

Aula 14

SOLID

Programação III

Prof. Augusto César Oliveira

augusto.oliveira@unicap.br

Na aula passada...

- Compreender os diferentes tipos de erros em programação;
- Compreender o que é uma exceção e como ela é usada para representar erros durante a execução do programa;
- Conhecer a hierarquia de classes de exceções;
- Utilizar as construções `try` e `catch` para capturar e tratar exceções;
- Criar e lançar exceções personalizadas.

O objetivo da aula de hoje...

- Compreender a importância prática dos princípios de projeto na criação de sistemas mais flexíveis;
- Conhecer e aplicar cada um dos cinco princípios SOLID ao escrever código.

1.

Princípios de projeto

SOLID

Coerência e padronização de funcionalidades

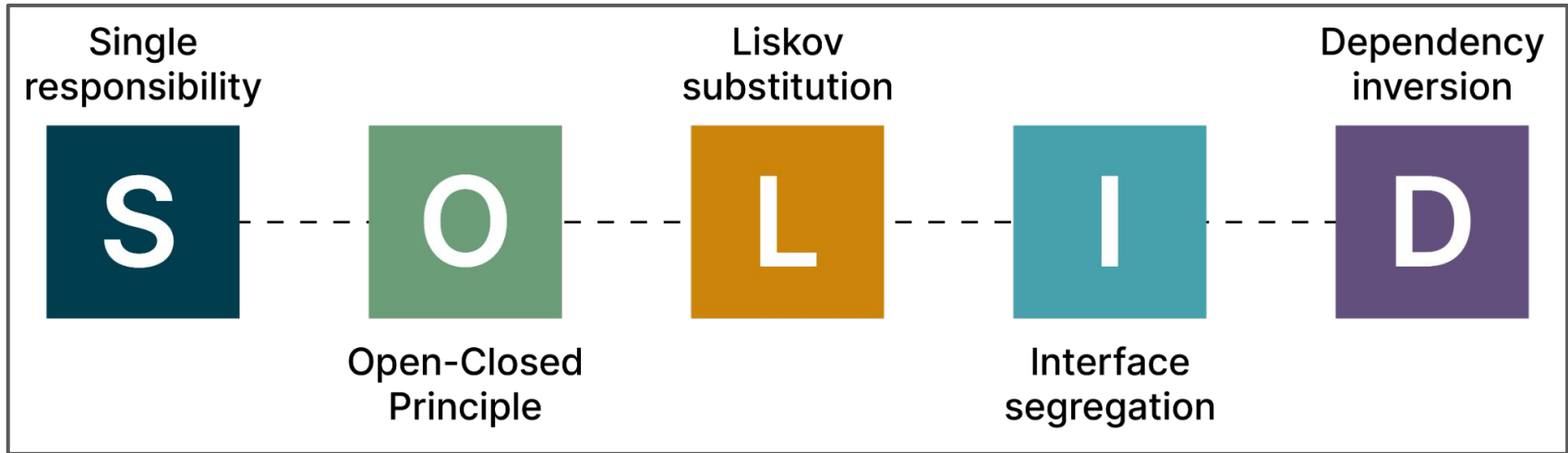


Exemplo



Contraexemplo

SOLID



2.

Princípio da **responsabilidade única**

SOLID



Princípios da responsabilidade única

1. Toda classe deve ter uma **única responsabilidade**.
2. Deve existir um **único motivo** para modificar uma classe.


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

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

Responsabilidade #1: **calcular** índice de desistência

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

Responsabilidade #2: **imprimir** índice de desistência

2.

Separando responsabilidades

SOLID



```
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;  
    }  
  
}
```

```
class Console {  
  
    void imprimeIndiceDesistencia(Disciplina disciplina) {  
        double indice = disciplina.calculaIndiceDesistencia();  
        System.out.println(indice);  
    }  
  
    Uma única responsabilidade: interface com o usuário  
}
```

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

```
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;  
    }  
  
}
```

Uma única responsabilidade: "lógica ou regra de negócio"

Vantagens do princípio de responsabilidade única

- Classe de negócio (Disciplina) pode ser **usada por mais de uma classe de interface com o usuário** (Console, WebApp, MobileApp, etc)
- Divisão de trabalho:
 - Classe de interface: frontend dev;
 - Classe de negócio: backend dev.

3.

Princípio da **segregação de interfaces**

SOLID



Segregação de interfaces

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

```
interface Funcionario {  
  
    double getSalario();  
  
    double getFGTS();// apenas funcionários CLT  
  
    int getSIAPE();// apenas funcionários públicos  
  
    ...  
}
```

4.

Segregando interfaces

SOLID

```
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



5.

Princípio da **inversão de dependências**

SOLID

Inversão de dependências

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

Exemplo: sem usar inversão de dependências

