Disciplina: Boas Práticas de Programação

Gibeon Aquino

DIMAp/UFRN

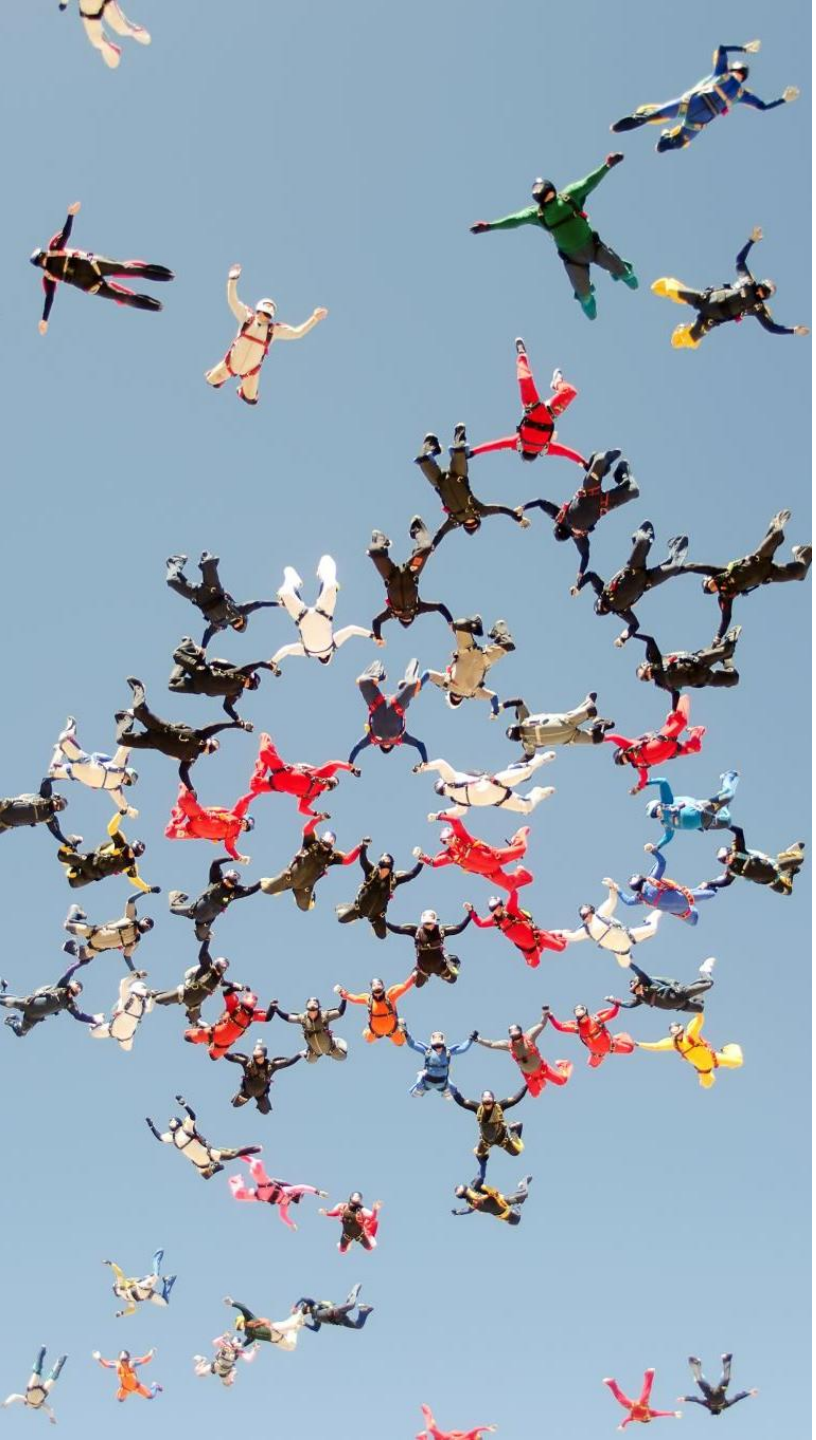
Princípios SOLID

Antes vamos revisar princípios OO

Coesão

- Quanto os elementos (métodos e atributos) de uma classe estão relacionados e trabalham juntos de maneira coesa para realizar uma única responsabilidade

