CENTRO UNIVERSITÁRIO DE DESENVOLVIMENTO DO CENTRO-OESTE
UNIDESC

BACHAREL EM SISTEMA DE INFORMAÇÃO

Arquitetura de Software
Teoria e Fundamentos
SOLID

ROGÉRIO PEREIRA DE SOUZA

LUZIÂNIA
2024

RESUMO

Neste trabalho vou falar sobre solid e factory como isso pode ser usado em uma arquitetura de e como ele pode resolver problemas com o seu código.

SOLID: SOLID é um acrônimo para cinco princípios de design orientado a objetos e programação que tornam o software mais compreensível, flexível e sustentável. São eles: Princípio da Responsabilidade Única (SRP), Princípio Aberto/Fechado (OCP), Princípio de Substituição de Liskov (LSP), Princípio de Segregação de Interface (ISP) e Princípio de Inversão de Dependência (DIP).

Arquitetura de Software: A Arquitetura de Software é como o esqueleto de um sistema. Ela é composta por várias partes - os componentes do software - que são visíveis externamente. Esses componentes se relacionam entre si de maneiras específicas para formar a estrutura completa. É como um plano mestre para o sistema de software.

Padrão Factory: O padrão Factory, também conhecido como Factory Method, é um padrão de design criacional que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados. O padrão Factory é usado quando temos uma superclasse com várias subclasses e com base na entrada, precisamos retornar uma das subclasses. Este padrão tira a responsabilidade da criação de objetos da classe cliente e a coloca na fábrica.

SUMARIO

SUMARIO DE FIGURAS

TEORIA E FUNDAMENTOS:

SOLID.

O SOLID e um acrônimo para cinco precipícios relacionados a programação orientada a objeto (POO) que são Single **Responsibility Principle** (Princípio da responsabilidade única), **Open-Closed Principle** (Princípio Aberto-Fechado**), Liskov Substitution Principle** (Princípio da substituição de Liskov), **Interface Segregation Principle** (Princípio da Segregação da Interface), **Dependency Inversion Principle** (Princípio da inversão da dependência) estes princípios foram definidos por Os princípios SOLID foram definidos por Robert C. Martin e vou falar um pouco sobre cada um deles aqui abaixo.

Single Responsibility Principle (Princípio da responsabilidade única):

Este princípio diz que uma classe só pode ter uma única responsabilidade executando uma única responsabilidade ou tarefa.
Isso significa que cada classe deve ser especializada em uma única tarefa ou função dentro do software. Ao seguir esse princípio, evitamos sobrecarregar uma classe com muitas responsabilidades.

Por exemplo, se uma classe é responsável por processar dados e também por exibir esses dados, ela tem duas responsabilidades. De acordo com o Princípio da Responsabilidade Única, essas responsabilidades devem ser divididas em duas classes separadas.

Esse princípio facilita a compreensão, o desenvolvimento e a manutenção do software, pois torna o código mais organizado, modular e fácil de entender. Com isso o código ira ficar menos propenso a erros.

Exemplo de aplicação Single Responsibility Principle:

Como podemos ver no código baixo, as classes LivroDAO que é usada para salvar e LivroImpresão que é usada apenas para salvar.

```
public class Livro {
    private String autor;
    private String titulo;
```

```
    // Construtor, getters e setters
}

public class LivroDAO {
    public void salvar(Livro livro) {
        // código para salvar o livro no banco de dados
    }
}

public class LivroImpressao {
    public void imprimir(Livro livro) {
        // código para imprimir o livro
    }
}
```

Open-Closed Principle (Princípio Aberto-Fechado):

Este princípio diz que “entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação”. Isso significa que você deve ser capaz de adicionar novas funcionalidades ou comportamentos a uma classe sem alterar o código existente. Em outras palavras, o design de uma classe deve ser feito de tal maneira que possa permitir que seu comportamento seja estendido, sem a necessidade de modificar seu código-fonte. Isso é geralmente alcançado usando a herança e/ou interfaces em linguagens de programação orientadas a objetos. O benefício deste princípio é que ele ajuda a evitar problemas que podem surgir devido a mudanças no código existente aumentado assim a robustez do código.

