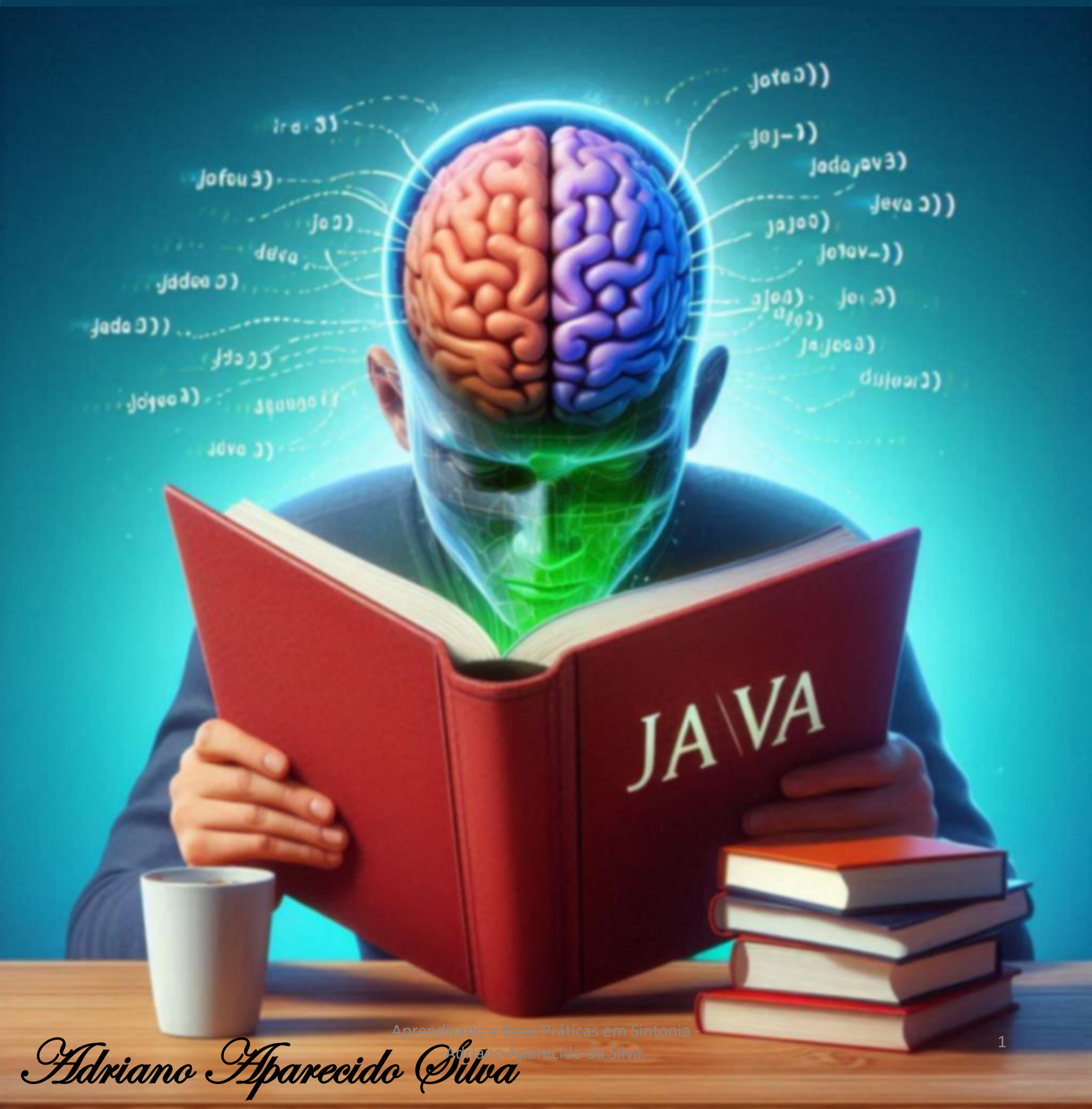
Java

Trilhando o Caminho Java: Aprendizado e Boas Práticas em Sintonia



Adriano Aparecido Silva

Desvendando os Fundamentos da Programação Orientada a Objetos e os Princípios SOLID em Java

A Programação Orientada a Objetos (POO) revolucionou a forma como desenvolvemos software, permitindo uma modelagem mais próxima do mundo real

A Orientação a Objetos (OO) é uma abordagem fundamental no desenvolvimento de software. Ela permite modelar sistemas como coleções de objetos interconectados, cada um com seu próprio estado e comportamento. Neste e-book, exploraremos os quatro pilares da OO e os princípios SOLID, fornecendo exemplos práticos em Java.

