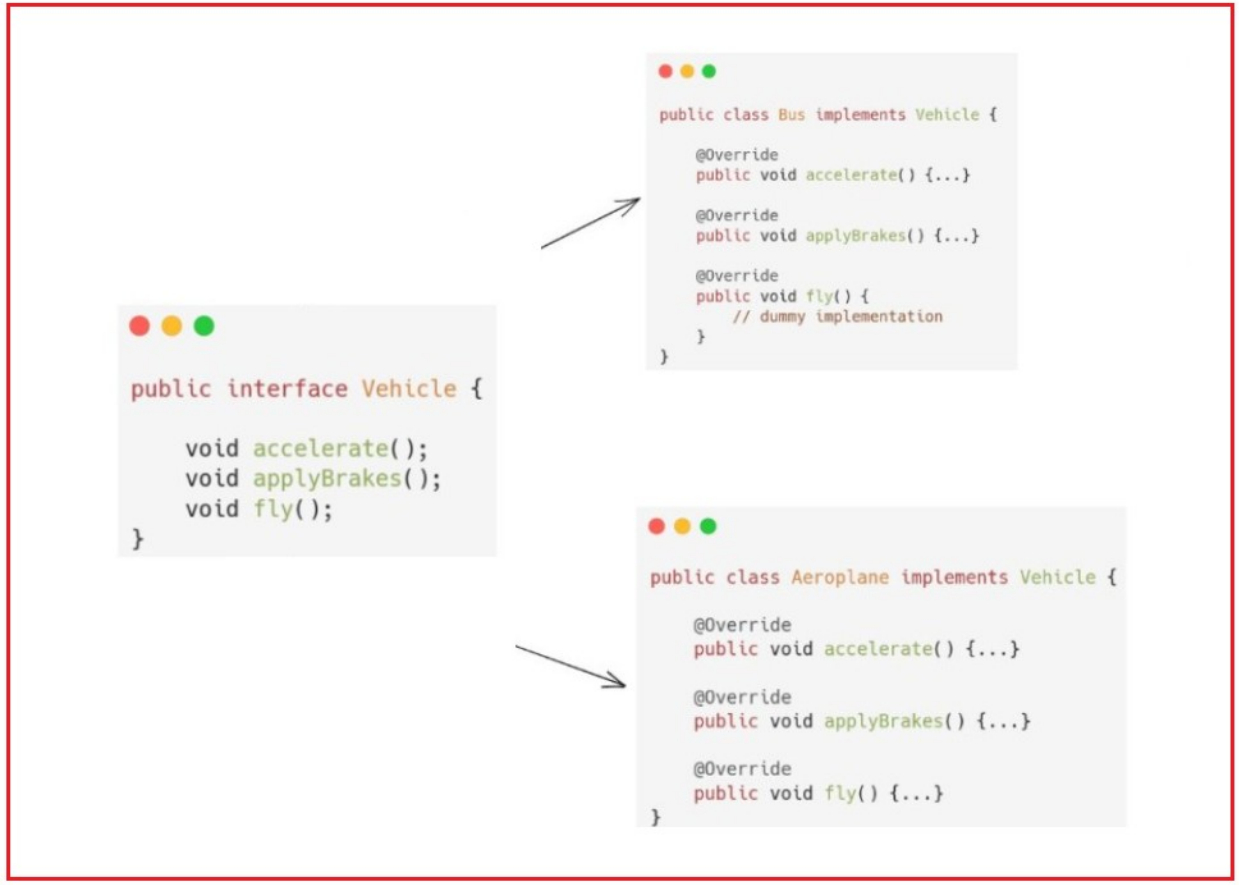
```
public class SetorDatascience implements IGerarRelatorioVendas {

    @Override
    public void gerarExcel() {
        System.out.println("Show! Super útil para mim.");
    }

    @Override
    public void gerarPDF() {
        System.out.println("Não, isso é muito inútil, nunca vou usar!");
    }

}
```