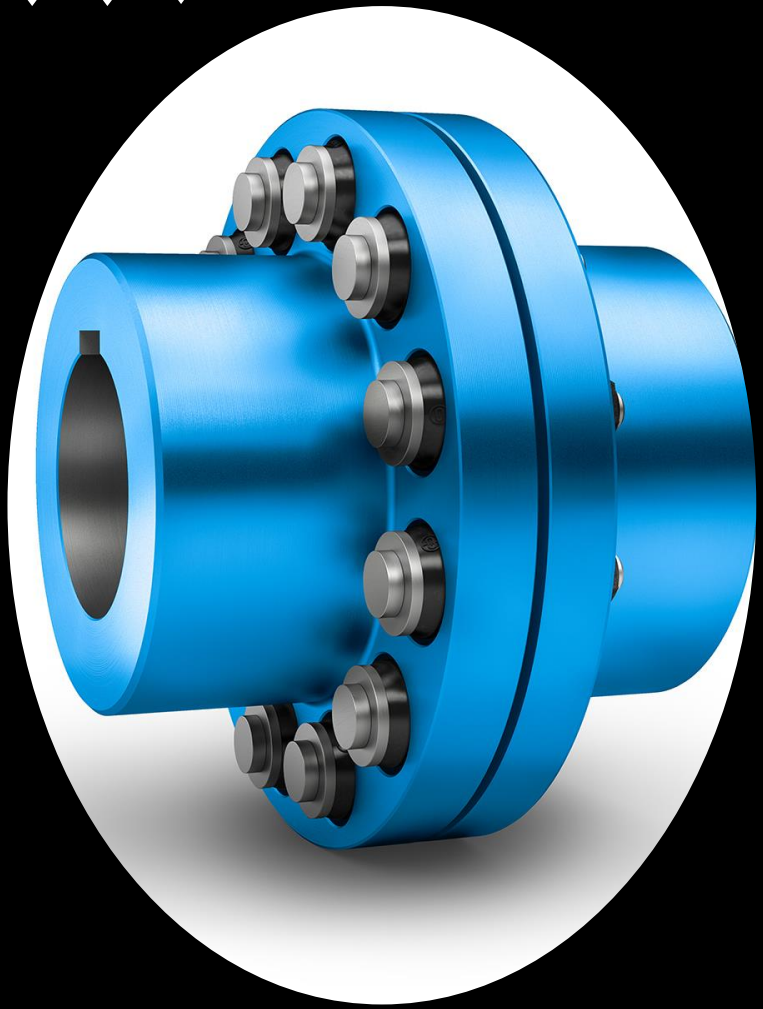
- Indica o grau de foco e propósito de uma classe



Coesão

- O objetivo é alcançar alta coesão, ou seja, que os elementos de uma classe estejam fortemente relacionados à sua finalidade principal

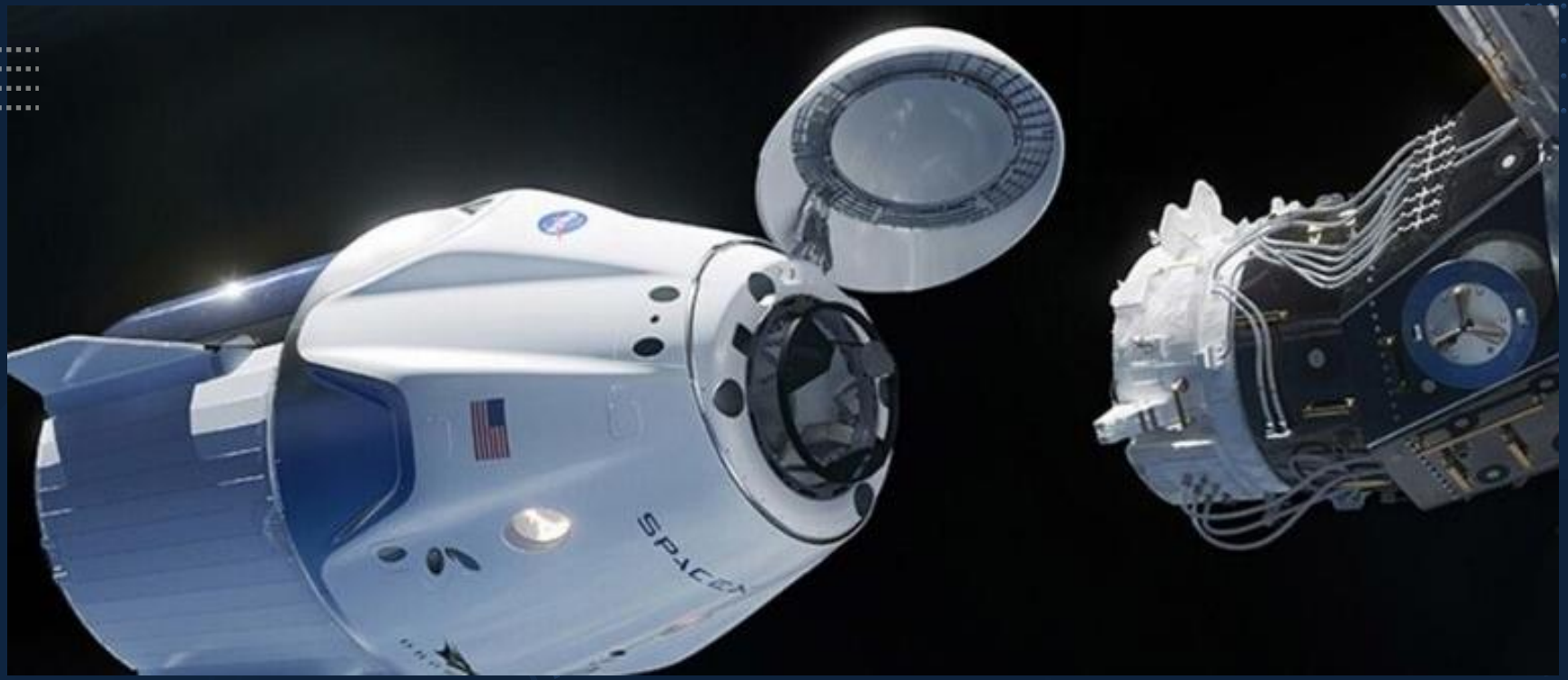
- Classes com baixa coesão tendem a ser confusas, difíceis de entender e propensas a efeitos colaterais inesperados.