Exemplo de aplicação Open-Closed Principle:

Neste princípio as classes devem estar abertas para extensões porem fedas para seres modificadas, no código abaixo foi implementado o método desenhar para aderir ao princípio de aberto e fechado.

```
public abstract class Forma {
    abstract void desenhar();
}

public class Circulo extends Forma {
    void desenhar() {
        // Desenha o círculo
    }
}

public class Desenho {
    void desenharFormas(List<Forma> formas) {
        for (Forma forma : formas) {
            forma.desenhar();
        }
    }
}
```

Liskov Substitution Principle (Princípio da substituição de Liskov):

O LSP afirma que “se S é um subtipo de T, então objetos do tipo T podem ser substituídos por objetos do tipo S sem alterar nenhuma das propriedades desejáveis do programa” e foi introduzido no SOLID em 1987 por Barbara Liskov.

Em termos mais simples, isso significa que qualquer classe (ou módulo ou função) que é uma subclasse de uma superclasse, deve poder ser usada no lugar da superclasse sem que isso resulte em um comportamento inesperado.

Por exemplo, se temos uma classe “Pássaro” e uma subclasse “Pinguim”, e a classe “Pássaro” tem um método “voar”, não podemos usar a classe “Pinguim” no lugar da classe “Pássaro” porque pinguins não voam. Isso violaria o Princípio da Substituição de Liskov.O LSP é fundamental para a reutilização de código. Ele permite que os programadores usem hierarquias de classes polimórficas para alcançar a

abstração, um dos principais benefícios da programação orientada a objetos. Quando o LSP é violado, pode levar a problemas de design e bugs no código.

Como já mencionado acima no exemplo abaixo temos a classe PASSARO para definir os pássaros de modo geral e com isso podemos incluir os Pinguins por exemplo e a classe PassaroVoador para definir apenas pássaros que podem voar.

```
// Classe base ou superclasse
public class Pássaro {
    // Comportamentos comuns a todos os pássaros podem ser definidos aqui
}

// Subclasse de Pássaro para pássaros que podem voar
public class PássaroVoador extends Pássaro {
    void voar() {
        System.out.println("O pássaro está voando.");
    }
}
// Subclasse de Pássaro para pinguins
public class Pinguim extends Pássaro {
    // Pinguins não podem voar, então não há método voar() aqui
}
public class Main {
    public static void main(String[] args) {
        PássaroVoador pássaroVoador = new PássaroVoador();
        pássaroVoador.voar(); // Isso funcionará corretamente
        Pássaro pássaro = new Pinguim();
        // pássaro.voar(); // Isso agora é um erro de compilação, não um erro em tempo de execução
    }
}
```

Interface Segregation Principle (Princípio da Segregação da Interface):

O **Princípio da Segregação da Interface** (ISP) é um dos princípios SOLID na programação orientada a objetos. Ele afirma que “nenhum cliente deve ser forçado a depender de interfaces que não usa” ou seja as classes devem usar apenas o que faz sentido para ela.

Isso significa que, em vez de ter uma interface grande com muitos métodos, é melhor ter várias interfaces menores, cada uma com um propósito específico. Cada classe deve implementar apenas as interfaces e os métodos que realmente usa.

Por exemplo, se temos uma interface **`Animal`** com os métodos **comer()**, **voar()** e **nadar()**, e uma classe **Pinguim** que implementa essa interface, a classe **Pinguim** seria forçada a implementar o método `voar()`, mesmo que pinguins não voem. Isso violaria o ISP.

Para seguir o ISP, poderíamos dividir a interface `Animal` em várias interfaces menores, como AnimalTerrestre, AnimalVoador e AnimalNadador. A classe Pinguim poderia então implementar apenas as interfaces AnimalTerrestre e AnimalNadador.

O ISP ajuda a manter o sistema desacoplado e fácil de refatorar, modificar e entender. Ele promove a coesão, pois cada interface e classe tem um propósito bem definido. Além disso, ele melhora a robustez do código, pois as alterações em uma interface ou classe têm menos probabilidade de afetar outras partes do sistema.