Princípio da Segregação da Interface

- Solução



Exercício: Segregação de Interface



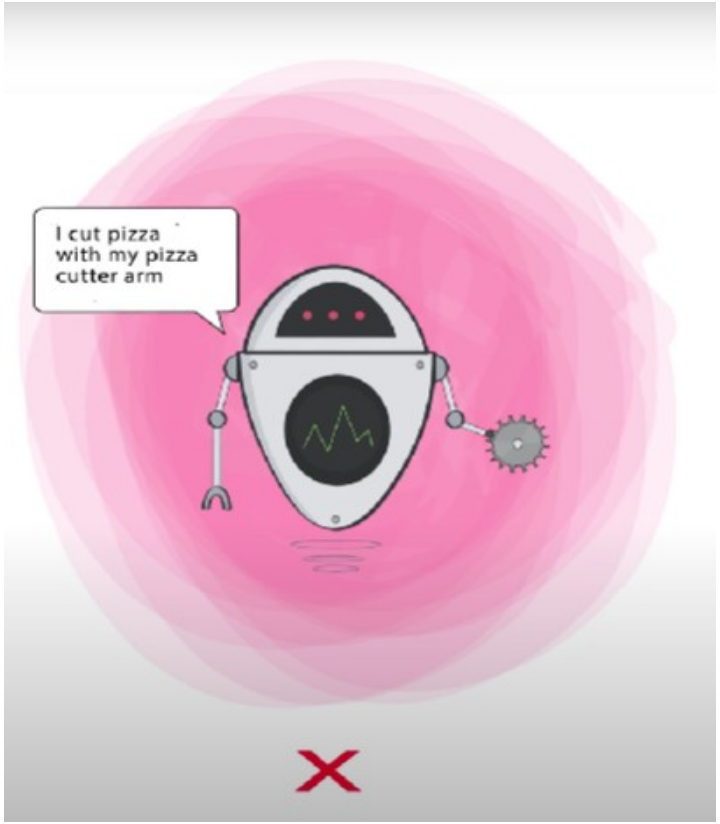
Princípio da Inversão de Dependência

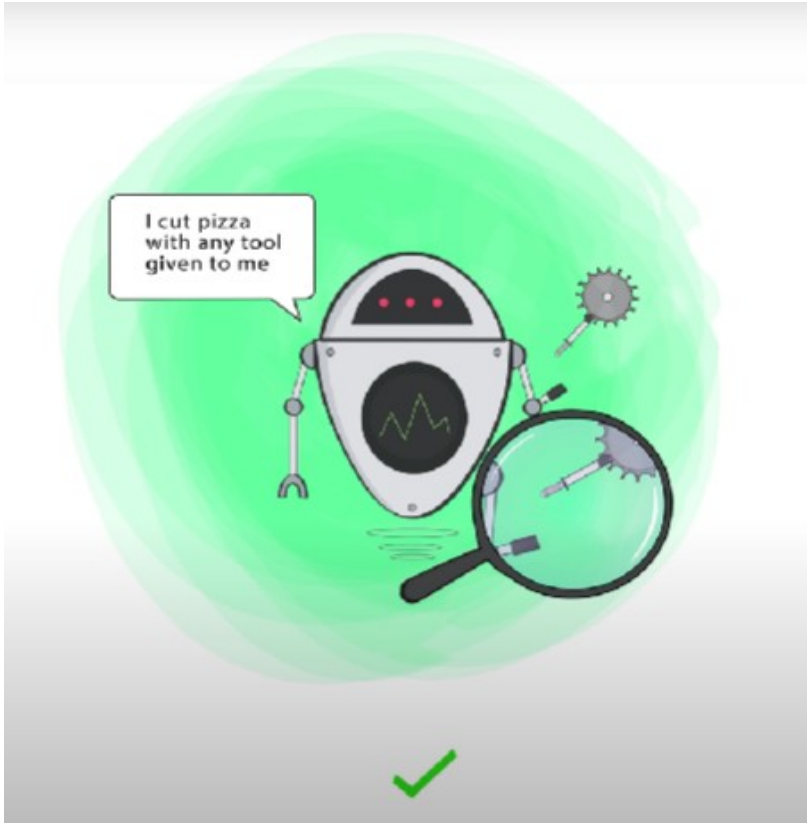
- Na verdade, vamos chamar esse princípio de "**Prefira Interfaces a Classes**"
- Pois transmite melhor a sua ideia!