Acoplamento

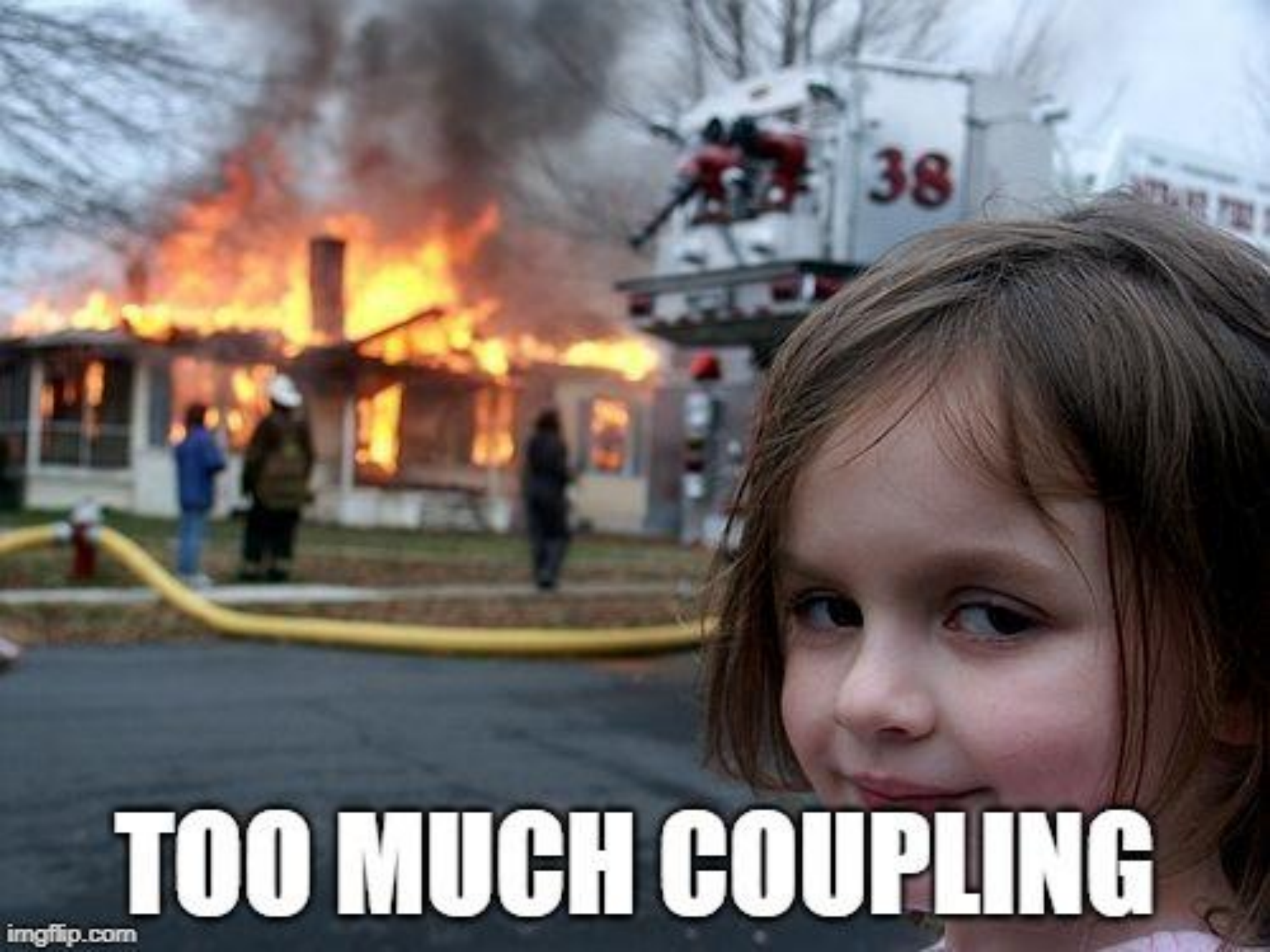
- Representa o quanto duas classes estão interconectadas ou dependentes uma da outra

