
BOOTCAMP
FULLSTACK DEVELOPER

SOBRE FELIPE

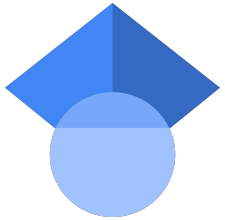
Felipe E. S. de Sousa (Ventania)
10 anos de TI
Há 4 anos na TQI
Eng. de Software
linkedin

SOBRE KLERISSON

BSc 2006, MSc 2009, PhD 2019





BOAS PRÁTICAS

- É um conjunto de técnicas que quando aplicadas geram um bom resultado.

- Exemplo:
  - Se você se alimentar bem e se exercitar, você tem uma probabilidade muito maior de ter uma boa saúde, mesmo se considerarmos outras questões como genética.

Acredite no processo!

Clean Code

Robert C. Martin, “Clean Code” (Código Limpo)

- Conjunto de boas práticas de programação
- Legibilidade e melhor manutenção de código.

- Nomes de variáveis e métodos
- Métodos
- Comentários
- Estrutura de código
- Objetos e estrutura de dados
- Testes
- Code Smells
- Tratamento de erros

Clean Code - Regras Gerais

- Siga as convenções;
- Reduza a complexidade o máximo possível;
- Regra dos escoteiros: “Deixe o acampamento mais limpo do que o encontrou”;
- Sempre encontre a causa raiz;
- Princípio da menor surpresa;
- Evite repetições;

Clean Code - Regras de Design

- Mantenha os dados configuráveis (por exemplo: constantes) em níveis altos. Eles devem ser fáceis de mudar.
- **Prefira polimorfismo a if/else ou switch/case. DIP**
- Evite a configuração excessiva.
- Use injeção de dependência.
- Siga a **Lei de Demeter.** Uma classe deve conhecer apenas suas dependências diretas. Não fale com estranhos ou lei do braço curto.

Clean Code - Regras de Design

```java
class Main {
    public static void main(String[] args) {
        //notApplied
        Person person = new Person();
        System.out.println(person.getAddress().getPostalCode().getNumber());

        //Applied
        System.out.println(person.getPostalCodeNumber());
    }
}
```

Clean Code - Compreensão de código

- Seja consistente;
- Use variáveis explicativas;
- Encapsule as condições de contorno. Lógicas em condições são difíceis de entender;
- Prefira objetos ao invés do tipo primitivo;
- Evite dependência lógica;
    - Métodos precisam funcionar independentemente de outras partes da mesma classe.
- Evite condicionais negativas;

REVOLUCIONE
SUA CARREIRA

Clean Code - Compreensão de código

```java
public class MeaningfulNames {

    int d; // tempo decorrido em dias

    int elapsedTimeInDays;
```

```java
if (level + 1 < args.length) {
    Main main = new Main(args, level + 1);
}

int nextLevel = level + 1;
if(nextLevel < args.length) {
    Main main = new Main(args, nextLevel);
}
```

```java
public static void main(String[] args) {
    String userName = args[1];
    if (isNull(userName))
        throw new RuntimeException("Bad Name");

}

private static boolean isNull(String userName) {
    if (userName == null) {
        return true;
    }
    return false;
}
```

Clean Code - Regras de nomes

- Escolha nomes descritivos e inequívocos.
- Faça uma distinção significativa.
- Use **nomes pronunciáveis**.
- Use **nomes pesquisáveis**.
- Substitua números mágicos por constantes nomeadas.
- Evite codificações. Não acrescente prefixos.

