ARTIGO - BOAS PRÁTICAS DE PROGRAMAÇÃO: S.O.L.I.D.

Os princípios **SOLID** são um conjunto de cinco princípios de design orientado a objetos que foram introduzidos por Robert C. Martin e são considerados essenciais para construir software de qualidade. Esses princípios incluem:

1. SRP (Single Responsibility Principle) - Princípio da Responsabilidade Única
2. OCP (Open-Closed Principle) - Princípio Aberto-Fechado
3. LSP (Liskov Substitution Principle) - Princípio da Substituição de Liskov
4. ISP (Interface Segregation Principle) - Princípio da Segregação de Interfaces
5. DIP (Dependency Inversion Principle) - Princípio da Inversão de Dependência

O Princípio da Responsabilidade Única (SRP) estabelece que cada classe deve ter apenas uma responsabilidade.

O Princípio Aberto-Fechado (OCP) afirma que as classes devem ser abertas para extensão, mas fechadas para modificação.

O Princípio da Substituição de Liskov (LSP) define que os objetos de uma classe filha devem ser capazes de serem substituídos pelos objetos da classe pai.

O Princípio da Segregação de Interfaces (ISP) estabelece que as interfaces devem ser segregadas em responsabilidades menores e mais específicas.

Por fim, o Princípio da Inversão de Dependência (DIP) indica que as classes de nível superior não devem depender de classes de nível inferior, mas sim de abstrações. Vamos abordar os princípios acima em detalhes a seguir.

SRP - Princípio da Responsabilidade Única

O Princípio da Responsabilidade Única (SRP), que afirma que **uma classe deve ter apenas uma responsabilidade**. Isso significa que a classe deve ter apenas um único propósito e uma única razão para mudar. Um exemplo que pode ilustrar o SRP é uma classe que tem a responsabilidade de calcular o imposto de renda - IRPF e, ao mesmo tempo, gerar um relatório com os valores dos impostos. Essa classe viola o princípio da SRP, pois tem duas responsabilidades diferentes. O ideal seria separar essas duas funcionalidades em duas classes diferentes, onde uma seria responsável pelo cálculo do imposto e outra seria responsável pela geração do relatório.

Outro exemplo, em linguagem Java, ilustra melhor esse princípio. No exemplo, a classe **`Funcionario`** tem três responsabilidades: atualizar o salário, promover o funcionário a um novo cargo e salvar o funcionário em um banco de dados. Isso viola o SRP, pois a classe tem mais de uma razão para mudar. Na segunda versão, foram criadas duas classes adicionais, **`Promocao`** e **`BancoDeDados`**, para lidar com as responsabilidades de promover o funcionário e salvar o funcionário no banco de dados, respectivamente. Dessa forma, a classe **`Funcionario`** agora tem apenas uma responsabilidade, tornando o código mais organizado e manutenível.

```java
// Classe sem a aplicação do SRP
public class Funcionario {
    private String nome;
    private String cargo;
    private double salario;
    private int id;

    public Funcionario(String nome, String cargo, double salario, int id)
{
        this.nome = nome;
        this.cargo = cargo;
        this.salario = salario;
        this.id = id;
    }

    public void atualizarSalario(double novoSalario) {
        // Atualiza o salário do funcionário
    }

    public void promoverFuncionario(String novoCargo) {
        // Promove o funcionário a um novo cargo
    }

    public void salvarFuncionario(Funcionario funcionario) {
        // Salva o funcionário em um banco de dados
    }
}

// Classe com a aplicação do SRP
public class Funcionario {
    private String nome;
    private String cargo;
    private double salario;
    private int id;

    public Funcionario(String nome, String cargo, double salario, int id)
{
        this.nome = nome;
        this.cargo = cargo;
        this.salario = salario;
        this.id = id;
    }

    public void atualizarSalario(double novoSalario) {
        // Atualiza o salário do funcionário
    }

    // Removeu os métodos de promoverFuncionario e salvarFuncionario para
outras classes
```

```
}

public class Promocao {
    public void promoverFuncionario(Funcionario funcionario, String
novoCargo) {
        // Promove o funcionário a um novo cargo
    }
}

public class BancoDeDados {
    public void salvarFuncionario(Funcionario funcionario) {
        // Salva o funcionário em um banco de dados
    }
}
```

OCP - Princípio Aberto-Fechado

O Princípio Aberto-Fechado (Open-Closed Principle - OCP) afirma que o software deve ser "**aberto para extensão**" e "**fechado para modificação**". Isso significa que um software deve ser projetado de tal forma que **novos recursos possam ser adicionados sem alterar o código existente**. Para isso, o código deve ser separado em módulos que possam ser estendidos ou substituídos por outros módulos.