- Refere-se à maneira como as classes se relacionam umas com as outras



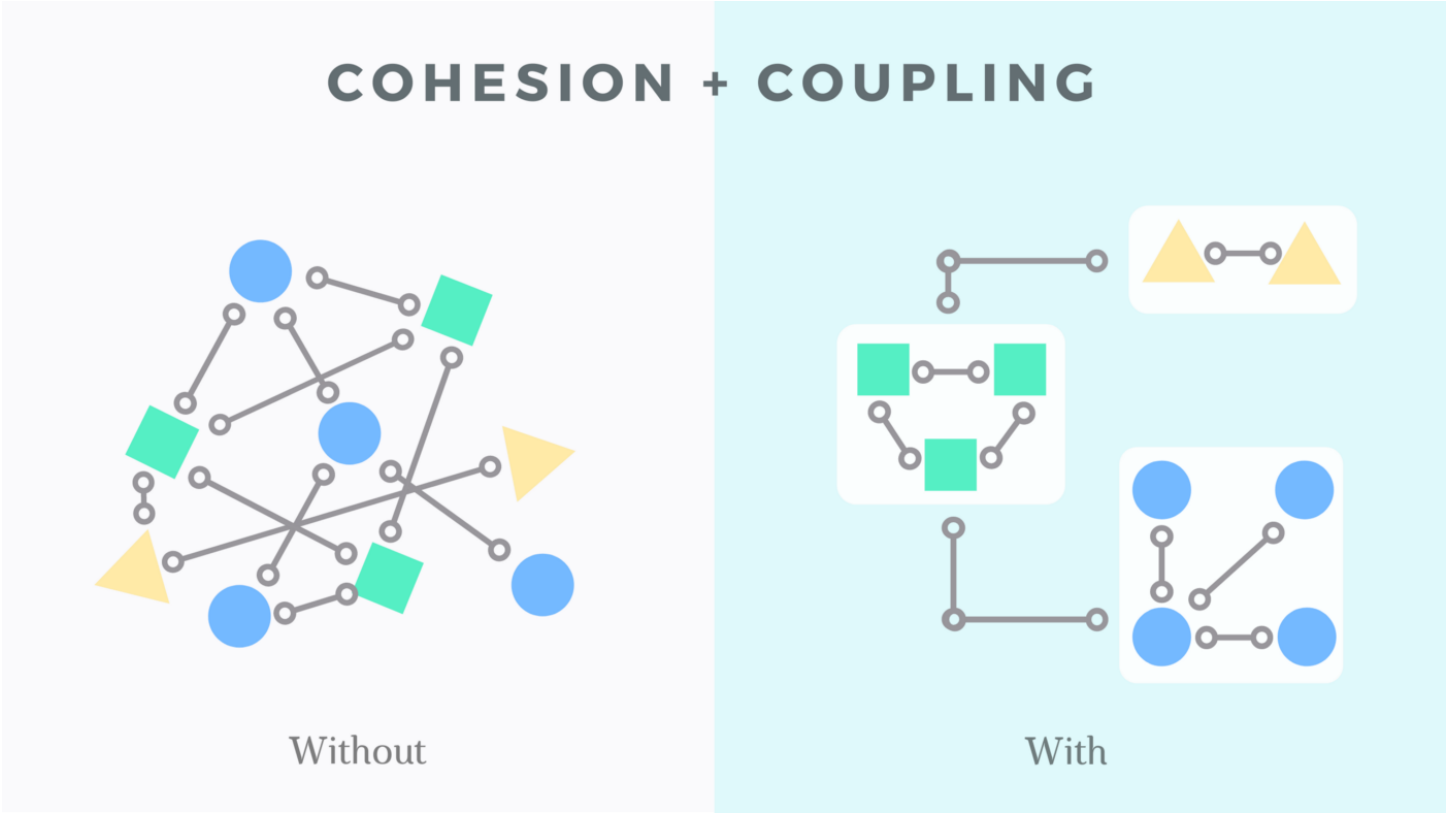
Acoplamento

- Quando há acoplamento entre classes, mudança em uma classe pode impactar nas outras
- É importante procurar reduzir o acoplamento entre as classes tanto quanto possível, visando criar um código mais modular, flexível e resiliente às mudanças


38
TOO MUCH COUPLING
imgflip.com

[Acoplamento e Coesão (2 "palavrões" essenciais na programação) // Dicionário do Programador - YouTube]

Em resumo...




