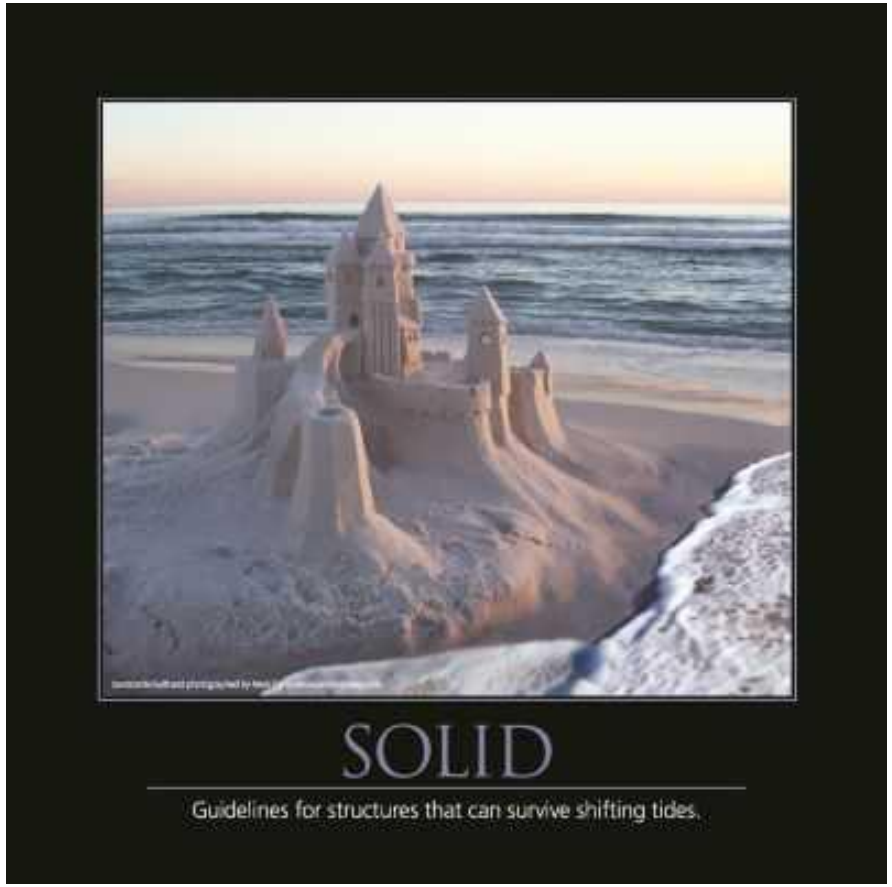
SOLID

Por que os princípios SOLID ainda são a base para a arquitetura de software moderna

