DESCREVER AS CARACTERÍSTICAS DOS PRINCÍPIOS SOLID

PRINCÍPIOS SOLID

Você já ouviu falar em **SOLID**?

Esse termo é frequentemente encontrado como um conhecimento requisitado em perfis de vagas para desenvolvimento de software.

SOLID é um acrônimo para cinco princípios de projeto orientado a objetos, definidos por Robert C. Martin, que devem ser seguidos para nos ajudar a produzir software com uma estrutura sólida, isto é, uma estrutura que permita sua evolução e manutenção com menor esforço.

Cada letra do acrônimo corresponde a um dos princípios:

S
Single-Responsibility Principle (SRP)

O
Open-closed Principle (OCP)

L
Liskov Substitution Principle (LSP)

I
Interface Segregation Principle (ISP)

D
Dependency Inversion Principle (DIP)

Princípio da Responsabilidade Única (SRP)

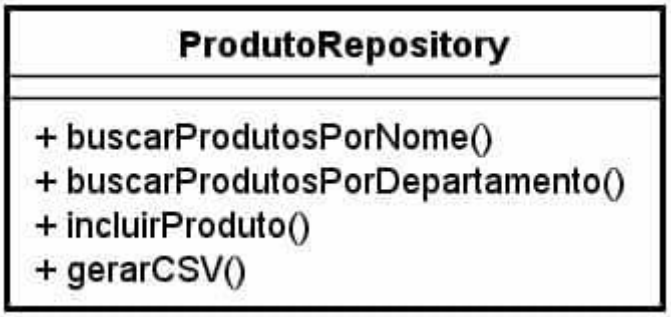
O princípio **SRP** (do inglês ***Single Responsibility Principle***) estabelece que o módulo deve ser responsável por um – e apenas um – ator, representando o papel desempenhado por alguém envolvido com o software. Nesse contexto, um módulo corresponde a um arquivo-fonte.

Exemplo

Em Java, um módulo corresponde a uma classe.

Suponha que a nossa loja virtual tenha requisitos de busca de produtos por nome e por departamento, cadastro de um novo produto e exportação de dados dos produtos em formato CSV. Se pensarmos conforme o padrão Especialista, por exemplo, todas essas

funcionalidades trabalham com os dados dos produtos e, portanto, poderiam ser alocadas ao módulo ProdutoRepository, conforme ilustrado na figura seguinte.



Violação do princípio SRP.

Atenção

Essa alocação de responsabilidades viola o princípio SRP, pois define um módulo que atende a diferentes atores.

Vamos analisar os atores envolvidos com cada responsabilidade do módulo:

Pesquisar produtos por nome e por departamento são demandas do cliente da loja.

Incluir produto é uma demanda da área de vendas.

GerarCSV é uma demanda da área de integração de sistemas.

Portanto, o módulo ProdutoRepository apresenta diferentes razões para mudar: a área de integração pode demandar um novo formato para a exportação dos dados dos produtos, enquanto os clientes podem demandar novas formas de pesquisar os produtos.

De acordo com o princípio SRP, devemos separar essas responsabilidades em três módulos, o que nos daria a flexibilidade de, por exemplo, separar a busca de produtos e a inclusão de produtos em serviços fisicamente independentes, o que possibilitaria atender diferentes demandas de escalabilidade de forma mais racional.

Exemplo

A busca de produtos pode ter que atender a milhares de clientes simultâneos, enquanto esse número, no caso da inclusão de produtos, não deve passar de algumas dezenas de usuários.

Princípio Aberto Fechado (OCP)

O princípio OCP (do inglês ***Open Closed Principle***) estabelece que um módulo deve estar aberto para extensões, mas fechado para modificações, isto é, seu comportamento deve ser extensível sem que seja necessário alterá-lo para acomodar as novas extensões.

Quando uma mudança em um módulo causa uma cascata de modificações em módulos dependentes, isso é sinal de que a estrutura do software não acomoda mudanças adequadamente. Se aplicado de forma adequada, este princípio permite que futuras mudanças desse tipo sejam feitas pela adição de novos módulos, e não pela modificação de módulos já existentes.

Você deve estar pensando como é possível fazer com que um módulo estenda seu comportamento sem que seja necessário mudar uma linha do seu código. A chave para esse enigma chama-se **abstração**. Em uma linguagem orientada a objetos, é possível criar uma interface abstrata que apresente diferentes implementações concretas e, portanto, um novo comportamento, em conformidade com essa abstração, pode ser adicionado simplesmente criando-se um módulo.