S.O.L.I.D

SOLID é um acrônimo para cinco princípios da programação orientada a objetos sugeridos por Robert C. Martin (ou Uncle Bob)

http://cleancoder.com/

Single Responsiblity Principle

“A class should have one, and only one, reason to change”

Single Responsiblity Principle

- Uma classe deve ter uma única responsabilidade e esta responsabilidade deve ser inteiramente encapsulada pela classe

- Todos os seus serviços devem estar estreitamente alinhados com essa responsabilidade

Single Responsiblity Principle

Isso promove a coesão, tornando as classes mais focadas e facilitando a manutenção.

```
public class Employee
{
    public double CalculatePay(Money money)
    {
        //business logic for payment here
    }

    public Employee Save(Employee employee)
    {
        //store employee here
    }
}
```

```
public class Employee
{
    public double CalculatePay(Money money)
    {
        //business logic for payment here
    }
}

public class EmployeeRepository
{
    public Employee Save(Employee employee)
    {
        //store employee here
    }
}
```

```php
<?php

class Order
{
    public function calculateTotalSum(){/*...*/}
    public function getItems(){/*...*/}
    public function getItemCount(){/*...*/}
    public function addItem($item){/*...*/}
    public function deleteItem($item){/*...*/}

    public function printOrder(){/*...*/}
    public function showOrder(){/*...*/}

    public function load(){/*...*/}
    public function save(){/*...*/}
    public function update(){/*...*/}
    public function delete(){/*...*/}
}

// Reference: https://www.apphp.com/tutorials/index.php?page=solid-principles-in-php-examples
```

```php
<?php

class Order
{
    public function calculateTotalSum(){/*...*/}
    public function getItems(){/*...*/}
    public function getItemCount(){/*...*/}
    public function addItem($item){/*...*/}
    public function deleteItem($item){/*...*/}
}

class OrderRepository
{
    public function load($orderID){/*...*/}
    public function save($order){/*...*/}
    public function update($order){/*...*/}
    public function delete($order){/*...*/}
}

class OrderViewer
{
    public function printOrder($order){/*...*/}
    public function showOrder($order){/*...*/}
}

//Reference: https://www.apphp.com/tutorials/index.php?page=solid-principles-in-php-examples
```

srp-solid-refactor.php hosted with ❤ by GitHub view raw

SRP em resumo...

- Promove uma melhor organização, manutenção mais fácil, maior reutilização e facilita a evolução do código ao garantir que cada classe tenha uma única responsabilidade bem definida

Open-Closed Principle

“software entities (classes, methods, modules, etc.) should be open for extension but closed for modification.”

Open-Closed Principle (OCP)

- Modificação significa **alterar** o código de uma classe existente
- Extensão significa **adicionar** novas funcionalidades
- OCP =>
  - **Aberto** para **Adicionar**
  - **Fechado** para **Alterar**

```
public enum PaymentType = { Cash, CreditCard };

public class PaymentManager
{
    public PaymentType PaymentType { get; set; }

    public void Pay(Money money)
    {
        if(PaymentType == PaymentType.Cash)
        {
            //some code here - pay with cash
        }
        else
        {
            //some code here - pay with credit card
        }
    }
}
```

Como adicionar o pagamento do tipo PIX?

```
public class Payment
{
    public virtual void Pay(Money money)
    {
        // from base
    }
}
```

Como adicionar o pagamento do tipo PIX?

```
public class CashPayment : Payment
{
    public override void Pay(Money money)
    {
        //some code here - pay with cash
    }
}
```

```
public class CreditCardPayment : Payment
{
    public override void Pay(Money money)
    {
        //some code here - pay with credit card
    }
}
```

```
1   public enum TipoEmail {
2           Texto,
3           Html,
4           Criptografado
5   }
6
7   public void class EnviarEmail(string mensagem, string assunto, TipoEmail tipo){
8           if(tipo == TipoEmail.Texto)
9           {
10                  mensagem = this.RemoverFormatacao(mensagem);
11          }
12          else if(tipo == TipoEmail.Html)
13          {
14                  mensagem = this.InserirHtml(mensagem);
15          }
16          else if(tipo == TipoEmail.Criptografado)
17          {
18                  mensagem = this.CriptografarMensagem(mensagem);
19          }
20
21          this.EnviarMensagem();
22  }
```

OCP Class Violation hosted with ❤ by GitHub view raw

https://medium.com/@tbaragao/solid-ocp-open-closed-principle-600be0382244

```
1   public abstract class Email
2   {
3           public abstract void Enviar(string assunto, string mensagem);
4   }
5
6   public class TextoEmail : Email
7   {
8           public override void Enviar(string assunto, string mensagem)
9           {
10                  Util.RemoverFormatacao(mensagem);
11          }
12  }
13
14  public class HtmlEmail : Email
15  {
16          public override void Enviar(string assunto, string mensagem)
17          {
18                  Util.InserirHtml(mensagem);
19          }
20  }
21
22  public class CriptografadoEmail : Email
23  {
24          public override void Enviar(string assunto, string mensagem)
25          {
26                  Util.CriptografarMensagem(mensagem);;
27          }
28  }
```