Neste exemplo do código abaixo podemos ver o princípio citado acima que nem um cliente e obrigado a usar aquilo que não precisa por isso em vez de ter uma única classe animal que teria os métodos comer (), nadar() e voar() se tem a classe pinguim que tem os métodos comedor() e nadador () e a classe pássaro com os métodos comedor e voador para assim assegurar que não teremos um métodos que não será usado pela classe.

```
// Interfaces segregadas
interface Comedor {
    void comer();
}
```

```
interface Nadador {
    void nadar();
}

interface Voador {
    void voar();
}

// Classe Pinguim implementa as interfaces que usa
class Pinguim implements Comedor, Nadador {
    public void comer() {
        System.out.println("O pinguim está comendo.");
    }

    public void nadar() {
        System.out.println("O pinguim está nadando.");
    }
}

// Classe Pássaro implementa as interfaces que usa
class Pássaro implements Comedor, Voador {
    public void comer() {
        System.out.println("O pássaro está comendo.");
    }

    public void voar() {
        System.out.println("O pássaro está voando.");
    }
}
```

Dependency Inversion Principle (Princípio da inversão da dependência):

Este princípio se baseia em que as ambas as classes tanto de alto nivel quando as classes de baixo nivel dependa da abstraçao e não uma da outra, ambos devem depender de abstrações. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações".

Isso significa que, em vez de ter classes de alto nível (aquelas que contêm a lógica de negócios) dependendo de classes de baixo nível (aquelas que realizam tarefas específicas), ambas as classes de alto e baixo nível devem depender de abstrações (como interfaces ou classes abstratas).

Por exemplo, se temos uma classe Aplicativo que usa uma classe BancoDeDados, em vez de a classe Aplicativo depender diretamente da classe BancoDeDados, ambas as classes poderiam depender de uma interface IBancoDeDados.

O DIP ajuda a reduzir o acoplamento entre as classes, tornando o sistema mais modular e fácil de modificar e testar. Ele promove a flexibilidade e a robustez do código, pois as alterações em uma classe de baixo nível têm menos probabilidade de afetar as classes de alto nível.

Neste exemplo vemos que a classe de alto nível não depende da classe debaixo nível e sim da abstração, no código abaixo temo a classe de alto nível que que não depende da classe de baixo nível carro e sim que as duas classes tanto piloto como carro dependem da abstração veiculo.

```
// Interface de abstração
interface Veículo {
    void acelerar();
}

// Classe de baixo nível
class Carro implements Veículo {
    public void acelerar() {
```

```
        System.out.println("O carro está acelerando.");
    }
}

// Classe de alto nível
class Piloto {
    private Veículo veículo;

    // Injeção de dependência através do construtor
    Piloto(Veículo veículo) {
        this.veículo = veículo;
    }

    void dirigir() {
        veículo.acelerar();
    }
}

public class Main {
    public static void main(String[] args) {
        Veículo carro = new Carro();
        Piloto piloto = new Piloto(carro);
        piloto.dirigir(); // Saída: O carro está acelerando.
    }
}
```

ARQUITETURA DE SOFTWARE

Pense na Arquitetura de Software como o esqueleto de um sistema. Ela é composta por várias partes - os componentes do software - que são visíveis externamente. Esses componentes se relacionam entre si de maneiras específicas para formar a estrutura completa. É como um plano mestre para o sistema de software.

Agora, esses componentes não são apenas blocos aleatórios. Eles são partes significativas do sistema, organizadas de uma maneira particular. Eles interagem entre si e seguem certos padrões que orientam como eles são compostos juntos. E o resultado? Um sistema que funciona como uma unidade coesa.

Esses componentes podem ser módulos de software, funções, interfaces e dados. Juntos, eles atendem às necessidades de negócios do sistema. E a Arquitetura de Software é o que garante que o software funcione bem, seja resiliente, reutilizável, compreensível, escalável e seguro, entre outras coisas.

Vamos pegar um exemplo simples: a Arquitetura em Camadas. Imagine que você tem:

A Interface do Usuário (UI): é a camada que conversa diretamente com o usuário. Pode ser uma interface gráfica bonita ou uma interface de linha de comando simples.

A Lógica de Negócio (Service): aqui é onde a mágica acontece. Esta camada contém o código que lida com as regras e processos do domínio do problema.

O Acesso a Dados (DAO): esta camada é como um intermediário. Ela lida com a persistência e recuperação de dados, abstraindo os detalhes do acesso ao banco de dados.