O exemplo a seguir apresenta um código que não segue o princípio ***Open Closed***. A classe **CalculadoraGeometrica** contém operações para calcular a área de triângulos e quadrados.

Violação do princípio OCP (clonagem).

```
public class Triangulo {
	private double base;
	private double altura;

	public double getBase() {
		return base;
	}

	public void setBase(double base) {
		this.base = base;
	}

	public double getAltura() {
		return altura;
	}

	public void setAltura(double altura) {
		this.altura = altura;
	}
}

public class Quadrado {
```

```
        double lado;

        public double getLado() {
            return lado;
        }

        public void setLado(double lado) {
            this.lado = lado;
        }
}

public class CalculadoraGeometrica {
        public double obterArea(Quadrado quadrado) {
            return quadrado.getLado() * quadrado.getLado();
        }

        public double obterArea(Triangulo triangulo) {
            return triangulo.getBase() * triangulo.getAltura() / 2;
        }
}
```

Imagine que o programa tenha que calcular a área de um círculo, por exemplo.

O módulo CalculadoraGeometrica poderá calcular a área de um círculo sem que o seu código seja alterado?

Não, pois teremos que adicionar uma nova operação obterArea à CalculadoraGeometrica. Essa é uma forma comum de violação, em que uma classe concentra diversas operações da mesma natureza que operam sobre tipos diferentes.

O exemplo a seguir ilustra outra forma comum de **violação do princípio OCP**.

```
public class FiguraGeometrica {
}

public class Triangulo extends FiguraGeometrica {
    private double base;
    private double altura;

    public double getBase() {
```

```
        return base;
    }

    public void setBase(double base) {
        this.base = base;
    }

    public double getAltura() {
        return altura;
    }

    public void setAltura(double altura) {
        this.altura = altura;
    }
}

public class Quadrado extends FiguraGeometrica {
    double lado;

    public double getLado() {
        return lado;
    }

    public void setLado(double lado) {
        this.lado = lado;
    }
}

public class CalculadoraGeometrica {
    public double obterArea(FiguraGeometrica figura) {
        if (figura instanceof Quadrado) {
            return obterAreaQuadrado((Quadrado) figura);
        } else if (figura instanceof Triangulo) {
            return obterAreaTriangulo((Triangulo) figura);
        } else {
            return 0.0;
        }
```

```
    }

    private double obterAreaQuadrado(Quadrado quadrado) {
        return  quadrado.getLado() * quadrado.getLado();
    }

    private double obterAreaTriangulo(Triangulo triangulo) {
        return  triangulo.getBase() * triangulo.getAltura() / 2;
    }
}
```

Neste caso, a classe CalculadoraGeometrica possui uma operação obterArea, que recebe um tipo genérico FiguraGeometrica, mas continua concentrando a lógica de cálculo para todos os tipos de figuras. Para incluir um círculo, precisaríamos adicionar a operação obterAreaCirculo e modificar a implementação da operação obterArea, inserindo uma nova condição. Imagine como ficaria o código dessa operação se ela tivesse que calcular a área de cinquenta tipos diferentes de figuras geométricas!

O que devemos fazer para que o módulo CalculadoraGeometrica possa estar aberto para trabalhar com novos tipos de figuras sem precisarmos alterar o seu código?

A resposta está no emprego da abstração e, neste exemplo, dos princípios do Especialista e do Polimorfismo.

Para isso, definimos que toda figura geométrica é capaz de calcular sua área, sendo que a fórmula de cálculo de cada figura específica deve ser implementada nas realizações concretas de FiguraGeometrica.

Dessa forma, para adicionar uma nova figura geométrica, basta adicionar um novo módulo com uma implementação específica para essa operação.

O próximo exemplo ilustra a nova estrutura do projeto.

A classe FiguraGeometrica define uma operação abstrata obterArea. Aplicando o princípio do Especialista, cada subclasse fica responsável pelo cálculo da sua área. Aplicando o princípio do Polimorfismo, o módulo CalculadoraGeometrica passa a depender apenas da abstração FiguraGeometrica, que define que todo elemento desse tipo é capaz de calcular a sua área. Dessa forma, a calculadora geométrica pode funcionar com novos tipos de figuras, sem que seja necessário alterar o seu código, uma vez que ela deixou de depender da forma específica da figura.

