PADRÕES DE PROJETO CRIACIONAIS

ACADÊMICOS:

S. Antônio Jr

Emanoel Leffa Mittmann de Oliveira

Joao Manoel Dias Pereira

Maria Cicera Ferreira dos Santos Ribeiro

Tiago Boff do Nascimento

Lorenzo Da Cunha Cardoso

PADROES CRIACIONAIS

1

O QUE SÃO

2

PARA QUE SERVEM

3

ONDE USAR

FACTORY METHOD

1. É um pattern que funciona apartir de uma base classe e um contrato de um objeto, que sera criado apartir da implementação da interface base

2. Permite a flexibilidade de criação de novos objetos

3. É muito utilizado quando não se sabe qual tipos exatos e dependências que se vai trabalhar

FACTORY METHOD CÓDIGO

```typescript
abstract class Creator {
    public abstract factoryMethod(): Transport

    public Message(): string {
        const transport = this.factoryMethod()
        return `Creator: ${transport.delivery()}`
    }
}

interface Transport {
    delivery(): string
}
```

```typescript
class Truck implements Transport {
    delivery() {
        return 'New Truck'
    }
}
```

```typescript
class Car implements Transport {
    delivery() {
        return 'New Car'
    }
}
```

FACTORY METHOD CÓDIGO

```
import construtor from "./construtor";

const {clientCode ,...props} = construtor

function Main(){
    return clientCode(new props.CreatorMotocycle)
}

Main()
```

```
function clientCode(creator : Creator){
    return console.log(creator.Message())

}
```

```
class CreatorTruck extends Creator{

    public factoryMethod(): Transport {
        return new Truck;
    }
}

class CreatorCar extends Creator{

    public factoryMethod(): Transport {
        return new Car;
    }
}
```

```
Emanoel@DESKTOP-Q3O2G9R MINGW64 ~/Desktop/factories
$ ts-node factoryMethod
Creator: New Motocycle
```

ABSTRACT FACTORY

1. Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas

2. Permite criar famílias de objetos relacionados, garantindo que os objetos criados sejam compatíveis entre si.

3. Oferece flexibilidade e extensibilidade na criação de objetos.

ABSTRACT CONTRAS

1. Pode aumentar o acoplamento entre as classes do código cliente e as classes de fábrica abstrata.

2. Pode envolver a criação de hierarquias de fábricas, onde existem várias camadas de abstração para diferentes famílias de objetos, isso pode levar a um aumento da complexidade do código.

3. Pode resultar em um aumento do número de classes no sistema.

ABSTRACT FACTORY CODIGO

```
namespace ExemploAbstractFactory.Domain.Interface
{
    public interface IAbstractFactory
    {
        IProductA CreateProductA();
        IProductB CreateProductB();
    }
}
```

ABSTRACT FACTORY CODIGO

```
namespace ExemploAbstractFactory.Domain.Factorys
{
    public class ConcreteFactoryA : IAbstractFactory
    {
        public IProductA CreateProductA()
        {
            return new ConcreteProductA1();
        }

        public IProductB CreateProductB()
        {
            return new ConcreteProductB1();
        }
    }
}
```

```
namespace ExemploAbstractFactory.Domain.Factorys
{
    public class ConcreteFactoryB : IAbstractFactory
    {
        public IProductA CreateProductA()
        {
            return new ConcreteProductA2();
        }

        public IProductB CreateProductB()
        {
            return new ConcreteProductB2();
        }
    }
}
```

ABSTRACT FACTORY CODIGO

```
namespace ExemploAbstractFactory.Domain.Interface
{
    public interface IProductA
    {
        void OperationA();
    }
}
```

```
namespace ExemploAbstractFactory.Domain.Interface
{
    public interface IProductB
    {
        void OperationB();
    }
}
```

ABSTRACT FACTORY CODIGO

```
namespace ExemploAbstractFactory.Domain.Factorys
{
    public class ConcreteProductA1 : IProductA
    {
        public void OperationA()
        {
            Console.WriteLine("ConcreteProductA1: OperationA");
        }
    }
}
```

```
namespace ExemploAbstractFactory.Domain.Factorys
{
    public class ConcreteProductA2 : IProductA
    {
        public void OperationA()
        {
            Console.WriteLine("ConcreteProductA2: OperationA");
        }
    }
}
```