Clean Code - Regras de nomes

```
class Main {
    public static void main(String[] args) {
        Prsn prsn = new Prsn();
        System.out.println(prsn.getPCN());
    }
}
```

Clean Code - Métodos

- Pequenos;
- Faça uma única tarefa;
- Use nomes descritivos;
- Prefira menos argumentos. Não mais do que “3”;
- Não tenha efeitos colaterais;
- Não use argumentos de sinalizador (boolean).
    - Quebre o método em vários métodos independentes que podem ser chamados do cliente sem o sinalizador(flag, boolean);

Clean Code - Métodos

```java
public static void main(String[] args) {
    SideEffect sideEffect = new SideEffect( name: "Klérisson Paixão");
    sideEffect.splitInToFirstName();
    System.out.println(sideEffect.getName());

    sideEffect = new SideEffect( name: "Klérisson Paixão");
    System.out.println(sideEffect.getFirtName());
}
```

```java
public int distanceInMiles(int pointA, int pointB) {
    //...
    return 1;
}

public int distanceInKilometers(int pointA, int pointB) {
    //...
    return 1;
}
```

Clean Code - Regras de comentários

- Sempre tente se explicar no código. Se não for possível, reserve um tempo para escrever um bom comentário.
- Não seja redundante (por exemplo: i++; // incrementa i).
- Não adicione ruído óbvio.
- Não use comentários de chave de fechamento (por exemplo: } // fim da função).
- Não comente o código. Basta remover.
- Caso necessário
  - Use como explicação da intenção.
  - Use como esclarecimento de código.
  - Use como aviso de consequências.

REVOLUCIONE
SUA CARREIRA

Clean Code - Regras de comentários

```
Felipe Eduardo Silvestre de Sousa
class Main {
    Felipe Eduardo Silvestre de Sousa
    public static void main(String[] args) {

        Prsn prsn = new Prsn(); // Instance a Person
        System.out.println(prsn.getPCN()); // Code Working don't comment this one
        //System.out.println(prsn.getAddrs().getPc().getN()); Code commented because I want to test if this is works in other way
    }
}
```

Clean Code - Estrutura de código

- Declare variáveis próximas ao seu uso.
- As funções dependentes devem estar próximas.
- Funções semelhantes devem estar próximas.
- Mantenha as linhas curtas.
- Não use alinhamento horizontal.
- Use espaço em branco para associar coisas relacionadas e desassociar fracamente relacionadas.
- **Não quebre a indentação.**

Clean Code - Objetos e estruturas de dados

- Deve ser pequeno.
- **Faça uma coisa.**
- **Pequeno número de variáveis de instância. Se sua classe tiver muitas variáveis de instância, provavelmente está fazendo mais de uma coisa.**
- A classe base não deve saber nada sobre suas derivadas.
- É melhor ter muitas funções do que **passar alguma flag** em uma função para selecionar um comportamento.
- Prefira métodos não estáticos a métodos estáticos.

REVOLUCIONE
SUA CARREIRA

Clean Code - Objetos e estruturas de dados

```
public static void main(String[] args) {
    Main main = new Main( personName: "Ammelia",  personAddressStreetName: "Ammelia",  personAddressPostalCodeNumber: "1234amelia5");
    System.out.println(main.buildStringConcatOfFields( allFields: true));
    System.out.println(main.buildStringConcatOfFields( allFields: false));
}
```

Clean Code - Testes

- Um assert por teste;
- Velozes;
- Independente;
- Repetível;
- Autovalidação;
- Legível;
- Fácil de executar;
- Use uma ferramenta de cobertura;

Clean Code - Code smells

- Rigidez. O software é difícil de mudar. Uma pequena mudança causa uma cascata de mudanças subsequentes.
- Fragilidade. O software quebra em muitos lugares devido a uma única alteração.
- Imobilidade. Você não pode reutilizar partes do código em outros projetos devido aos riscos envolvidos e ao alto esforço.
- Complexidade Desnecessária.
- Repetição Desnecessária.
- Opacidade. O código é difícil de entender.

Clean Code - Tratamento de Erros

- Não misture manipulação de erros e lógica do negócio.
- Use exceções em vez de retornar códigos de erro.
- Não retorne null, também não passe null.
- Lance exceções com contexto.