Reestruturação adequada ao princípio OCP.

```
public abstract class FiguraGeometrica {
    public abstract double obterArea();
```

```
}

public class Triangulo extends FiguraGeometrica {
    private double base;
    private double altura;

    public double obterArea() {
        return  this.getBase() * this.getAltura() / 2;
    }

    public double getBase() {
        return base;
    }
    public void setBase(double base) {
        this.base = base;
    }
    public double getAltura() {
        return altura;
    }
    public void setAltura(double altura) {
        this.altura = altura;
    }
}

public class Quadrado extends FiguraGeometrica {
    double lado;

    public double obterArea() {
        return  this.getLado() * this.getLado();
    }
    public double getLado() {
        return lado;
    }
    public void setLado(double lado) {
        this.lado = lado;
    }
}
```

```
public class CalculadoraGeometrica {
    public double obterArea(FiguraGeometrica figura) {
        return figura.obterArea();
```

Princípio da Substituição de Liskov (LSP)

O princípio LSP (do inglês ***Liskov Substitution Principle***) foi definido em 1988 por Barbara Liskov e estabelece que um tipo deve poder ser substituído por qualquer um de seus subtipos sem alterar o correto funcionamento do sistema.

O exemplo seguinte apresenta um caso clássico de violação do princípio de Liskov. Na geometria, um quadrado pode ser visto como um caso particular de um retângulo. Se levarmos essa propriedade para a implementação em software, podemos definir uma classe Quadrado como uma extensão da classe Retangulo.

Atenção

Note que, como a largura do quadrado é igual ao comprimento, a implementação dos métodos de acesso (setLargura e setComprimento) do quadrado deve se preocupar em preservar essa propriedade.

Princípio LSP (Quadrado)

```
public class Retangulo {
    private double largura;
    private double comprimento;

    public double area() {
        return largura * comprimento;
    }

    public double getLargura() {
        return largura;
    }
    public void setLargura(double largura) {
        this.largura = largura;
    }
    public double getComprimento() {
        return comprimento;
    }
    public void setComprimento(double comprimento) {
```

```
        this.comprimento = comprimento;
    }
}

public class Quadrado extends Retangulo {
    public void setLargura(double largura) {
        super.setLargura(largura);
        super.setComprimento(largura);
    }
    public void setComprimento(double largura) {
        super.setLargura(largura);
        super.setComprimento(largura);
    }
}
```

Porém, o próximo exemplo mostra como essa solução viola o princípio da substituição de Liskov. O método verificarArea da classe ClienteRetangulo recebe um objeto r do tipo Retangulo.

De acordo com esse princípio, o código desse método deveria funcionar com qualquer objeto de um tipo derivado de Retangulo. Entretanto, no caso de um objeto do tipo Quadrado, a execução da chamada setComprimento fará com que os dois atributos do quadrado passem a ter o valor 10. Em seguida, a execução da chamada setLargura fará com que os dois atributos do quadrado passem a ter o valor 8 e, portanto, o valor retornado pelo método área definido na classe Retangulo será 64 e não 80, violando o comportamento esperado para um retângulo.

Violação do princípio LSP

```
public class ClienteRetangulo {
    public void verificarArea(Retangulo r) throws Exception {
        r.setComprimento(10);
        r.setLargura(8);
        if (r.area() != 80) {
            throw new Exception("area incorreta");
        }
    }
}
```

Esse caso ilustra que, embora um quadrado possa ser classificado como um caso particular de um retângulo, o princípio de Liskov estabelece que um subtipo não pode estabelecer restrições adicionais que violem o comportamento genérico do supertipo. Nesse caso, o fato de o quadrado exigir que comprimento e largura sejam iguais o torna

mais restrito que o retângulo. Portanto, definir a classe Quadrado como um subtipo de Retangulo é uma solução inapropriada para este problema, do ponto de vista da orientação a objetos, por violar o princípio de Liskov.

Princípio da Segregação de Interfaces (ISP)

O princípio ISP (do inglês ***Interface Segregation Principle***) estabelece que clientes de uma classe não devem ser forçados a depender de operações que eles não utilizem.

O código do exemplo a seguir apresenta uma interface que viola esse princípio por reunir operações que dificilmente serão utilizadas em conjunto. As operações login e registrar estão ligadas ao registro e à autorização de acesso de um usuário, enquanto a operação logErro registra mensagens de erro ocorridas durante algum processamento, e a operação enviarEmail permite enviar um e-mail para o usuário logado.