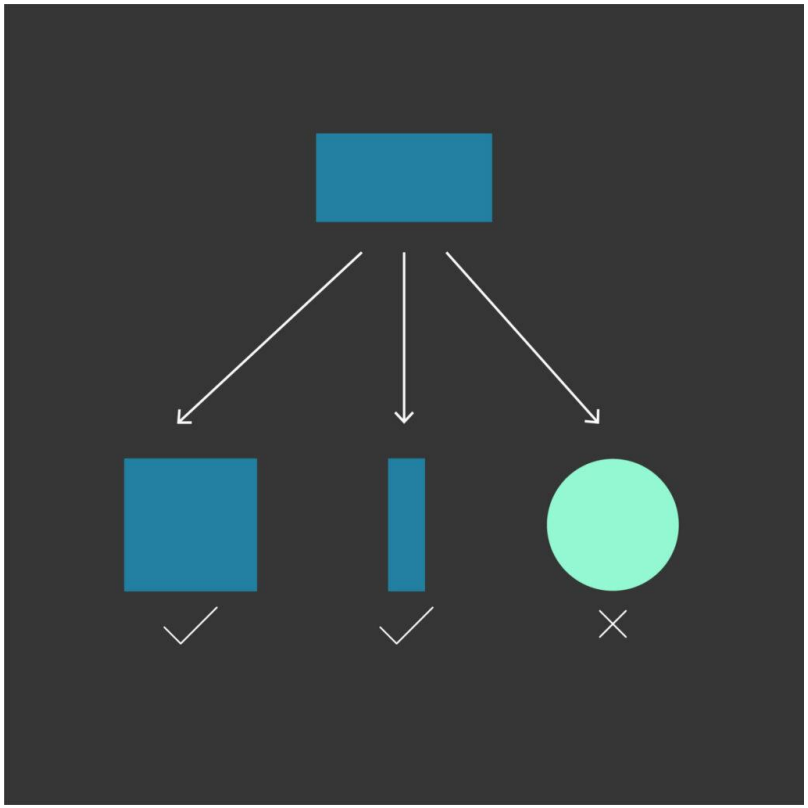
OCP Class Solution hosted with ❤ by GitHub view raw

OCP em resumo...

“Você deve ser capaz de estender um comportamento de uma classe sem a necessidade de modificá-lo.”

Liskov Substitution Principle

"Subtypes must be substitutable for their base types."



Liskov Substitution Principle

- Introduzido por Barbara Liskov, 1987, como:

> *"Em um programa orientado a objetos bem escrito, os objetos de uma classe base devem ser substituídos por objetos de uma classe derivada sem afetar a corretude do programa."*

Liskov Substitution Principle

- Ajuda a garantir que a hierarquia de herança seja bem projetada e que as subclasses não violem os contratos estabelecidos pelas suas classes base

- Promove tipagem forte e o polimorfismo de uma forma que aumenta a robustez e a manutenibilidade do código
  - A violação do LSP pode levar a bugs inesperados e software frágil.

```java
class Bird {
    void fly() {
        // Base class fly method, which does nothing by default
    }
}

class Ostrich extends Bird {
    @Override
    void fly() {
        // Ostriches cannot fly, so this method is overridden
        throw new UnsupportedOperationException("Ostriches can't fly");
    }
}

public class LSPViolationExample {
    static void makeBirdFly(Bird bird) {
        bird.fly();
    }

    public static void main(String[] args) {
        Bird bird = new Bird();
        Ostrich ostrich = new Ostrich();

        makeBirdFly(bird);   // No problem, a Bird can fly
        makeBirdFly(ostrich); // Exception! Ostriches can't fly
    }
}
```

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

    public Retangulo() {
    }

    public Retangulo(int largura, int altura) {
        this.largura = largura;
        this.altura = altura;
    }

    public int getLargura() {
        return largura;
    }

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

    public int getAltura() {
        return altura;
    }

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

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

```java
class Quadrado extends Retangulo {
    public Quadrado() {}

    public Quadrado(int tamanho) {
        largura = altura = tamanho;
    }

    @Override
    public void setLargura(int largura) {
        super.setLargura(largura);
        super.setAltura(largura);
    }

    @Override
    public void setAltura(int altura) {
        super.setAltura(altura);
        super.setLargura(altura);
    }
}
```


?


?