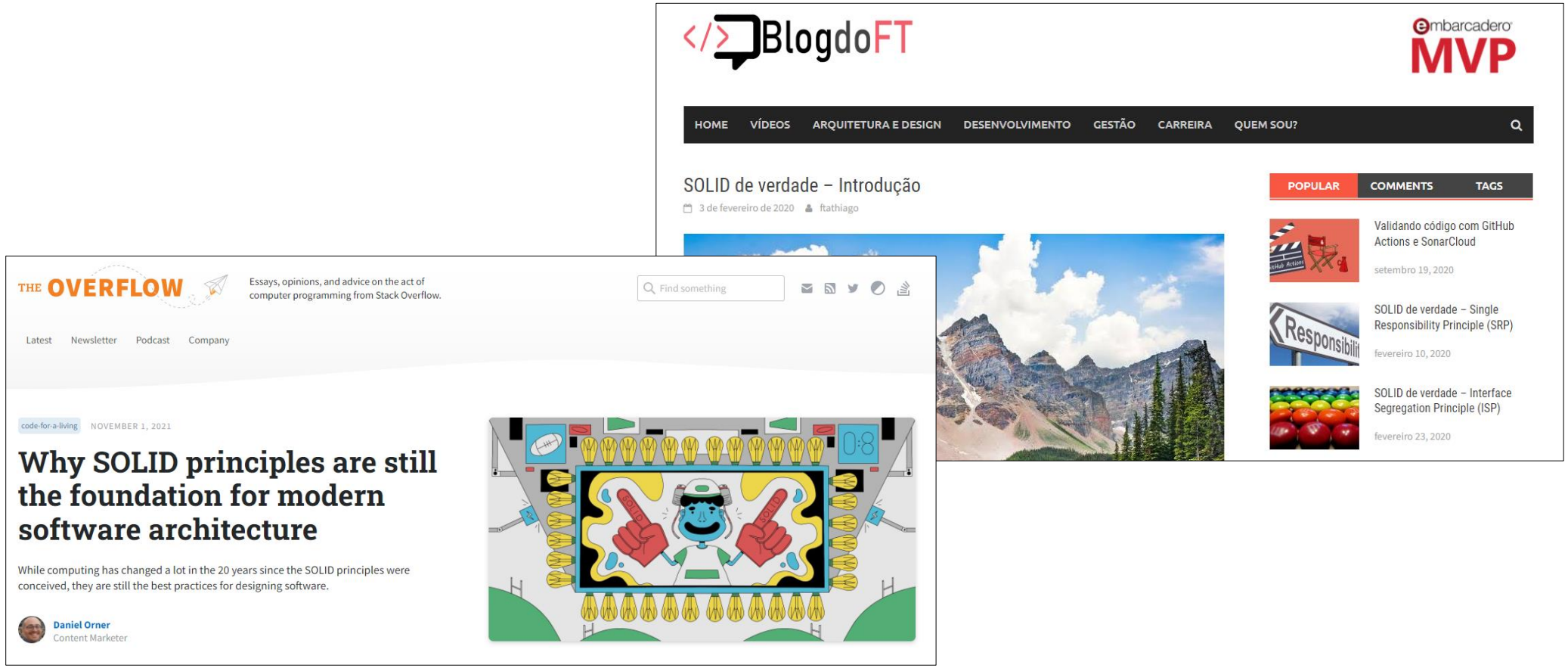
Denise Garcia Telli

LIGHTNING TALKS

13-12-2021

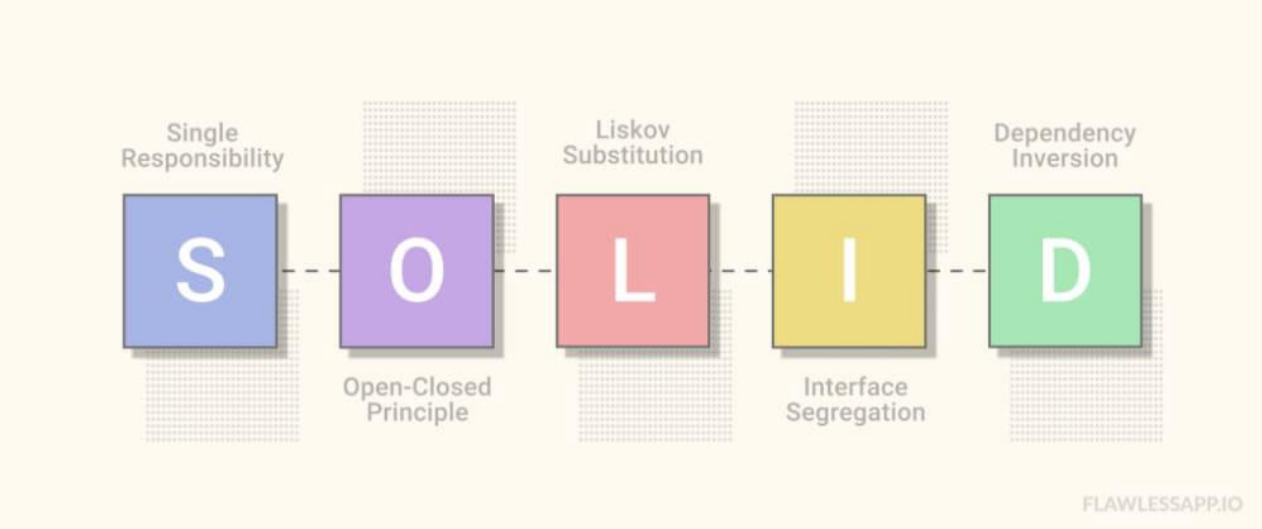


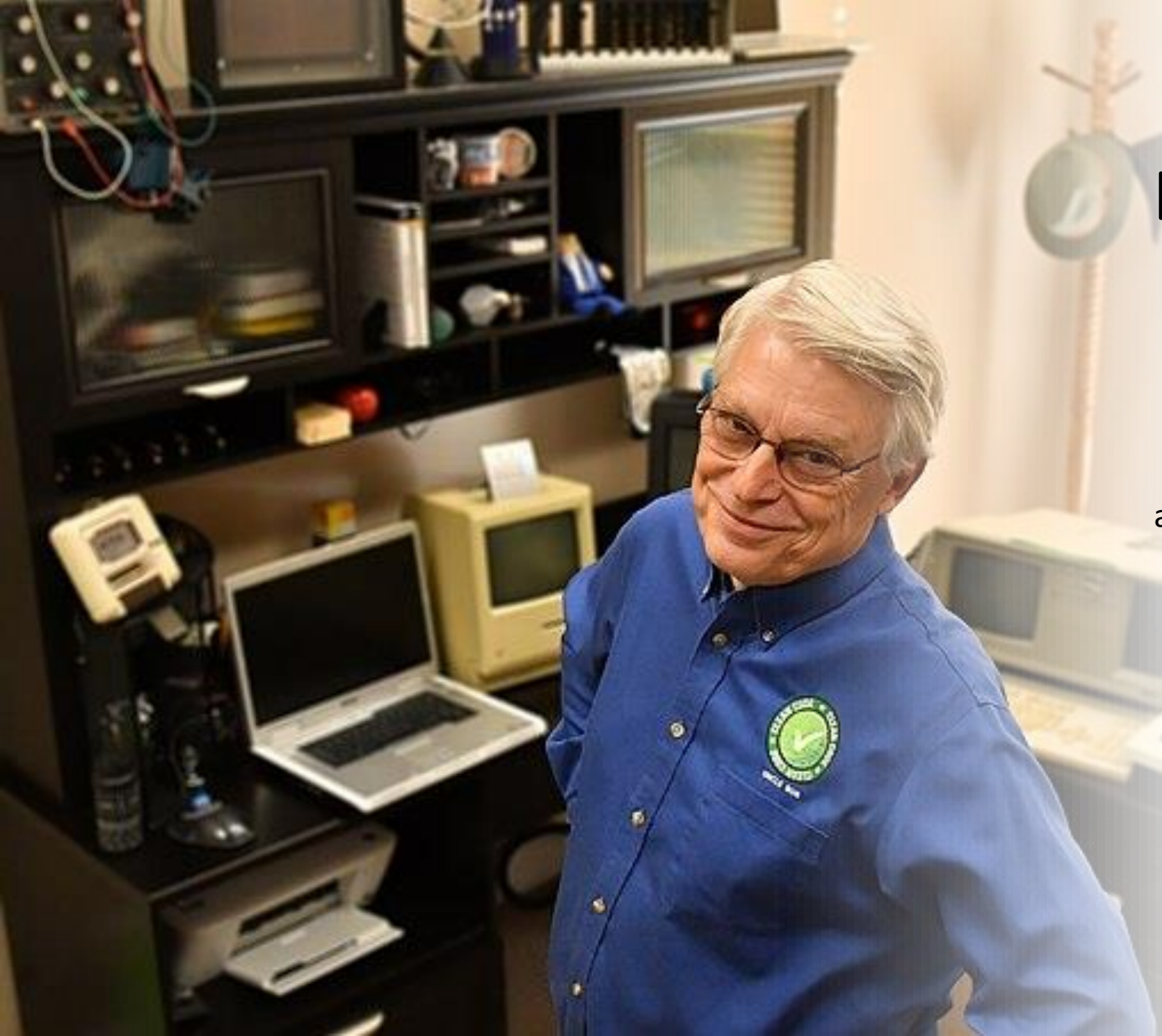
Artigos de Daniel Orner e Francisco Thiago



O que é?

Acrônimo mnemônico para cinco princípios de design destinados a **tornar os designs de software** mais **compreensíveis**, **flexíveis** e de **fácil manutenção**.





De onde surgiu?

Os princípios são um subconjunto de muitos princípios promovidos pelo pelo famoso Cientista da Computação Robert C. Martin (Uncle Bob), apresentados pela primeira vez em seu artigo de 2000 “Design Principles and Design Patterns”.

A sigla SOLID foi introduzida posteriormente por Michael Feathers.