O Banco de Dados: é aqui que os dados são armazenados. Pode ser um banco de dados relacional, um banco de dados NoSQL, um sistema de arquivos, etc.

A beleza disso é que cada camada depende apenas da camada imediatamente abaixo dela. Isso significa que se algo mudar em uma camada, as outras camadas não serão afetadas. Isso torna o sistema mais fácil de manter e atualizar.

APLICAÇÃO PRATICA DE UM PROJETO USANDO FACTORY:

Na verdade, o que chamamos de padrão Factory na verdade são dois padrões distintos: o Factory Method e o Abstract Factory. Esses padrões têm como principal objetivo nos ajudar a reduzir o acoplamento em nosso software. Eles mantêm as dependências flexíveis e evitam que sejam explícitas. Como o Uncle Bob nos lembra: “A abstração não deve depender de detalhes, os detalhes é que devem depender de abstrações”. Hoje, vamos falar sobre o padrão Factory Method.

Por que usar o Factory Method? Bem, como mencionado antes, o objetivo é tornar o software mais flexível, removendo as dependências explícitas. O grande problema surge quando instanciamos diretamente um objeto, pois estamos criando uma dependência explícita para esse objeto específico.

À primeira vista, pode parecer inofensivo. Mas quando se trata de um software mais complexo com uma hierarquia, por exemplo, essa simples instanciação estabeleceria uma dependência concreta e custosa de remover. Se fosse necessário mudar de JogadorFutebol para JogadorBasquete, teríamos um problema para alterar o software. É aí que entra o Factory Method. Em uma arquitetura, ele se torna o responsável por criar determinados objetos, reduzindo assim o acoplamento e até nos ajudando a aplicar o princípio da responsabilidade única.