.

Orientação a Objetos

A Orientação a Objetos (OO) é uma abordagem fundamental no desenvolvimento de software. Ela permite modelar sistemas como coleções de objetos interconectados, cada um com seu próprio estado e comportamento..

01

Encapsulamento

Orientação a Objetos

Encapsulamento: Protegendo o Interior

O encapsulamento é o primeiro pilar da OO. Ele envolve esconder os detalhes internos de um objeto e expõem apenas uma interface pública. Isso promove a segurança e a manutenção do código.
Observe: Nesse exemplo, o saldo é encapsulado e só pode ser acessado através dos métodos depositar e consultarSaldo.

Encapsulamento

```java
public class ContaBancaria {
    private double saldo;

    public void depositar(double valor) {
        saldo += valor;
    }

    public double consultarSaldo() {
        return saldo;
    }
}
public class ContaBancaria {
    private double saldo;

    public void depositar(double valor) {
        saldo += valor;
    }

    public double consultarSaldo() {
        return saldo;
    }
}
```

codesnap.dev

02

Herança

Orientação a Objetos

Herança

A herança permite criar novas classes baseadas em classes existentes. Ela promove a reutilização de código e a extensibilidade.
Observe: Aqui, Cachorro herda de Animal e sobrescreve o método emitirSom.

Herança

```java
class Animal {
    void emitirSom() {
        System.out.println("Som genérico de animal");
    }
}

class Cachorro extends Animal {
    @Override
    void emitirSom() {
        System.out.println("Latido de cachorro");
    }
}
```

codesnap.dev

03

Polimorfismo

Orientação a Objetos

Polimorfismo

O polimorfismo permite que objetos de diferentes classes sejam tratados de forma uniforme.
Observe: Aqui, podemos tratar tanto Forma quanto Circulo de maneira polimórfica.

Herança

```
class Animal {
    void emitirSom() {
        System.out.println("Som genérico de animal");
    }
}

class Cachorro extends Animal {
    @Override
    void emitirSom() {
        System.out.println("Latido de cachorro");
    }
}
```

codesnap.dev

04

Abstração

Orientação a Objetos

Abstração

A abstração envolve a criação de classes e métodos que representam conceitos do mundo real.
Observe : Veiculo é uma classe abstrata que define o método acelerar, que deve ser implementado pelas subclasses, como Carro.

Abstração

```java
abstract class Veiculo {
    abstract void acelerar();
}

class Carro extends Veiculo {
    @Override
    void acelerar() {
        System.out.println("Carro acelerando");
    }
}
```

codesnap.dev

Princípios do SOLID

Os princípios SOLID são diretrizes para escrever código flexível e de fácil manutenção.

05

Princípio da Responsabilidade Única

Princípios SOLID

Princípio da Responsabilidade Única(RSP)

O Princípio da Responsabilidade Única (SRP) é um dos pilares do SOLID e enfatiza que uma classe deve ter apenas um motivo para mudar. Em outras palavras, cada classe deve ter uma única responsabilidade e desempenhar apenas essa função de maneira eficiente.
Vamos considerar um exemplo para ilustrar o SRP:
Situação Inicial:
Temos uma classe chamada User que possui três métodos com diferentes responsabilidades:

- registrarUsuario: Responsável por registrar um novo usuário.
- enviarEmail: Responsável por enviar e-mails.
- gerarRelatorio: Responsável por gerar relatórios.

Princípios SOLID

Princípio da Responsabilidade Única(RSP)

Sólido (RSP)

```
class User {
class User {
class User {
    void registrarUsuario() {
        // Lógica para registrar um usuário
    }

    void enviarEmail() {
        // Lógica para enviar e-mails
    }

    void gerarRelatorio() {
        // Lógica para gerar relatórios
    }
}
```

codesnap.dev

Princípios SOLID

Princípio da Responsabilidade Única(RSP)

Então temo sum Problema:
Essa classe está assumindo responsabilidades que não são suas. Ela está fazendo coisas demais e violando o SRP.
Se qualquer uma dessas responsabilidades mudar, a classe precisará ser alterada.