Todos eles têm o mesmo propósito:

"*Para criar um código compreensível, legível e testável no qual muitos desenvolvedores possam trabalhar de forma colaborativa*."

O que mudou?

No início dos anos 2000, Java e C ++ eram os reis.

Desde então, as mudanças na indústria de software foram profundas. Alguns notáveis:

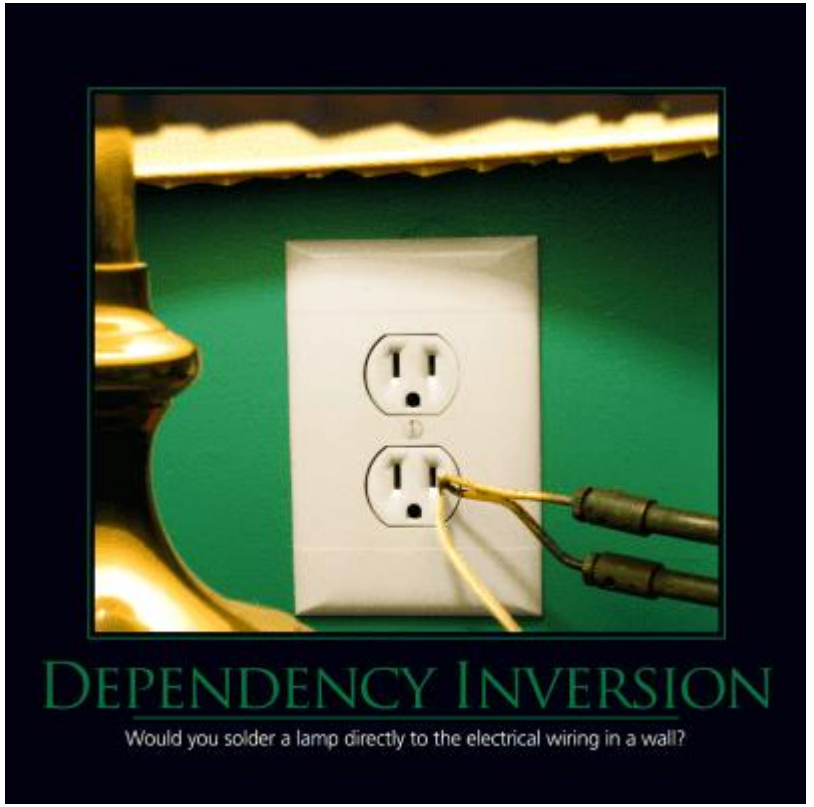
- **Linguagens com tipos dinâmicos.**
- **Paradigmas não orientados a objetos.**
- **O software de código aberto proliferou.**
- **Microserviços e SaaS explodiram.**

O que não mudou?

- **O código é escrito e modificado por pessoas.**
- **O código é organizado em módulos.**
- **O código pode ser interno ou externo.**

Princípio da Inversão da Dependência

Dependency Inversion Principle (DIP)



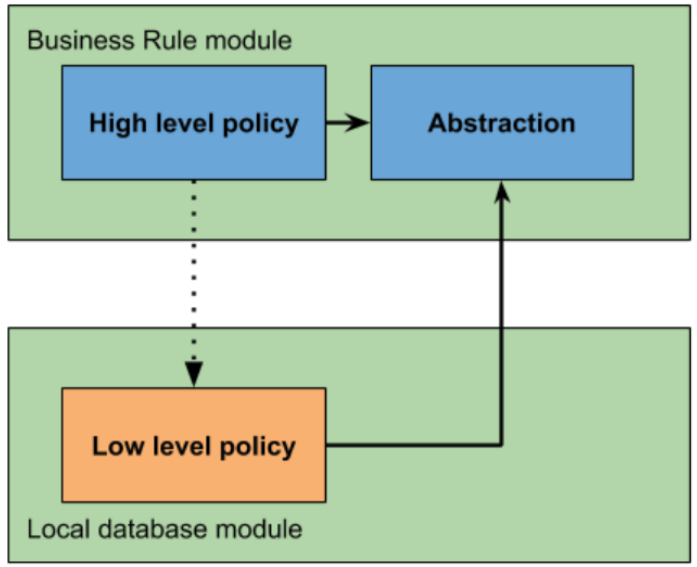
Princípio da Inversão da Dependência

Definição original:

"Dependa de abstrações, não de implementações".