Clean Code - Tratamento de Erros

REVOLUCIONE SUA CARREIRA

```
try {
    MealExpenses mealExpenses = expensesReportDAO.getMeals(userName);
    total += mealExpenses.getTotal();
} catch (Exception e) {
    total += getMealPerDiem();
}
```

SOLID

É um conjunto de cinco princípios de boas práticas que foram **agrupados** por **Robert C. Martin** e por ele nomeados no acrônimo SOLID.
Faz parte do que podemos chamar de design principles da programação orientada a objeto (POO).

Muitos princípios convergem entre si. Quando pegamos a ideia por cima dos conceitos, eles parecem um pouco repetitivos, vamos exercitar isto.

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

Crédito a **Ugonna Thelma** pois vamos utilizar as imagens criadas por ela.

Single Responsibility Principle (Princípio de responsabilidade única)

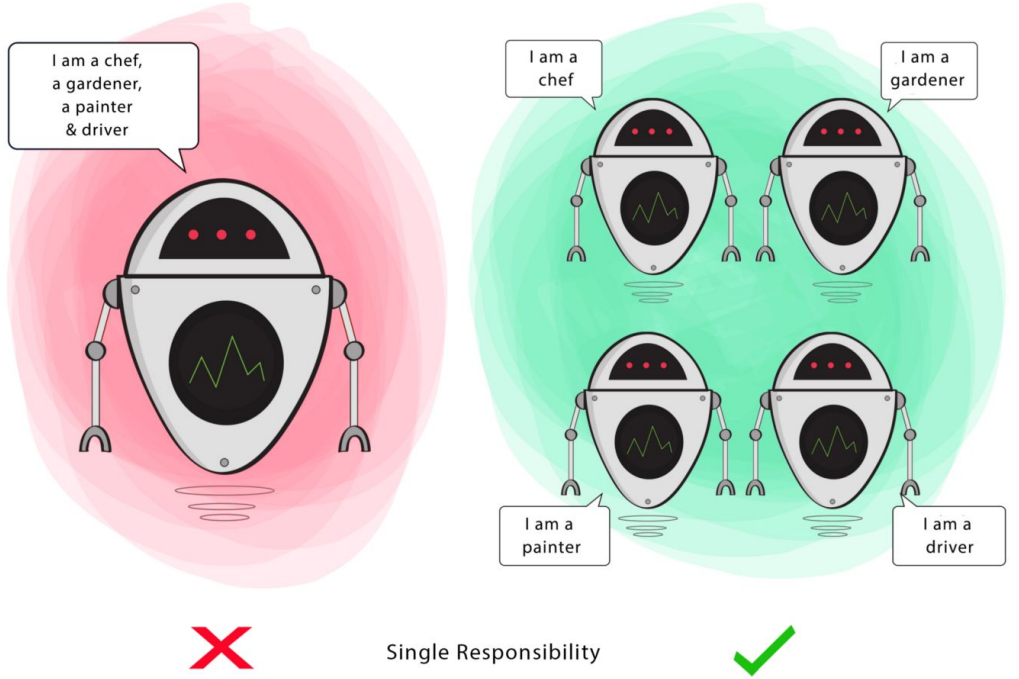
Uma classe deve ter apenas uma responsabilidade.

A responsabilidade de fazer tudo? **Não!!!**

O exercício é conseguir diminuir ao máximo a responsabilidade da classe.

Por quê?

Uma alteração irá impactar apenas o contexto onde ela é necessária.