ABSTRACT FACTORY CODIGO

```
namespace ExemploAbstractFactory.Domain.Factorys
{
    public class ConcreteProductB1 : IProductB
    {
        public void OperationB()
        {
            Console.WriteLine("ConcreteProductB1: OperationB");
        }
    }
}
```

```
namespace ExemploAbstractFactory.Domain.Factorys
{
    public class ConcreteProductB2 : IProductB
    {
        public void OperationB()
        {
            Console.WriteLine("ConcreteProductB2: OperationB");
        }
    }
}
```

ABSTRACT FACTORY CODIGO

```
namespace ExemploAbstractFactory.Domain
{
    public class Client
    {
        private IProductA productA;
        private IProductB productB;

        public Client(IAbstractFactory factory)
        {
            productA = factory.CreateProductA();
            productB = factory.CreateProductB();
        }

        public void Run()
        {
            productA.OperationA();
            productB.OperationB();
        }
    }
```

```
public class Program
{
    public static void Main(string[] args)
    {
        IAbstractFactory factoryA = new ConcreteFactoryA();
        Client clientA = new Client(factoryA);
        clientA.Run();

        Console.WriteLine();

        IAbstractFactory factoryB = new ConcreteFactoryB();
        Client clientB = new Client(factoryB);
        clientB.Run();

        Console.ReadKey();
    }
}
```

ABSTRACT FACTORY CODIGO

```
PS C:\Users\joabi\Desktop\Trabalho
ConcreteProductA1: OperationA
ConcreteProductB1: OperationB

ConcreteProductA2: OperationA
ConcreteProductB2: OperationB
```

BUILDER

1. É um padrão de projeto de software que oferece uma forma de construir objetos complexos passo a passo

2. Ele permite a criação de tipos e representações de construir um objeto usando o mesmo código de construção.

3. Ele separa a construção de um objeto de sua representação, permitindo que o mesmo processo de construção possa criar diferentes representações do objeto.

BUILDER CONTRAS

1. O uso do padrão Builder pode introduzir uma camada adicional de complexidade no código

2. A implementação do padrão Builder geralmente requer a criação de várias classes e métodos adicionais.

3. Em alguns casos, o uso do padrão Builder pode resultar em uma pequena perda de desempenho, especialmente quando o objeto a ser construído é simples e não requer um processo de construção complexo.

BUILDER CÓDIGO

```java
public class Carro {
      private String marca;
      private String modelo;
      private int ano;
      private int potencia;

      public Carro(String marca, String modelo,
int ano, int potencia) {
          this.marca = marca;
          this.modelo = modelo;
          this.ano = ano;
          this.potencia = potencia;
    }
      public String getMarca() {
          return marca;
    }
      public String getModelo() {
          return modelo;}
      public int getAno() {
          return ano;
    }
      public int getPotencia() {
          return potencia;
    }}
```

```java
public interface CarroBuilder {
    CarroBuilder setMarca(String marca);
    CarroBuilder setModelo(String modelo);
    CarroBuilder setAno(int ano);
    CarroBuilder setPotencia(int potencia);
    Carro build();}
```

BUILDER CÓDIGO

```java
public class CarroBuilderImplents implements CarroBuilder {
    private String marca;
    private String modelo;
    private int ano;
    private int potencia;

    public CarroBuilder setMarca(String marca) {
        this.marca = marca;
        return this;
      }
    public CarroBuilder setModelo(String modelo) {
        this.modelo = modelo;
        return this;
      }
    public CarroBuilder setAno(int ano) {
        this.ano = ano;
        return this;
      }
    public CarroBuilder setPotencia(int potencia) {
        this.potencia = potencia;
        return this;
      }
    public Carro build() {
        return new Carro(marca, modelo, ano, potencia);
    }}
```

```java
public class Main {
    public static void main(String[] args) {
        CarroBuilder builder = new CarroBuilderImplents();

        Carro carro = builder
                .setMarca("Ford")
                .setModelo("Mustang")
                .setAno(2022)
                .setPotencia(450)
                .build();

        System.out.println("Marca: " + carro.getMarca());
        System.out.println("Modelo: " + carro.getModelo());
        System.out.println("Ano: " + carro.getAno());
        System.out.println("Potência: " + carro.getPotencia());
    }}
```

PROTOTYPE

1. Use o padrão quando precisar que seu código não dependa de classes concretas para criação de novos objetos.
2. É usado quando quiser evitar a explosão de subclasses para objetos muito similares.
3. Use o padrão prototype para evitar a recriação de objetos caros.