Um exemplo de implementação do OCP em Java pode ser uma classe de pagamento que pode processar diferentes tipos de pagamento, como cartão de crédito, transferência bancária, PayPal, PIX entre outros. Em vez de incluir todas as lógicas de pagamento em uma única classe, essa classe pode ser abstrata e criar sub-classes específicas para cada tipo de pagamento, que estendem a classe abstrata. Dessa forma, se um novo tipo de pagamento precisar ser adicionado, basta criar uma nova sub-classe e não modificar o código existente.

Segue abaixo um exemplo de código Java que ilustra o Princípio Aberto-Fechado:

```
abstract class Pagamento {
   public abstract void processarPagamento();
}

class PagamentoCartaoCredito extends Pagamento {
   public void processarPagamento() {
      // Lógica para processar pagamento com cartão de crédito
   }
}

class PagamentoTransferenciaBancaria extends Pagamento {
   public void processarPagamento() {
      // Lógica para processar pagamento com transferência bancária
   }
}

class PagamentoPayPal extends Pagamento {
```

```
    public void processarPagamento() {
        // Lógica para processar pagamento com PayPal
    }

class PagamentoPix extends Pagamento {
    public void processarPagamento() {
        // Lógica para processar pagamento com Pix
    }
}
```

Nesse exemplo, a classe abstrata **`Pagamento`** define um método **`processarPagamento()`** que é implementado em cada sub-classe específica de pagamento. Dessa forma, novos tipos de pagamento podem ser adicionados sem modificar a classe **`Pagamento`**. Isso exemplifica o princípio OCP, onde o código está fechado para modificações, mas aberto para extensões.

LSP - Princípio da Substituição de Liskov

O princípio de LSP estende o princípio OCP, com foco no comportamento de uma superclasse e seus subtipos. Ele afirma que **as subclasses devem ser substituíveis por suas classes base**, ou seja, o código esperando uma determinada classe ser utilizada deve funcionar se for passado qualquer uma das subclasses dessa classe

Um exemplo simples de código em Java que ilustra o princípio de LSP é uma hierarquia de classes com uma classe base `Animal` e duas subclasses, `Dog` e `Cat`. A classe Animal possui um método `makeSound()`, que retorna um som genérico. As subclasses `Dog` e `Cat` sobrescrevem esse método para retornar seus respectivos sons específicos. Um cliente que espera uma instância da classe Animal pode receber qualquer uma das subclasses (`Dog` ou `Cat`) sem precisar modificar o código, pois ambas **as subclasses têm o mesmo comportamento da classe base.** Segue abaixo um exemplo de código em Java que ilustra o princípio de LSP:

```
class Animal {
    public String makeSound() {
        return "generic sound";
    }
}

class Dog extends Animal {
    public String makeSound() {
        return "bark";
    }
}

class Cat extends Animal {
    public String makeSound() {
        return "meow";
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();
        System.out.println(animal1.makeSound());
        System.out.println(animal2.makeSound());
    }
}
```

Nesse exemplo, a classe `Animal` é a classe base, enquanto as classes `Dog` e `Cat` são suas subclasses. A sobrescrita do método `makeSound()` em cada uma das subclasses não viola o princípio de LSP, pois ambos os métodos retornam uma String e possuem o mesmo comportamento que a implementação na classe base.

ISP - Princípio da Segregação de Interfaces

o princípio da segregação das interfaces é definido como "Clientes não devem ser forçados a depender de interfaces que eles não usam."

Em outras palavras, o ISP diz que uma classe deve ter apenas métodos que sejam relevantes para sua funcionalidade e não deve forçar seus usuários a implementar métodos desnecessários. Isso é importante para evitar interfaces "gordas" que contêm muitos métodos, muitos dos quais não são relevantes para a classe em questão.