Princípio da Inversão de Dependência

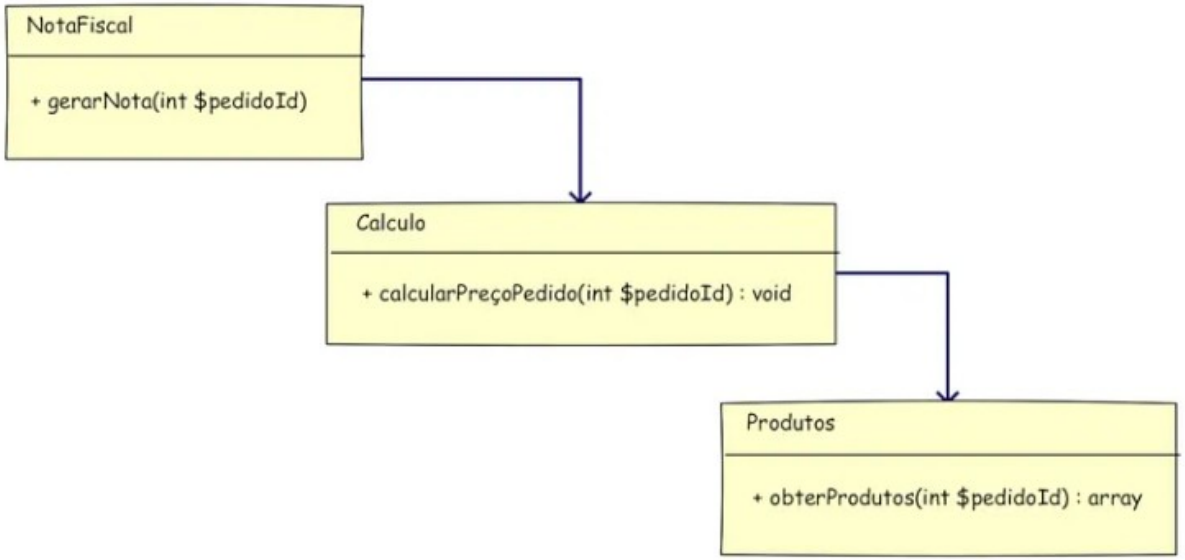
- A classe deve depender de abstrações (interface e classe abstrata) em vez de implementações concretas.
- Isso faz com que nossas classes se desacoplem umas das outras.
- Se a implementação mudar, a classe que se refere a ela via abstração não será alterada.