CONTRAS DO PROTOTYPE

CLONAR OBJETOS QUE TEM REFERÊNCIAS PARA OUTROS OBJETOS PODE SER SUPER COMPLEXO

JS Prototype 2

JS Prototype > ...

```
const person1 = {

name: 'Luiz',

age: 30,

};

const person2= Object.create(person1);

console.log(person1.name); // Luiz console.log(person2.name); // Luiz

// person1 é o prototype de person2

console.log(person1 Object.getPrototypeof(person2));
```

```
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
}

const personPrototype = {
  firstName: 'Luiz',
  lastName: 'Miranda',
  age: 30,

  fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
};

Person.prototype = Object.create(personPrototype);

const person1 = new Person('Luiz', 'Miranda', 30);
console.log(person1.fullName());
```

SINGLETON

1. É um padrão de projeto que possibilita limitar a criação de instâncias de classes.

2. Garante que uma classe irá possuir apenas uma instância, sendo possível acessá-la globalmente.

3. É utilizado em projetos que possuem uma ou mais classes que devem obter apenas uma instância.

SINGLETON | CONTRAS

1. É um padrão que viola o princípio de responsabilidade única.

2. A utilização do Singleton pode afetar a clareza das classes.

3. Dificulta futuras mudanças, devido ao alto acoplamento

SINGLETON | CÓDIGO

```
public class Carro
{
    2 references
    public string Placa { get; set; }
    2 references
    public string Cor { get; set; }
    2 references
    public string Modelo { get; set; }

    1 reference
    public Carro(){}
}
```

```
public class CarrosRepositorio
{
    3 references
    public List<Carro> Carros = new List<Carro>();
    3 references
    private static CarrosRepositorio instance;
    1 reference
    private CarrosRepositorio(){}

    3 references
    public static CarrosRepositorio Instance
    {
        get
        {
            if(instance == null)
            {
                instance = new CarrosRepositorio();
            }
            return instance;
        }
    }
    public void Adicionar(Carro carro)
    {
        Carros.Add(carro);
    }

    0 references
    public List<Carro>ListarCarros()
    {
        return Carros.ToList();
    }
```

SINGLETON | CÓDIGO

```
private static void Main(string[] args)
{
    Console.WriteLine("Criando um carro...\n");
    var carro1 = new Carro()
        {
            Placa = "abc123",
            Cor = "azul",
            Modelo = "celta"
        }
    ;

    System.Console.WriteLine("Instanciando uma lista de carros\n");
    CarrosRepositorio listaCarros1 = CarrosRepositorio.Instance;
    listaCarros1.Adicionar(carro1);

    System.Console.WriteLine("Instanciando segunda lista de carros\n");
    CarrosRepositorio listaCarros2 = CarrosRepositorio.Instance;
```

SINGLETON | CÓDIGO

```
if (listaCarros1 == listaCarros2)
{
    Console.WriteLine("Existe somente uma
    instância (listaCarros1 ==
    listaCarros2)\n");
    MostraCarros();
}
else
{
    Console.WriteLine("Existem instâncias
    diferentes (listaCarros1 e
    listaCarros2)\n");
}
```

```
static void MostraCarros()
{
    var carros = CarrosRepositorio.Instance.
    Carros;

    foreach(var c in carros)
    {
        System.Console.WriteLine($"Carro: {c.
        Modelo}, Cor: {c.Cor}, Placa: {c.
        Placa}");
    }
}
```

RESULTADO

```
Criando um carro...

Instanciando uma lista de carros

Instanciando segunda lista de carros

Existe somente uma instância (listaCarros1 == listaCarros2)

Carro: celta, Cor: azul, Placa: abc123
```