Single Responsibility Principle (Princípio de responsabilidade única)

```java
package br.com.tqi.cleancodesolid.srp.base;

import java.math.BigInteger;

public class Employee {

    private String name;
    1 usage
    private String addressStreet;
    1 usage
    private BigInteger salaryInCents;


    public BigInteger calculatePayment() {
        return salaryInCents;
    }

    public String getAddressStreet() {
        return addressStreet;
    }
}
```

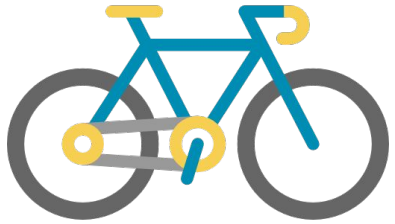
```java
package br.com.tqi.cleancodesolid.srp.applied;

public class Employee extends Person {

    private Address address;
    private Salary salary;

}
```

Open-Close Principle

(Princípio do aberto-fechado)

Uma classe deve estar aberta para extensão e fechada para modificação

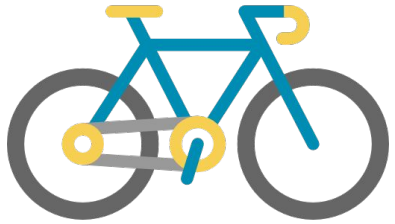
- Capacidade de combinar comportamentos ao invés de alterá-los.



Open-Close Principle

(Princípio do aberto-fechado)

```java
public class Bike {

    private String manufacturer;
    private Gears gears;
    private Wheels wheels;
```



```java
public class EBike extends Bike{

    private Batery batery;
```

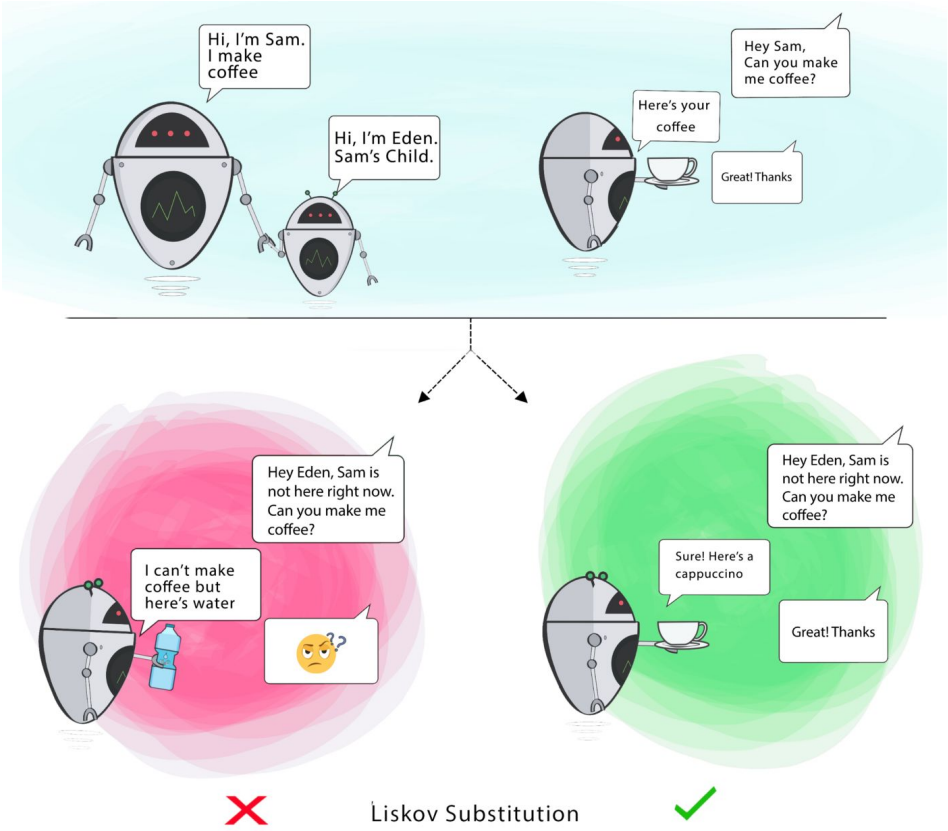
Liskov Substitution Principle (Princípio de substituição de Liskov)

Prover a capacidade de alteração/manutenção de comportamento do sistema por meio de alteração de uma subclasse mantendo o mesmo resultado.