Esse princípio pode ser definido da seguinte forma:

1. Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
2. Abstrações não devem depender de detalhes. Detalhes (implementações concretas) devem depender de abstrações.



Nova definição: "Dependa de abstrações, não de implementações".

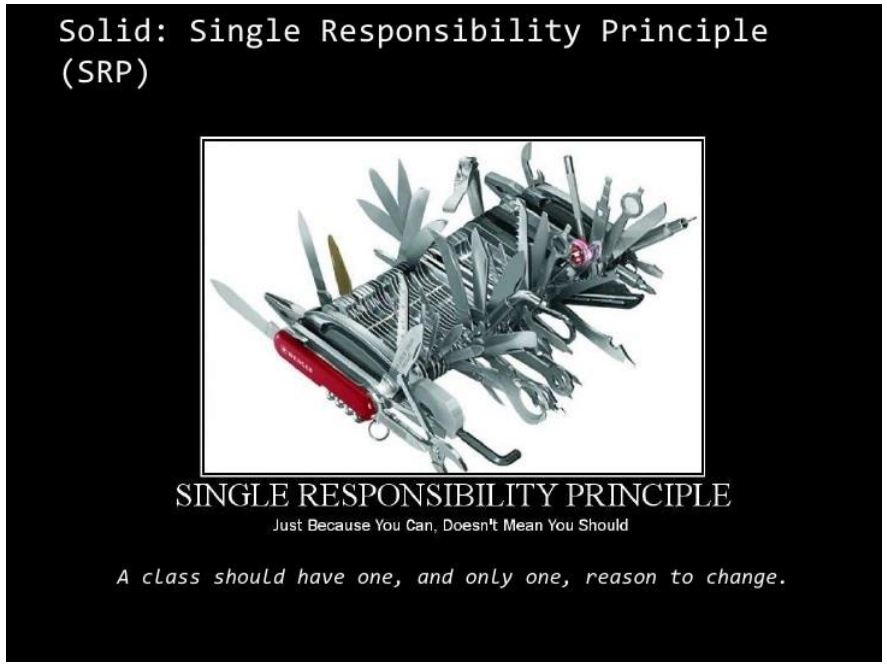
Princípio da Inversão da Dependência

```
class ImportantBusinessRule {

    private val localDatabase: LocalDatabase = LocalDatabase()

    fun doImportantStuff() {
        localDatabase.getInformation()
    }
}
```

```
class ImportantBusinessRules(private val dataProvider: DataProvider) {

    fun doImportantStuff() {
        dataProvider.getInformation()
    }
}
```

Princípio da Responsabilidade Única

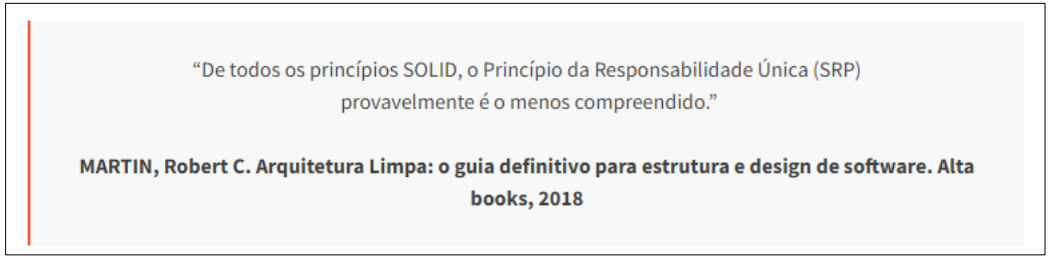
Single Responsibility Principle (SRP)



Princípio da Responsabilidade Única

Definição original: “Nunca deve haver mais de um motivo para uma classe mudar.”

Se você escreve uma classe com muitas interesses, ou “motivos para mudar”, então você precisa mudar o mesmo código sempre que qualquer um desses interesses tiver que mudar. Isso aumenta a probabilidade de que uma alteração em um recurso acidentalmente quebre um recurso diferente.