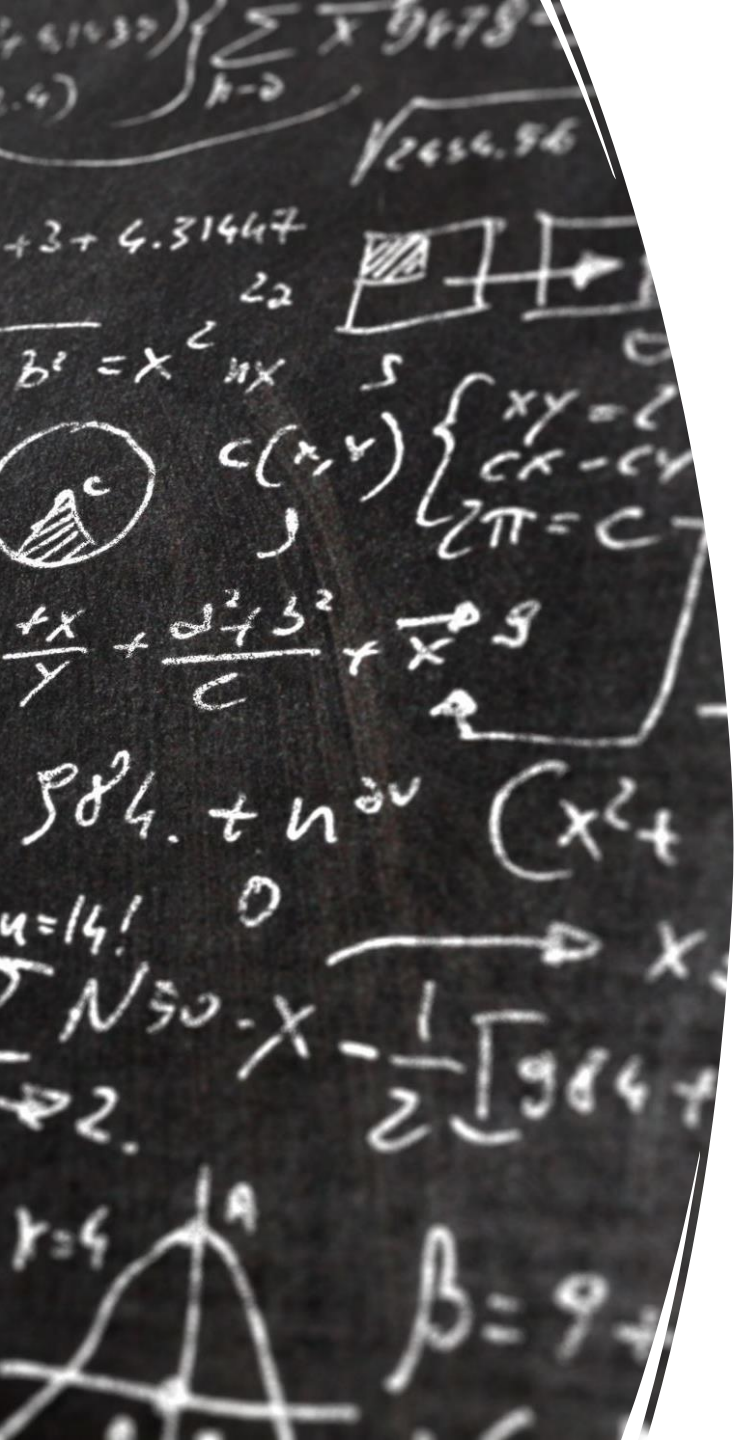
```java
class Test {

    static void esticarVertical(Retangulo r, int incremento) {

        int novaLargura = r.getLargura() + incremento;

        r.setLargura(novaLargura);
    }

    public static void main(String[] args) {
        Retangulo rc = new Retangulo();
        Retangulo sq = new Quadrado();

        rc.setLargura(5);
        rc.setAltura(10);

        sq.setLargura(5);

        ...

        esticarVertical(rc,15);

        esticarVertical(sq,15); //  :(

    }
}
```

LSP em resumo...

- Busca garantir que as classes derivadas sejam verdadeiras extensões de suas classes base, preservando o comportamento e os contratos pré-estabelecidos e não introduzindo efeitos colaterais inesperados

Interface Segregation Principle

"Clients should not be forced to depend on interfaces they don't use."



Interface Segregation Principle

- Classes ou objetos não devem ser obrigados a implementar métodos de que não precisam

- Defende a divisão de interfaces grandes e monolíticas em interfaces menores e mais específicas, cada uma adaptada às necessidades específicas de um cliente

```java
public interface Estacionamento {

    void estacionarCarro(); // Diminuir contagem de vagas em 1
    void sairDaVagaComCarro(); // Aumentar contagem de vagas em 1
    void getCapacidade();   // Retornar capacidade de carros
    double calcularTaxa(Carro carro); // Retornar o preço com base no número de horas
    void pagar(Carro carro);
}

class Carro {

}
```

```java
public class EstacionamentoGratuito implements Estacionamento {

    @Override
    public void estacionarCarro() {

    }

    @Override
    public void sairDaVagaComCarro() {

    }

    @Override
    public void getCapacidade() {

    }

    @Override
    public double calcularTaxa(Carro carro) {
        return 0;
    }

    @Override
    public void pagar(Carro carro) {
        throw new Exception("Estacionamento gratuito");
    }
}
```

```java
class Manager implements Worker {
    @Override
    public void work() {
        // Manager-specific work implementation
    }

    @Override
    public void eat() {
        // Manager-specific eating behavior
    }
}

class Janitor implements Worker {
    @Override
    public void work() {
        // Janitor-specific work implementation
    }

    @Override
    public void eat() {
        // Janitor-specific eating behavior
    }
}
```

```java
interface Worker {
    void work();
    void eat();
}
```

```java
class Robot implements Worker {
    @Override
    public void work() {
        // Robot-specific work implementation
    }

    @Override
    public void eat() {
        // Oops, robots don't eat! This method doesn't make sense
    }
}
```

```java
// Separate interface for work-related behavior
interface Workable {
    void work();
}

// Separate interface for eating-related behavior
interface Eatable {
    void eat();
}

// Manager class implements Workable and Eatable
class Manager implements Workable, Eatable {
    @Override
    public void work() {
        // Manager-specific work implementation
    }

    @Override
    public void eat() {
        // Manager-specific eating behavior
    }
}
```