A Solução (Aplicando o SRP) é:
Refatorar a classe User para separar essas responsabilidades em classes distintas:
Depois de refatorada teremos três classes, cada uma com uma única responsabilidade.
Isso torna o código mais organizado, facilita a manutenção e permite que cada classe evolua independentemente.

Lembre-se: uma classe, um motivo para mudar!

Princípios SOLID

Princípio da Responsabilidade Única(RSP)

Sólido (RSP)

```java
class UsuarioService {
    void registrarUsuario() {
        // Lógica para registrar um usuário
    }
}

class EmailService {
    void enviarEmail() {
        // Lógica para enviar e-mails
    }
}

class RelatorioService {
    void gerarRelatorio() {
        // Lógica para gerar relatórios
    }
}
```

codesnap.dev

06

Princípio do Aberto/Fechado

Princípios SOLID

Princípio do Aberto/Fechado (OCP)

O Princípio do Aberto/Fechado (OCP) é um dos pilares do SOLID e enfatiza que um módulo deve estar aberto para extensão, mas fechado para modificação. Vamos explorar esse princípio com um exemplo em Java.

Suponha que estamos desenvolvendo um sistema de processamento de pagamentos. Inicialmente, temos uma classe chamada Pagamento que lida com diferentes tipos de pagamento:

Sólido (OCP)

```
class Pagamento {
    void efetuarPagamento(String tipo, Integer codigo, Double valor) {
        if ("BOLETO".equals(tipo)) {
            new IntegracaoBoletoBanco().pagarBoleto(codigo, valor);
        } else if ("CARTAO".equals(tipo)) {
            new IntegracaoCartaoBanco().pagarCartao(codigo, valor);
        } else if ("DINHEIRO".equals(tipo)) {
            new IntegracaoContaBanco().pagarDinheiro(valor);
        }
    }
}
```

codesnap.dev

Princípios SOLID

Princípio do Aberto/Fechado (OCP)

O Princípio do Aberto/Fechado (OCP) é um dos pilares do SOLID e enfatiza que um módulo deve estar aberto para extensão, mas fechado para modificação. Vamos explorar esse princípio com um exemplo em Java.

Suponha que estamos desenvolvendo um sistema de processamento de pagamentos. Inicialmente, temos uma classe chamada Pagamento que lida com diferentes tipos de pagamento:

Sólido (OCP)

```
class Pagamento {
    void efetuarPagamento(String tipo, Integer codigo, Double valor) {
        if ("BOLETO".equals(tipo)) {
            new IntegracaoBoletoBanco().pagarBoleto(codigo, valor);
        } else if ("CARTAO".equals(tipo)) {
            new IntegracaoCartaoBanco().pagarCartao(codigo, valor);
        } else if ("DINHEIRO".equals(tipo)) {
            new IntegracaoContaBanco().pagarDinheiro(valor);
        }
    }
}
```

codesnap.dev

Princípios SOLID

Princípio do Aberto/Fechado (OCP)

No código mostrado anteriormente, estamos verificando o tipo de pagamento e chamando métodos específicos para cada tipo.
No entanto, essa abordagem é altamente acoplada e requer modificações sempre que um novo tipo de pagamento é adicionado ou alterado.

Agora, vamos aplicar o Princípio do Aberto/Fechado para tornar o código mais flexível e extensível:
O código a seguir é mais genérico e permite adicionar novas implementações de pagamento sem modificar o código existente.
O Princípio do Aberto/Fechado nos ajuda a criar sistemas mais flexíveis e menos propensos a erros quando novos requisitos surgem

07

Princípio da Substituição de Liskov

Princípios SOLID

Princípio da Substituição de Liskov (LSP)

O Princípio da Substituição de Liskov (LSP) é um dos pilares do SOLID e tem a ver com a herança em programação orientada a objetos. Vamos simplificar esse conceito com um exemplo em Java.

Imagine que estamos modelando um sistema de animais. Temos uma classe base chamada Animal com três propriedades: Id, Nome e Patas. Além disso, temos duas subclasses: Mamífero e Ave.

Classe Base Animal:
A classe Animal possui as seguintes propriedades:

Sólido (LSP)

```
class Animal {
    int Id;
    String Nome;
    int Patas;
}
```

codesnap.dev

Princípios SOLID

Princípio da Substituição de Liskov (LSP)

Subclasse Mamífero:
A classe Mamífero herda de Animal e adiciona quatro propriedades: Latir, Miar, Voar e Nadar.