Violação do princípio ISP

```
public interface IUsuario {
    boolean login(String login, String senha);
    boolean registrar(String login, String senha, String email);
    void logErro(String msgErro);
    boolean enviarEmail(String assunto, String conteudo);
}
```

Segundo este princípio, devemos manter a coesão funcional das interfaces, evitando colocar operações de diferentes naturezas em uma única interface. Aplicando ao exemplo, poderíamos segregar a interface original em três interfaces (exemplo a seguir):

- **IAutorizacao**, com as operações de login e registro de usuários;
- **IMensagem**, com as operações de registro de erro, aviso e informações;
- **IEmail** para o envio de e-mail.

Segregação das interfaces

```
public interface IAutorizacao {
    boolean login(String login, String senha);
    boolean registrar(String login, String senha, String email);
}

public interface IEmail {
    boolean enviarEmail(String assunto, String conteudo);
}
```

```
public interface IMensagem {
    void logErro(String msgErro);
    void logAviso(String msgAviso);
    void logInfo(String msgInfo);
}
```

Princípio da Inversão de Dependências (DIP)

O princípio DIP (do inglês ***Dependency Inversion Principle***) estabelece que as entidades concretas devem depender de abstrações e não de outras entidades concretas. Isso é especialmente importante quando estabelecemos dependências entre entidades de camadas diferentes do sistema.

Segundo este princípio, um módulo de alto nível não deve depender de um módulo de baixo nível.

Módulos de alto nível

São aqueles ligados estritamente ao domínio da aplicação (ex.: Loja, Produto, ServiçoVenda etc.).

Módulos de baixo nível

São ligados a alguma tecnologia específica (ex.: acesso a banco de dados, interface com o usuário etc.) e, portanto, mais voláteis.

O próximo exemplo apresenta um caso de dependência de um módulo de alto nível em relação a um módulo de baixo nível. A classe ServicoConsultaProduto depende diretamente da implementação de um repositório de produtos que acessa um banco de dados relacional Oracle.

Essa dependência é estabelecida na operação obterPrecoProduto da classe ServicoConsultaProduto pelo comando *new ProdutoRepositoryOracle()*.

Violação do princípio da Inversão de Dependências

```
public class ProdutoRepositoryOracle {
    public Produto obterProduto(String codigo) {
        Produto p = new Produto();
        // implementacao SQL de recuperação do produto
        return p;
    }
```

```
    public void  salvarProduto(Produto produto) {
        // implementacao SQL de insert ou update do produto
    }
}

public class ServicoConsultaProduto {
    public double obterPrecoProduto(String codigo) {
        ProdutoRepositoryOracle repositório
          = new ProdutoRepositoryOracle();
        Produto produto = repositorio.obterProduto(codigo);
        return produto.getPreco();
    }
}
```

A figura a seguir apresenta, em um diagrama UML, a relação de dependência existente entre a classe concreta ServicoConsultaProduto e a classe concreta ProdutoRepositoryOracle.

Violação do DIP (Diagrama UML).

Por que essa dependência é inadequada?

Como aspectos tecnológicos são voláteis, devemos isolar a lógica de negócio, permitindo que a implementação concreta desses aspectos possa variar sem afetar as classes que implementam a lógica de negócio.

O exemplo a seguir apresenta uma solução resultante da aplicação deste princípio. A interface abstrata (ProdutoRepository) define um contrato abstrato entre cliente (ServicoConsultaProduto) e fornecedor (ProdutoRepositoryOracle). Em vez de o cliente depender de um fornecedor específico, ambos passam a depender dessa interface abstrata.

Assim, a classe ServicoConsultaProduto pode trabalhar com qualquer implementação concreta da interface ProdutoRepository, sem que seu código precise ser alterado.

```
public interface ProdutoRepository {
    Produto obterProduto(String codigo);
    void  salvarProduto(Produto produto);
}
```

```
public class ProdutoRepositoryOracle implements ProdutoRepository {
    public Produto obterProduto(String codigo) {
        Produto p = new Produto();
        // implementacao SQL de recuperação do produto
        return p;
    }

    public void  salvarProduto(Produto produto) {
        // implementacao SQL de insert ou update do produto
    }
}

public class ServicoConsultaProduto {
    ProdutoRepository repositorio;

    public ServicoConsultaProduto(ProdutoRepository repositorio) {
                this.repositorio = repositorio;
    }
    public double obterPrecoProduto(String codigo) {
        Produto produto = repositorio.obterProduto(codigo);
        return produto.getPreco();
    }
}
```

A figura a seguir apresenta o diagrama UML correspondente à nova estrutura da solução após a aplicação desse princípio. A classe ServicoConsultaProduto depende apenas da interface ProdutoRepository, enquanto a classe ProdutoRepositoryOracle é uma das possíveis implementações concretas dessa interface.