```
class ControleRemoto {  
    TVSamsung tv;  
    ... // métodos  
}
```

```
class TVSamsung {  
    ...  
}
```

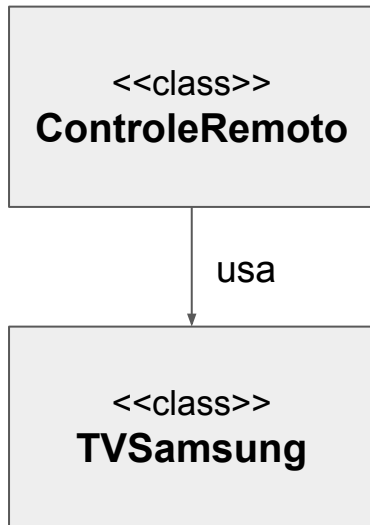
Qual o problema desse design relacionado com acoplamento e extensibilidade?



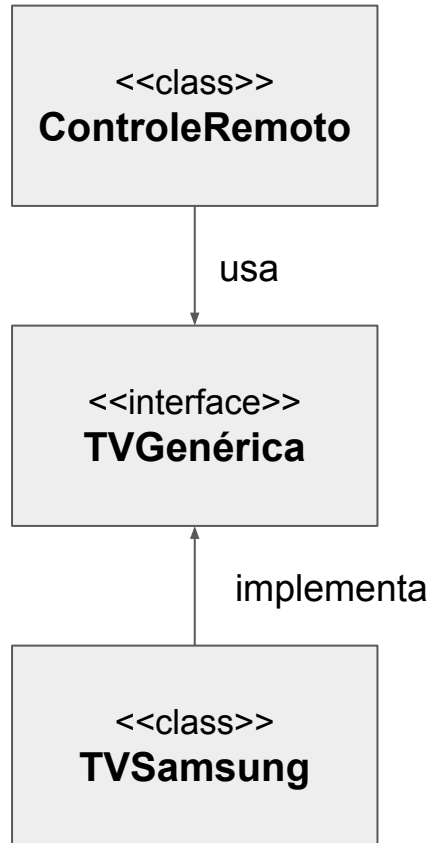
Exemplo: usando inversão de dependências

```
class ControleRemoto {  
    TVGenerica tv;  
    ... // métodos  
}  
  
interface TVGenerica {  
    ... // métodos de qualquer TV  
}  
  
class TVSamsung implements TVGenerica {  
    ...  
}
```

Sem DIP



Com DIP



Vantagem: controle remoto genérico, para qualquer TV.

Posso mudar de TV, sem alterar o código da classe **ControleRemoto**.



6.

Princípio Aberto/Fechado

SOLID



Princípio Aberto/Fechado

- Proposto por Bertrand Meyer.
- Ideia: uma classe deve estar **fechada** para modificações, mas **aberta para extensões**.



1988

Explicando melhor

- Suponha que você vai implementar uma **classe**.
- Usuários ou clientes vão querer usar a classe (óbvio!).
- Mas vão querer também **customizar, parametrizar, configurar, flexibilizar e estender** a classe!
- Você deve se antecipar e **tornar possível** tais extensões.
- Mas sem que os clientes tenham que **editar o código da classe**.

Como tornar uma classe **aberta** a extensões, mas mantendo o seu código **fechado** para modificações?

- Parâmetros;
- Funções de mais alta ordem;
- Padrões de projeto;
- Herança; etc.

Exemplo: ordenando uma lista

Ordena uma lista passada com parâmetro

```
List<String> nomes;  
nomes = Arrays.asList("joao", "maria", "alexandre", "ze");  
Collections.sort(nomes);
```

Exemplo: ordenando uma lista

Ordena uma lista passada com parâmetro

```
List<String> nomes;  
nomes = Arrays.asList("joao", "maria", "alexandre", "ze");  
Collections.sort(nomes);
```

```
System.out.println(nomes);  
// resultado: ["alexandre","joao","maria","ze"]
```

Mudança de planos!

Agora eu quero ordenar as strings da lista pelo seu tamanho, isto é, pelo número de chars

Dúvidas


- Será que o método `sort` está aberto (**preparado**) para permitir essa extensão?
- Mas, mantendo o seu código fechado, isto é, sem ter que mexer no seu código.

Solução: princípio Aberto/Fechado

```
Comparator<String> comparador = new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
};  
Collections.sort(nomes, comparador);
```



Lista de strings



Objeto com um método **compare**, que vai comparar duas strings.
Não existe almoço grátis, cliente tem que implementar esse método

7.

Princípio de substituição de Liskov

SOLID



Princípio de substituição de Liskov

- Nome é uma referência à Profa. **Barbara Liskov**.
- Princípio define **boas práticas para uso de herança**.
- Especificamente, boas práticas para **redefinição de métodos em subclasses**.



Entendendo o termo substituição

```
void f(A a) {  
    ...  
    a.g(int n);  
    ...  
}
```

Entendendo o termo substituição

```
void f(A a) {  
    ...  
    a.g(int n);  
    ...  
}
```

```
f(new B1()); // f pode receber objetos da subclasse B1  
...  
f(new B2()); // e de qualquer outra subclasse de A, como B2  
...  
f(new B3()); // e B3
```

Entendendo o termo substituição

```
void f(A a) {  
    ...  
    a.g(int n);  
    ...  
}
```

- Tipo A pode ser substituído por B1, B2, B3,...
 - Desde que eles sejam subclasses de A.

```
f(new B1()); // f pode receber objetos da subclasse B1  
...  
f(new B2()); // e de qualquer outra subclasse de A, como B2  
...  
f(new B3()); // e B3
```

Princípio de substituição de Liskov

- Substituições de **A** por **B** podem acontecer desde que B ofereça os mesmos serviços de A.
- Ou seja: para um código feito para usar **A**, a substituição de **A** por **B** é "**imperceptível**".

Exemplo que segue substituição de Liskov

```
class ControleRemoto {  
    // alcance de 10 m  
}
```

```
class ControleRemotoPremium extends ControleRemoto {  
    // alcance de 20 m  
}
```

Exemplo que **não** segue substituição de Liskov

```
class ControleRemoto {  
    // alcance de 10 m  
}
```

```
class ControleRemotoXYZ extends ControleRemoto {  
    // alcance de 5 m  
}
```

8.

Considerações finais

SOLID



O que aprendemos hoje?

- A importância prática dos princípios de projeto na criação de sistemas mais flexíveis;
- Os cinco princípios SOLID.

Próxima aula...

-

9.

Exercício de fixação

Teams



SOLID

- **Link da atividade**: clique aqui.

Aula 14

SOLID

Programação III

Prof. Augusto César Oliveira

augusto.oliveira@unicap.br