Por quê?

Você consegue mudar o comportamento do sistema apenas trocando a subclasse, ou consegue adicionar um comportamento diferente.

Ex: Mamifero -> (Baleia, Cão) Ação (Locomover). Mesmo resultado



Liskov Substitution Principle (Princípio de substituição de Liskov)

```java
package br.com.tqi.cleancodesolid.lsp.base;

public class MainBaseLsp {

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.walk();

        Whale whale = new Whale();
        whale.swim();
    }

}
```

```java
package br.com.tqi.cleancodesolid.lsp.applied;

public class MainAppliedLsp {
    public static void main(String[] args) {
        Mammal mammal = new Whale();
        mammal.move();
    }
}
```

Interface Segregation Principle (Princípio da segregação de interfaces)

Mantenha as interfaces pequenas.

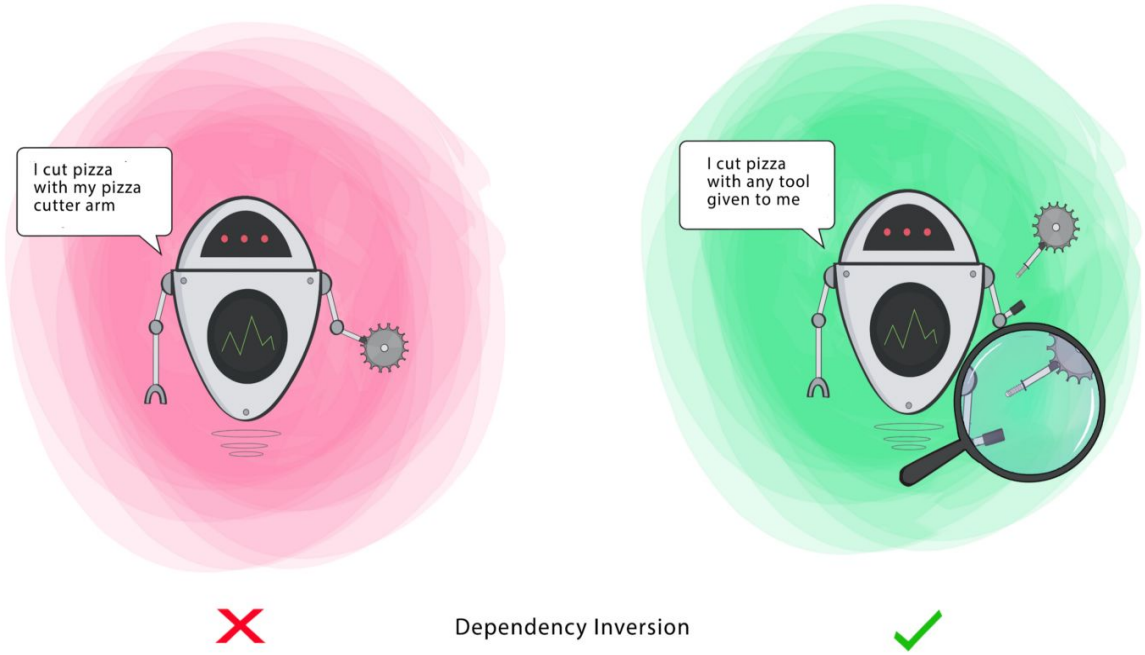
- Assim como no princípio de responsabilidade única, quanto menor for o contexto de uma interface mais fácil será de utilizá-la.