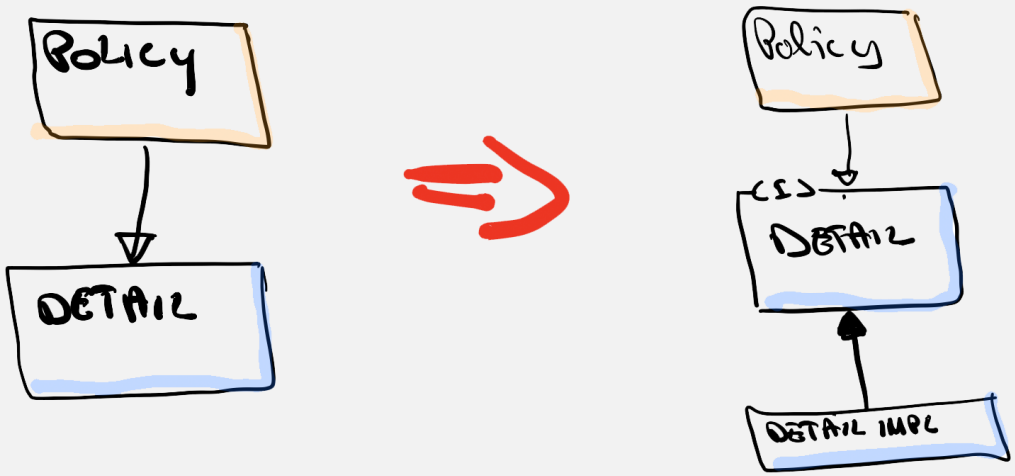
```java
// Janitor class implements only Workable
class Janitor implements Workable {
    @Override
    public void work() {
        // Janitor-specific work implementation
    }
}

// Robot class implements only Workable
class Robot implements Workable {
    @Override
    public void work() {
        // Robot-specific work implementation
    }
}
```

ISP em Resumo

- Promove a criação de interfaces coesas e com foco restrito, o que leva a uma melhor organização do código, maior capacidade de manutenção e menor acoplamento entre classes

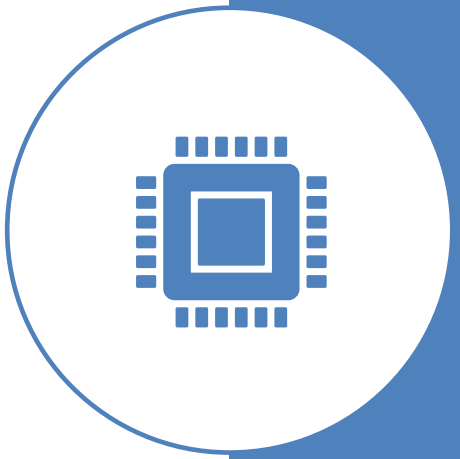
Dependency Inversion Principle

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



Dependency Inversion Principle

- Projete seu software de forma a minimizar dependências diretas entre diferentes partes de sua base de código
- Dependências devem ser baseadas em abstrações ou interfaces em vez de implementações concretas



```
class HighLevelModule {
    private DatabaseConnection database;

    public HighLevelModule() {
        this.database = new DatabaseConnection();
    }

    public void doSomethingWithDatabase() {
        // Use the database connection
    }
}
```

`HighLevelModule` cria diretamente uma instância de `DatabaseConnection`, estabelecendo uma forte dependência.

```
interface DatabaseConnection {
    void connect();
    void disconnect();
}

class HighLevelModule {
    private DatabaseConnection database;

    public HighLevelModule(DatabaseConnection database) {
        this.database = database; // Dependency is now on the abstraction,
    }

    public void doSomethingWithDatabase() {
        database.connect();
        // Perform operations on the database
        database.disconnect();
    }
}
```

```
class MySQLDatabaseConnection implements DatabaseConnection {
  public void connect() {...} // Connection to MySQL
  public void disconnect() {...}
}

class PostgreSQLDatabaseConnection implements DatabaseConnection {
  public void connect() {...} // Connection to PostgreSQL
  public void disconnect() {...}
}
```

DIP em resumo

- Ajuda a criar um código mais flexível e de fácil manutenção porque fica mais fácil alterar ou estender o sistema sem afetar outras partes.

- Permite melhor testabilidade, pois pode-se substituir dependências por implementações simuladas (“mockadas”)

Disciplina: Boas Práticas de Programação

Gibeon Aquino

DIMAp/UFRN