Princípio da Responsabilidade Única

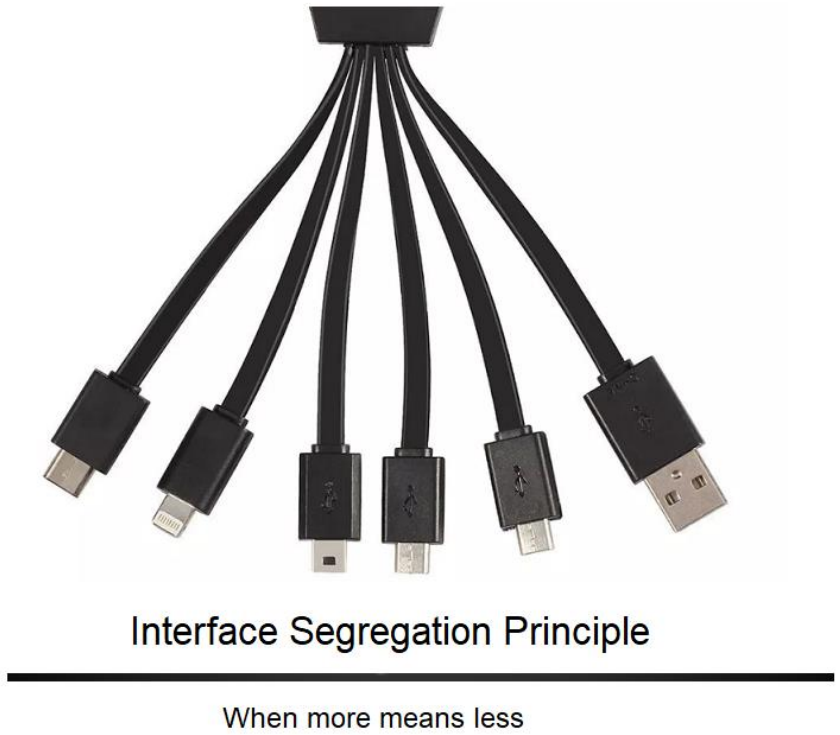
Nova definição: “Um módulo deve ser responsável por um, e apenas um, ator.”

> Não se confunda, há um princípio parecido com este. Uma função deve fazer uma, e somente uma, coisa. Nós usamos este princípio quando estamos refatorando funções grandes em menores; nós usamos isto no nível mais baixo. Mas este não é um dos princípios SOLID – isto não é SRP.
>
> **Clean Architecture – Traduzido livremente**

Este princípio está intimamente relacionado ao tópico de alta coesão. Essencialmente, seu código não deve misturar várias funções ou finalidades.

Princípio da Segregação das Interfaces

Interface Segregation Principle (ISP)



Princípio da Segregação das Interfaces

Definição original: “Muitas interfaces específicas do cliente são melhores do que uma interface de uso geral” ou “Nenhum cliente deveria ser forçado a depender de métodos que ele não usa”.

Nova definição: “Não mostre a seus clientes mais do que eles precisam ver”.

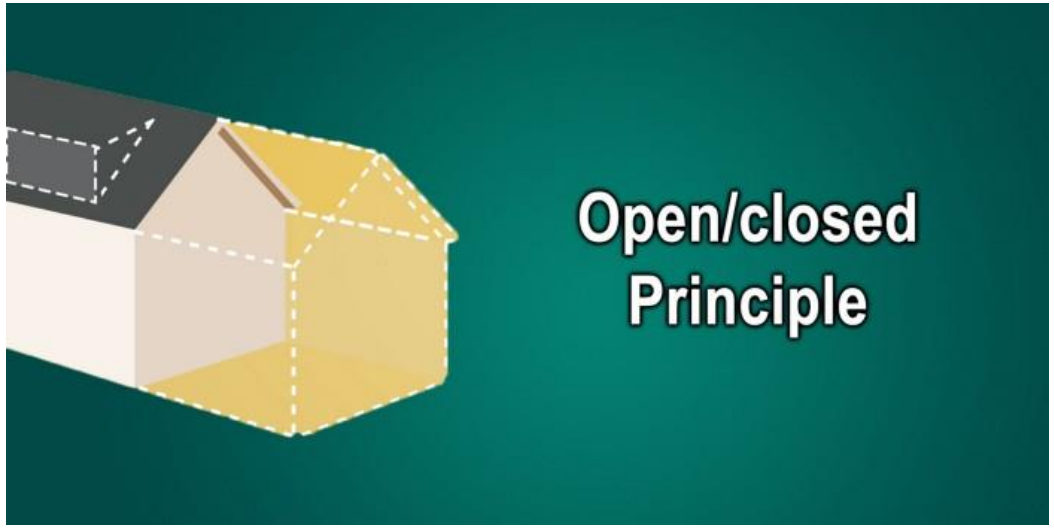
Princípio da Segregação das Interfaces

```
class PrintRequest {
   public void createRequest() {}
   public void deleteRequest() {}
   public void workOnRequest() {}
}
```

```
interface PrintRequestModifier {
   public void createRequest();
   public void deleteRequest();
}

interface PrintRequestWorker {
   public void workOnRequest()
}

class PrintRequest implements PrintRequestModifier, PrintRequestWorker {
   public void createRequest() {}
   public void deleteRequest() {}
   public void workOnRequest() {}
}
```