Dependency Inversion Principle (Princípio da inversão de Dependência)

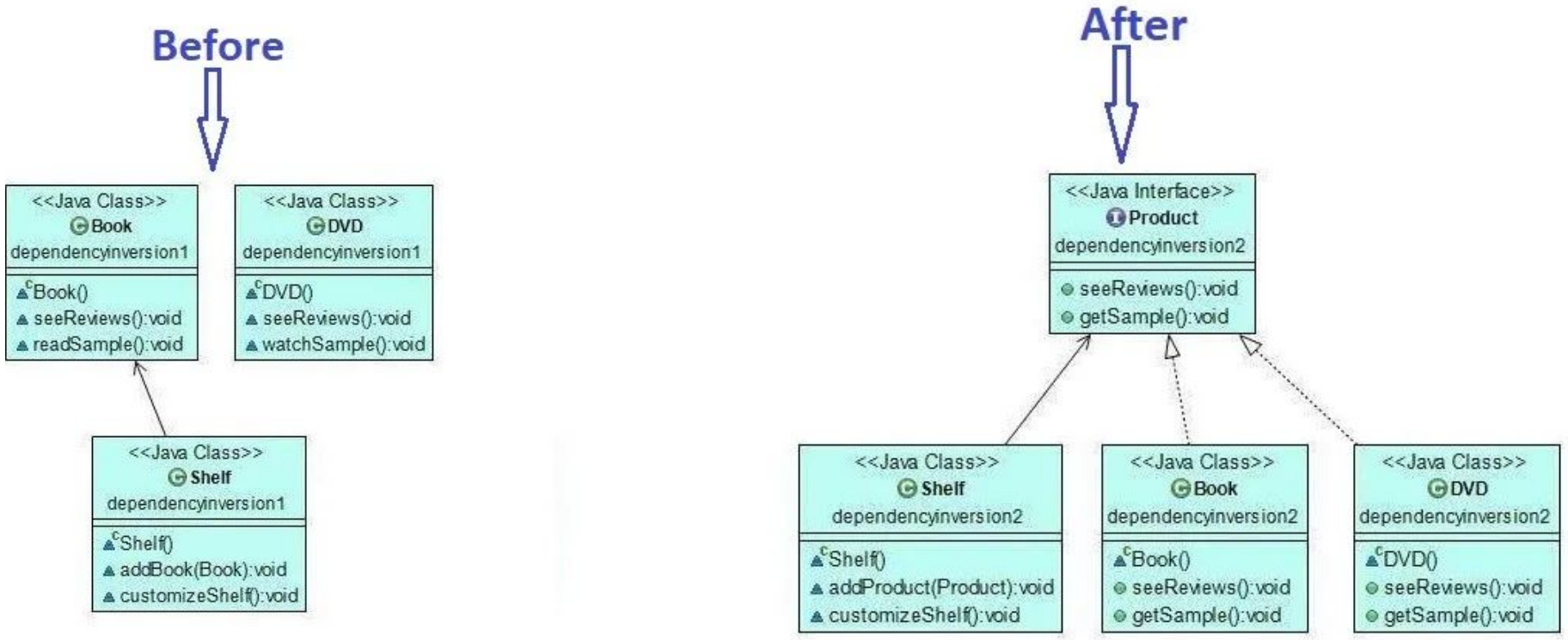
Módulos mais altos não devem depender de módulos mais baixos, ambos devem depender de abstrações.
As abstrações não devem depender de detalhes, os detalhes dependem das abstrações.

Por quê?

Você consegue mudar o comportamento do sistema apenas trocando a subclasse, ou consegue adicionar um comportamento diferente sem que a classe superior fique sabendo.



Dependency Inversion Principle (Princípio da inversão de Dependência)



Mais sobre boas práticas

Livros:

- Refatoração: Aperfeiçoando o Design de Códigos Existentes (Martin Fowler)
- Working Effectively with Legacy Code (Michael Feathers)
- Effective software testing (Maurício Aniche)

Referências

- Código Limpo (Robert C. Martin) Alta Books
- Clean Code Fundamentals (Robert C. Martin), O'Reilly Video Series
- SRP https://javatechonline.com/solid-principles-the-single-responsibility-principle
- LSP https://javatechonline.com/solid-principles-the-liskov-substitution-principle/
- DIP https://javatechonline.com/solid-principles-the-dependency-inversion-principle/