+Devs2Blu

Linguagem de programação JAVA

Profª. Heloisa Moura

+Devs2Blu

Fundamentos avançados OOP

O que vamos ver:

- Princípios avançados de OOP(SOLID)
- O que é SOLID
- Os 5 princípios
- Princípio da Responsabilidade Única
  (S - Single Responsibility Principle)

Princípios avançados de OOP (SOLID)

- A qualidade de software é um fator fundamental para o sucesso no desenvolvimento de qualquer projeto.
- Aplicações bem projetadas, com boas práticas, códigos limpos e organizados, não só reduzem a incidência de erros, como também facilitam a manutenção e expansão ao longo do tempo.

E aí surge uma questão importante: o que podemos fazer para aumentar a qualidade do software?

Princípios avançados de OOP (SOLID)

Não tem uma única resposta para essa pergunta. Mas, dentre tantas possibilidades, uma opção é utilizar os **princípios SOLID.**

Princípios avançados de OOP (SOLID)

E porque aprender **SOLID** pode ser útil?

- **Melhorar seu código;**
- **Passar nas provas;**
- **Passar nas entrevistas.**

Princípios avançados de OOP (SOLID)

E porque aprender **SOLID** pode ser útil?

O primeiro motivo é mais obvio: Aplicar os princípios do **SOLID** em nossos projetos realmente pode facilitar a nossa vida.

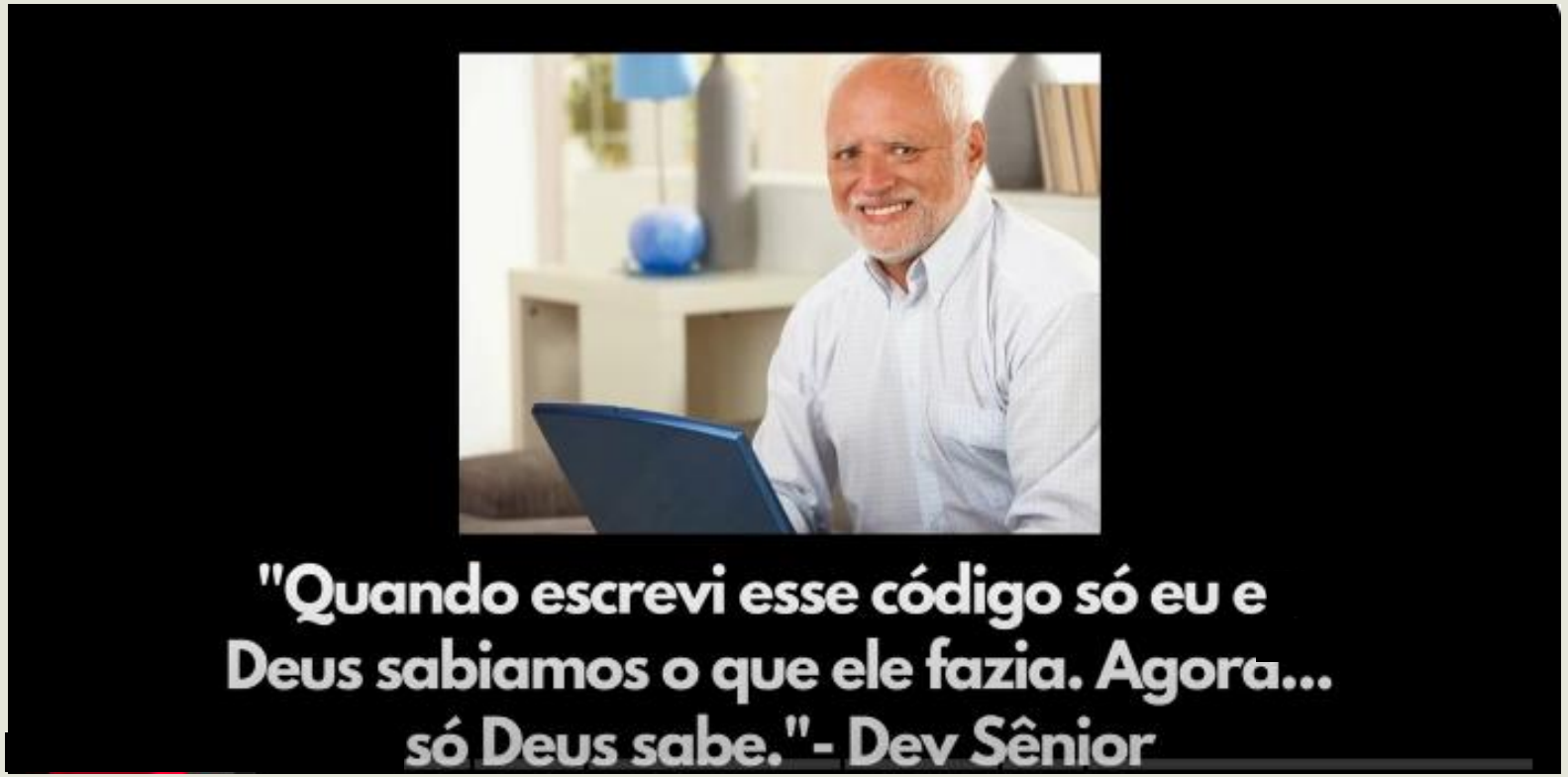
Um código mais limpo, mais organizado e bem estruturado automaticamente vai ser mais fácil de entender. Mesmo que seja um código que só você trabalhe.