Sólido (LSP)

```
class Mamifero extends Animal {
    boolean Latir;
    boolean Miar;
    boolean Voar;
    boolean Nadar;
}
```

codesnap.dev

Princípios SOLID

Princípio da Substituição de Liskov (LSP)

Subclasse Ave:
A classe Ave também herda de Animal e adiciona duas propriedades: Voar e OvosPorMes.

Sólido (LSP)

```
class Ave extends Animal {
    boolean Voar;
    int OvosPorMes;
}
```

codesnap.dev

Princípios SOLID

Princípio da Substituição de Liskov (LSP)

Conclusão:

Se temos um software que usa a classe Animal e seu comportamento não muda quando substituímos uma instância de Ave (por exemplo, um pássaro) por uma instância de Mamífero (por exemplo, um cachorro), então estamos seguindo o LSP.

Isso significa que Ave e Mamífero são subtipos de Animal.

Em resumo, o LSP nos diz para garantir que as subclasses sejam 100% compatíveis com a classe pai. Isso ajuda a manter a consistência e a prevenir problemas quando substituímos objetos em nosso código

08

Princípio da Segregação de Interface

Princípios SOLID

Princípio da Segregação de Interface (ISP)

O Princípio da Segregação de Interface (ISP) é um dos pilares do SOLID e enfatiza que interfaces específicas são melhores do que uma única interface de propósito geral. Vamos explorar esse princípio com um exemplo em Java.

Imagine que estamos desenvolvendo um sistema de envio de e-mails. Temos três tipos de e-mails: simples, com anexo e com formatação especial. Vamos aplicar o ISP para garantir que nossas interfaces sejam específicas e atendam apenas aos comportamentos necessários.
Situação Inicial:
Temos uma classe Mailer que lida com e-mails simples e outra classe AttachmentMailer que lida com e-mails que possuem anexos:

Sólido (ISP)

```
class Mailer {
    void deliver(String recipient, String message) {
        // Lógica para enviar e-mail simples
    }
}

class AttachmentMailer {
    void enqueue(String recipient, String message, List<Attachment> attachments) {
        // Lógica para enfileirar e-mails com anexos
    }
}
```

codesnap.dev

Princípios SOLID

Princípio da Segregação de Interface (ISP)

Temos um Problema:
A classe MailerService precisa lidar com ambos os tipos de e-mails, mas não precisa de todos os métodos da interface AttachmentMailer.
Solução (Aplicando o ISP):
Vamos criar interfaces segregadas para cada tipo de e-mail.

Desta forma teremos interfaces específicas para cada tipo de e-mail, permitindo que as classes implementem apenas os comportamentos necessários.
Isso torna nosso código mais flexível e evita a dependência de métodos não utilizados

Princípios SOLID

Princípio da Segregação de Interface (ISP)

Sólido (ISP)

```java
interface SimpleMailer {
    void deliver(String recipient, String message);
}

interface AttachmentMailer {
    void enqueue(String recipient, String message, List<Attachment> attachments);
}

class MailerService {
    private final SimpleMailer simpleMailer;
    private final AttachmentMailer attachmentMailer;

    MailerService(SimpleMailer simpleMailer, AttachmentMailer attachmentMailer) {
        this.simpleMailer = simpleMailer;
        this.attachmentMailer = attachmentMailer;
    }

    void deliverTo(String recipient, String message, boolean sendAttachment) {
        if (sendAttachment) {
            attachmentMailer.enqueue(recipient, message, getAttachments());
        } else {
            simpleMailer.deliver(recipient, message);
        }
    }

    private List<Attachment> getAttachments() {
        // Lógica para obter anexos
        return Collections.emptyList();
    }
}
```

OBRIGADO POR LER ATÉ AQUI

Esse Ebook foi gerado por IA, todo conteúdo gerado pelo Chat GPT e Copilot foram revisados e combinados, para ter um conteúdo mais completo, simples e conciso! Todo o processo de diagramação foi realizado por um ser humano.
O passo a passo se encontra no meu Github.

https://github.com/AdrianoProfileAdsCloud/Bootcamp-Santander-2024-Fundamentos-de-IA-para-Devs-Cria-de-um-Ebook-com-ChatGPT-MidJourney

Autor

Adriano Aparecido da Silva

GitHub | **Linkedin**