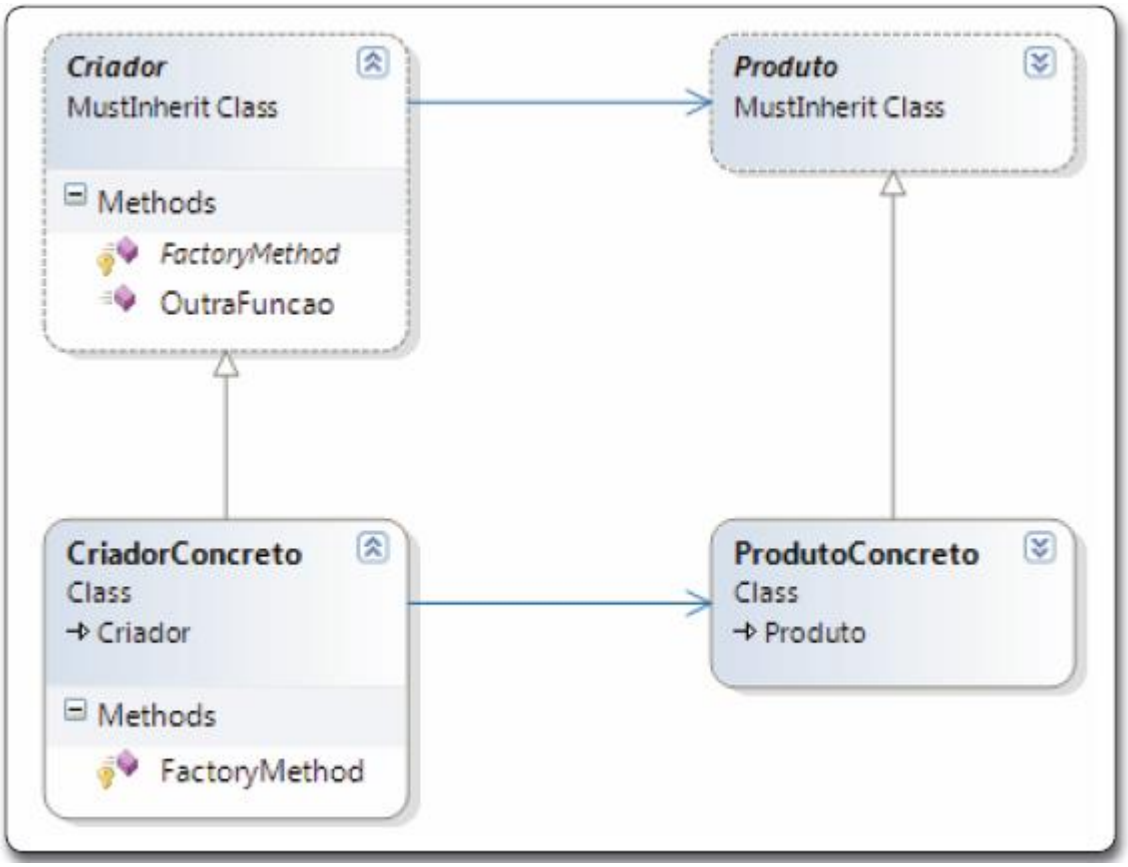

Figura 1 Diagra,a Factory https://www.devmedia.com.br/patterns-factory-method/18954

Como visto acima, temos uma classe base abstrata(criadora), que contém um método abstrato(factory method) que cria um determinado objeto(produto). Utilizando o Factory Method, você cria uma interface para os clientes utilizarem, mais deixa que eles, a responsabilidade de decidir qual classe concreta instanciar seguindo como no exemplo abaixo.

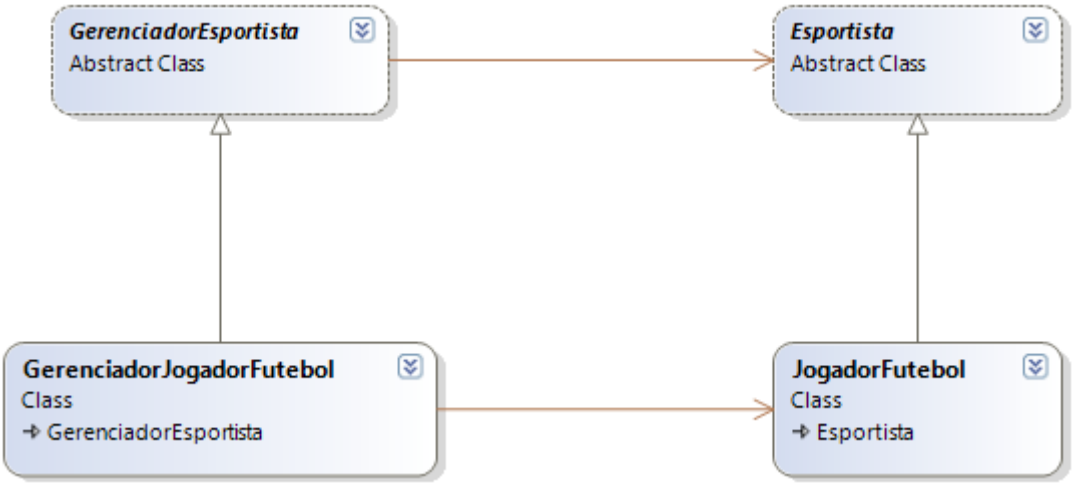

Figura 2 https://www.devmedia.com.br/patterns-factory-method/18954

Figura

Como neste exemplo de implementação no código:

```
abstract class GerenciadorEsportista
{
    public abstract Esportista CriaEsportista();
}

abstract class Esportista
{

}
```

Factory Method

Figura 3 implementação do código https://www.devmedia.com.br/patterns-factory-method/18954

```
public class GerenciadorJogadorFutebol : GerenciadorEsportista
{
    public override Esportista CriaEsportista()
    {
        return new JogadorFutebol();
    }
}

public class JogadorFutebol : Esportista
{
}
```

Figura 4 concreta https://www.devmedia.com.br/patterns-factory-method/18954

Imagine que você está montando um quebra-cabeça. Cada peça do quebra-cabeça é como uma classe no padrão Factory Method. Algumas peças (as subclasses) são responsáveis por decidir como criar outras peças (os produtos concretos).

A beleza disso é que se você quiser mudar uma peça, você só precisa criar uma nova subclasse. O resto do quebra-cabeça (a base) permanece o mesmo. Isso torna o quebra-cabeça (ou seja, seu software) mais fácil de ajustar e manter.

Em termos técnicos, o Factory Method é como um plano para criar objetos. Ele permite que as subclasses decidam qual classe concreta instanciar. Isso dá mais flexibilidade ao seu software, pois a criação de objetos pode ser adiada para as subclasses.