+Devs2Blu

Fundamentos avançados OOP

Princípios avançados de OOP(SOLID)

Quem nunca ouviu ou falou essa frase:



O que é SOLID?

- Acrônimo que representa os cinco princípios que facilitam o processo de desenvolvimento — o que facilita a manutenção e a expansão do software.
- Estes princípios são fundamentais na programação orientada a objetos e podem ser aplicados em qualquer linguagem que adote este paradigma.

+Devs2Blu

Fundamentos avançados OOP

Os 5 princípios são:

- **S** — Single Responsibility Principle (Princípio da responsabilidade única)
- **O** — Open-Closed Principle (Princípio Aberto-Fechado)
- **L** — Liskov Substitution Principle (Princípio da substituição de Liskov)
- **I** — Interface Segregation Principle (Princípio da Segregação da Interface)
- **D** — Dependency Inversion Principle (Princípio da inversão da dependência)

Vamos ver cada um

Princípio da Responsabilidade Única
(S - Single Responsibility Principle)

Princípio da Responsabilidade Única (S - Single Responsibility Principle)

- Defende que uma classe ou um método deve ter apenas uma razão para mudar.
- Em outras palavras, uma classe deve ter uma única responsabilidade no sistema.

Importância do SRP no desenvolvimento de software

- Manutenibilidade;
- legibilidade ;
- reusabilidade.

Exemplo do dia a dia

- Comparação com um canivete suíço: Muitas funcionalidades em uma única ferramenta podem tornar seu uso confuso e ineficaz.

Exemplo do dia a dia

- Se você precisar afiar algo, usaria uma faca comum ou o canivete suíço?

Problemática: Classe com muitas Responsabilidades

O que fazer?

Problemática: Classe GerenciadorTarefas

Código inicial:

```
public class GerenciadorTarefas {

public String conectarAPI(){ //... }
public void criarTarefa(){ //... }
public void atualizarTarefa(){ //... }
public void removerTarefa(){ //... }
public void enviarNotificacao(){ //... }
public void produzirRelatorio(){ //... }
public void enviarRelatorio(){ //... }

}
```

Perguntas:

- Quantas responsabilidades essa classe tem?
- Quais problemas podemos enfrentar com esse design?

Exercício

- Enumerarem as responsabilidades da classe GerenciadorTarefas

Gerenciamento de tarefas.

Conexão com API.

Envio de notificações.

Geração e envio de relatórios.

Análise crítica:

Um gerenciador de tarefas deveria cuidar de notificações e relatórios?

Não! Um gerenciador de tarefas gerencia as tarefas, não e-mails ou relatórios.

Solução : Aplicando o Princípio da Responsabilidade Única

Refatoração do código:

- Divisão da classe GerenciadorTarefas em várias classes menores, cada uma com sua própria responsabilidade.
- Nossa classe GerenciadorTarefas terá apenas o código relacionando a operação com tarefas. Assim, teremos a classe GerenciadorTarefas refatorada:

Princípio da Responsabilidade Única (S - Single Responsibility Principle)

```
public class GerenciadorTarefas {

public void criarTarefa(){ //... }
public void atualizarTarefa(){ //... }
public void removerTarefa(){ //... }


}
```

Princípio da Responsabilidade Única (S - Single Responsibility Principle)

Assim, vamos criar uma classe para consumir uma API externa, outra classe para enviar notificações e uma última classe para lidar com os relatórios.

```
public class ConectorAPI {
	public String conectarAPI() { //... }
}
public class Notificador {
	public void enviarNotificacao() { //... }
}
public class GeradorRelatorio {
	public void produzirRelatorio(){ //... }
	public void enviarRelatorio(){ //... }
}
```

Resumo

Cada classe reflete exatamente a responsabilidade que ela tem.
Se precisarmos adicionar algum método, por exemplo, relacionado ao consumo da API, vamos saber exatamente em qual parte do código devemos ir. Ou seja, fica muito mais fácil alterar o que for preciso.

Exercício

Discussão: Vantagens de Aplicar o SRP

Facilidade de manutenção:
Alterar ou adicionar funcionalidades sem impactar outras áreas do código.

Reusabilidade:
Classes menores podem ser reaproveitadas em outros projetos.

Facilidade de teste:
Testar uma classe com responsabilidade única é mais simples e direto.

Legibilidade:
Código mais claro e fácil de entender.

Conclusão

Reforçando a importância do SRP:

- Manutenibilidade;
- legibilidade ;
- reusabilidade.

Contribui para um código mais limpo e eficiente.