Princípio da Inversão de Dependência



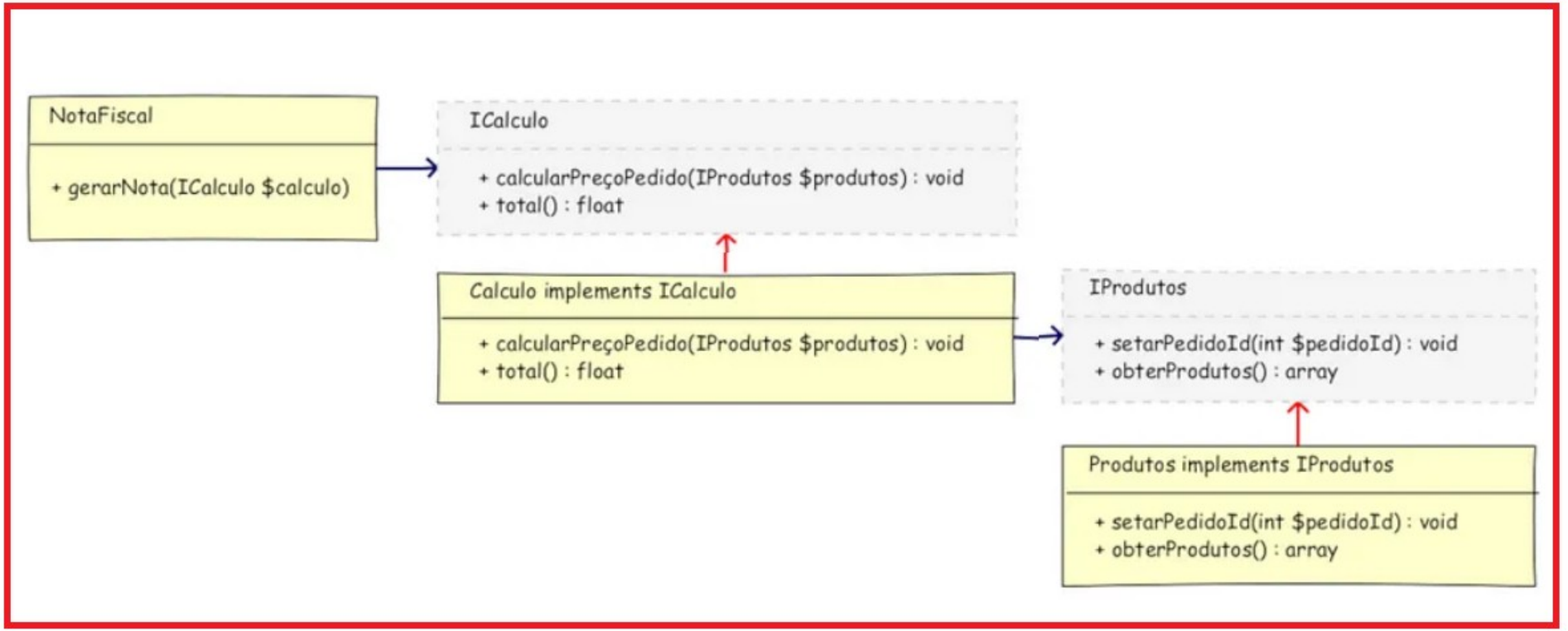


Em Código...



No exemplo, as dependências seguem uma sequência, onde o fluxo começa com a função de **alto nível** (*gerarNotaFiscal*), que depende de uma função de **baixo nível** (*calcular*) que, por sua vez, depende de um **detalhe** (*obterProdutos*).

Corrigindo...


NotaFiscal
+ gerarNota(ICalculo $calculo)
ICalculo
+ calcularPreçoPedido(IProdutos $produtos) : void
+ total() : float
Calculo implements ICalculo
+ calcularPreçoPedido(IProdutos $produtos) : void
+ total() : float
IProdutos
+ setarPedidoId(int $pedidoId) : void
+ obterProdutos() : array
Produtos implements IProdutos
+ setarPedidoId(int $pedidoId) : void
+ obterProdutos() : array

Exercício: Inversão Dependência

```
class SQLRepository{
  public void save() {...}
}

class NoSQLRepository{
    public void save() {...}
}

public class Service {

    //Here we've hard-coded SQLRepository
    //in-future if we need to support NoSQLRepository
    //then we need to modify our code

    private SQLRepository repository = new SQLRepository();

    public void save() {
        repository.save();
    }
}
```

Exercício

```
1  public class ClienteController {
2      public void save(ClienteVO c)
3      {
4          if (this.validate(c))
5          {
6              ClienteDAO.getInstance().save(c);
7              this.sendEmail();
8          }
9      }
10     private boolean validate(ClienteVO c) {
11         if (c.nome.equals(""))
12             return true;
13     }
14     private void sendEmail()
15     {
16         MimeMessage msg = new MimeMessage(session);
17         msg.setFrom(new InternetAddress("email@email.com"));
18         ...
19         Transport.send(msg);
20     }
21 }
```

Definir qual é o princípio de projeto que Está sendo violado no código abaixo.