VER COMO PODEMOS APLICAR O FACTORY:

Imagine que temos uma fábrica de carros, a CarroFactory. Ela é especialista em criar diferentes tipos de carros - Sedan e SUV. Quando alguém (neste caso, a classe cliente Main) precisa de um carro, eles simplesmente dizem à fábrica que tipo de carro eles querem. A fábrica então vai em frente e cria o carro para eles. Isso mantém as coisas simples para a classe cliente, que pode se concentrar em suas próprias responsabilidades sem se preocupar com os detalhes de como os carros são criados.

Agora, digamos que no futuro queremos adicionar novos tipos de carros à nossa fábrica. Desde que esses novos carros sigam a mesma interface Carro, a classe cliente não precisará mudar nada. Ela ainda poderá pedir à fábrica um carro, sem se preocupar com o tipo específico de carro que está sendo criado.

Isso é o que torna o padrão Factory tão útil. Ele nos ajuda a manter nosso código organizado, flexível e fácil de manter, mesmo quando nosso sistema continua a crescer e evoluir.

```
// Interface do produto
interface Carro {
    void exibirInfo();
}

// Produtos Concretos
class Sedan implements Carro {
    public void exibirInfo() {
        System.out.println("Este é um carro Sedan.");
    }
}

class SUV implements Carro {
    public void exibirInfo() {
        System.out.println("Este é um carro SUV.");
    }
}

// Factory
class CarroFactory {
    static Carro getCarro(String tipo) {
        if (tipo == null) {
            return null;
        }
        if (tipo.equalsIgnoreCase("SEDAN")) {
            return new Sedan();
        } else if (tipo.equalsIgnoreCase("SUV")) {
            return new SUV();
        }
        return null;
    }
}
```

```
// Classe cliente
public class Main {
    public static void main(String[] args) {
        Carro carro = CarroFactory.getCarro("SUV");
        carro.exibirInfo(); // Saída: Este é um carro SUV.
    }
}
```

INTEGRAÇÃO COM SOLID E ARQUITETURA DE SOFTWARE:

SRP (Princípio da Responsabilidade Única): Cada classe no padrão Factory tem uma única responsabilidade. Por exemplo, a classe Sedan é responsável por implementar a interface Carro e definir o comportamento do método exibirInfo() para um Sedan. Da mesma forma, a classe CarroFactory tem a única responsabilidade de criar objetos Carro.

OCP (Princípio Aberto/Fechado): A Factory permite que novos tipos de produtos sejam criados sem modificar o código existente. Se quisermos adicionar um novo tipo de carro, como um Caminhao, só precisamos criar uma nova classe Caminhao que implementa a interface Carro e modificar a CarroFactory para criar um Caminhao quando necessário.

LSP (Princípio de Substituição de Liskov): Os produtos criados pela Factory podem ser usados de forma intercambiável. No código do cliente (Main), podemos usar qualquer objeto Carro retornado pela CarroFactory, seja ele um Sedan ou um SUV.

ISP (Princípio de Segregação de Interface): A Factory não impõe nenhuma interface desnecessária aos clientes. A CarroFactory retorna um Carro, que é a única interface que o cliente precisa conhecer.

DIP (Princípio de Inversão de Dependência): A Factory depende de abstrações (a interface do produto), não de concretizações. A CarroFactory não

depende de classes concretas como Sedan ou SUV; em vez disso, ela depende da interface Carro.

Portanto, o padrão Factory está bem alinhado com os princípios SOLID, ajudando a criar um código mais limpo, mais modular e mais fácil de manter.

Bibliografia.

SEM AUTOR. O que é SOLID?. Luby Software, s.d. Disponível em: https://luby.com.br/desenvolvimento/software/o-que-e-solid/. Acesso em: 02-jun-2024

DEVMEDIA. **Arquitetura de Software: Desenvolvimento Orientado para Arquitetura**. Disponível em: Arquitetura de Software: Desenvolvimento Orientado para Arquitetura (devmedia.com.br). Acesso em: 8 jun. 2024.

DEVMEDIA. **Patterns: Factory Method**. Disponível em: Patterns: Factory Method (devmedia.com.br). Acesso em: 8 jun. 2024.