Para ilustrar o conceito de ISP em Java, considere o exemplo a seguir. Digamos que temos uma interface chamada **`Animal`** que define um método **`voar()`**. No entanto, nem todos os animais voam. Ao aplicar o ISP, podemos criar uma nova interface chamada **`AnimalQueVoa`** que estende a interface **`Animal`** e define o método **`voar()`**. Então, apenas as classes que precisam implementar a funcionalidade de voo implementarão a nova interface. Dessa forma, classes que não precisam implementar o método **`voar()`** não serão forçadas a implementar uma funcionalidade desnecessária.

```
interface Animal {
    void mover();
}

interface AnimalQueVoa extends Animal {
    void voar();
}

class Pato implements AnimalQueVoa {
    @Override
    public void mover() {
        // implementação para mover
    }

    @Override
    public void voar() {
        // implementação para voar
```

```
    }
}

class Cachorro implements Animal {
    @Override
    public void mover() {
        // implementação para mover
    }
}
```

No exemplo acima, **`Animal`** define o método **`mover()`** que é implementado por todas as classes que representam animais. A nova interface **`AnimalQueVoa`** estende **`Animal`** e adiciona o método **`voar()`**. A classe **`Pato`** implementa **`AnimalQueVoa`** e implementa os métodos **`mover()`** e **`voar()`**. A classe **`Cachorro`**, por outro lado, implementa apenas **`Animal`** e não precisa implementar o método **`voar()`**.

DIP - Princípio da Inversão de Dependência

O princípio da inversão de dependência é um paradigma de programação simples, mas poderoso, que pode ser usado para implementar componentes de software bem estruturados, altamente desacoplados e reutilizáveis.

O DIP é a prática de depender de abstrações e não de implementações, ou seja, as classes de nível superior devem depender de abstrações (interfaces ou classes abstratas), e não de classes de baixo nível. As abstrações são usadas para desacoplar as classes, tornando-as independentes e fáceis de manter e modificar. Isso significa que as classes de nível superior não precisam saber os detalhes de implementação das classes de nível inferior, apenas precisam saber o que essas classes fazem através de suas abstrações.

Segue abaixo um exemplo em Java para ilustrar o conceito de DIP:

```
interface UserService {
    void save(User user);
}

class DatabaseUserService implements UserService {
    public void save(User user) {
        // salva usuário no database
    }
}

class MemoryUserService implements UserService {
    public void save(User user) {
        // salva usuário na memória
    }
}

class UserController {
    private UserService userService;
```

```
    public UserController(UserService userService) {
        this.userService = userService;
    }

    public void saveUser(User user) {
        userService.save(user);
    }
}

public class Main {
    public static void main(String[] args) {
        UserService userService = new MemoryUserService();
        UserController userController = new UserController(userService);
        User user = new User("John", "Doe");
        userController.saveUser(user);
    }
}
```

Neste exemplo, temos a interface **`UserService`** que define um método **`save`** para salvar um usuário. Em seguida, temos duas implementações diferentes dessa interface, **`DatabaseUserService`** e **`MemoryUserService`**, que salvam o usuário em um banco de dados e em memória, respectivamente.

Por fim, temos a classe **`UserController`**, que recebe uma instância de **`UserService`** no construtor e usa o método **`saveUser`** para salvar um usuário através dessa instância. Note que a classe **`UserController`** depende apenas da abstração **`UserService`** e não da implementação específica, permitindo a fácil troca de implementação no futuro, sem alterações na classe **`UserController`**.

REFERÊNCIAS:

https://stackify.com/solid-design-principles/https://www.geeksforgeeks.org/single-responsibility-principle-in-java-with-examples/https://www.baeldung.com/java-single-responsibility-principlehttps://www.freecodecamp.org/news/open-closed-principle/https://www.freecodecamp.org/news/open-closed-principle-solid-architecture-concept-explained/https://stackify.com/solid-design-open-closed-principle/https://www.baeldung.com/cs/liskov-substitution-principlehttps://stackify.com/solid-design-liskov-substitution-principle/https://reflectoring.io/lsp-explained/https://stackify.com/interface-segregation-principle/https://www.baeldung.com/java-interface-segregationhttps://reflectoring.io/interface-segregation-principle/https://www.baeldung.com/java-dependency-inversion-principlehttps://stackify.com/dependency-inversion-principle/https://www.geeksforgeeks.org/dependecy-inversion-principle-solid/

Livros escritos por Rober C. Martin:

https://www.goodreads.com/author/list/45372.Robert_C_Martin

Caso queira se aprofundar mais nos princípios S.O.L.I.D, seguem alguns links de referência:

https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

https://apiumhub.com/tech-blog-barcelona/solid-principles/

https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/