Princípio Aberto Fechado

The Open Closed Principle (OCP)



Princípio Aberto Fechado

Definição original: “Entidades de software devem ser abertas para extensão, mas fechadas para modificação.”

Nova definição: “Você deve ser capaz de usar e adicionar a um módulo sem reescrevê-lo.”

Princípio Aberto Fechado

```
fun listResponsibilities(employee: Employee): List<String> {
    return if (employee.role == EmployeeRole.DEV) {
        listOf(“code”, “code review”)
    } else if (employee.role == EmployeeRole.TEST) {
        listOf(“write test plan”, “execute test plan”)
    } else if (employee.role == EmployeeRole.UX) {
        listOf(“analyze usability”, “deliver UX specs”)
    }
}
```

```
abstract class Employee {
    abstract fun getResponsibilities(): List<String>
}

class SoftwareEngineer: Employee() {
    override fun getResponsibilities() =
        listOf("code", "code review")
}

class QaAnalyst: Employee() {
    override fun getResponsibilities() =
        listOf("write test plans", "execute tests")
}

class UxDesigner: Employee() {
    override fun getResponsibilities() =
        listOf("analyze usability", "UI specs")
}

fun listResponsibilities(employee: Employee): List<String> =
    employee.getResponsibilities()
```

Princípio de Substituição de Liskov

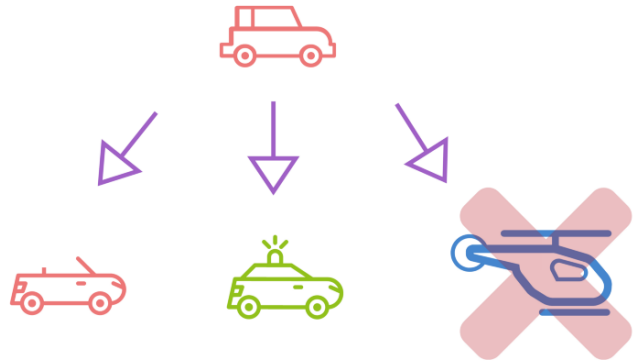
Liskov's Substitution Principle (LSP)

Princípio de Substituição de Liskov

O princípio foi introduzido por Barbara Liskov em 1987 e, de acordo com este princípio, "Classes derivadas ou filhas devem ser substituíveis por suas classes básicas ou parentais".

Definição original: "Se S é um subtipo de T, então os objetos do tipo T podem ser substituídos por objetos do tipo S sem alterar qualquer uma das propriedades desejáveis do programa."

Nova definição: você deve ser capaz de substituir uma coisa por outra se for declarado que essas coisas se comportam da mesma maneira.



Princípio de Substituição de Liskov

```
class Vehicle {
   public int getNumberOfWheels() {
       return 4;
   }
}

class Bicycle extends Vehicle {
   public int getNumberOfWheels() {
       return 2;
   }
}

// calling code
public static int COST_PER_TIRE = 50;
public int tireCost(Vehicle vehicle) {
    return COST_PER_TIRE * vehicle.getNumberOfWheels();
}

Bicycle bicycle = new Bicycle();
System.out.println(tireCost(bicycle)); // 100
```

O motivo?

Referências

- https://www.blogdoft.com.br/2020/02/03/solid-de-verdade-introducao/
- https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf
- https://stackoverflow.blog/2021/11/01/why-solid-principles-are-still-the-foundation-for-modern-software-architecture/
- https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/
- https://www.venturus.org.br/o-que-e-solid/