

Women Techmakers - Devs JavaGirl

22 Ago - 19:00

Stefanini
SCN Q 1 Bl A Ed Number One - Asa Norte
Brasília - DF

Apoio:



\$WHOAMI

Gleice Elen Silva

Bacharel em Sistemas de Informação;
Analista de Sistemas no Banco do Brasil;
Criadora do grupo Devs JavaGirl;
Co-criadora do Brasília Dev Festival;

 **@gleiceellen**

 **gleice-ellen-silva**

João Paulo Sossoloti

Dev há 12 anos;
Trabalhou no BB, TSE, TCU, etc;
Instrutor na Caelum;
Co-criador do Brasília Dev Festival;

@jopss 

jopss 



POR QUE EMBARCAR NESSA JORNADA DE COMUNIDADE?

Nós precisamos de apoio!

O ambiente de TI é um dos mais competitivos e exigentes, o profissional necessita lidar sempre com muita pressão, nesse caso, um local onde possa socializar é super válido para melhorar a qualidade de vida do desenvolvedor. E não é só apoio, a área de TI está em constante modificação e as comunidades compartilham conhecimento de maneira gratuita e liberada. Além disso há um ganho de bem estar, uma vez que você contribui para o crescimento de outras pessoas.

E O WOMEN
TECHMAKERS,
O QUE E?



Women
Techmakers

História e propósito

Criado por **Megan Smith** e atualmente liderado por **Natalie Villalobos** o WTM é uma marca do Google, que representa o programa global para mulheres na tecnologia.

Provê, **comunidade**, **visibilidade** e **recursos** para mulheres de tecnologia conduzirem inovação e participação na área;

Incentiva a conexão entre essas mulheres, como forma de inspiração, além de promover capacitação e dar destaque a elas.

E O DEVS JAVAGIRL?

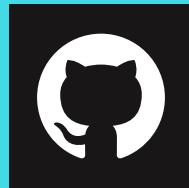
643

Membros do grupo Devs JavaGirl no
facebook.

Comunidade de Java para mulheres

Somos um grupo de mulheres que trabalham, estudam ou se interessam por Java. Temos um perfil ativo em todas as redes sociais e promovemos discussões a respeito de carreira e da linguagem. Além de iniciativas como o grupo de estudos, meetups, cursos gratuitos e mentoria.

NOSSAS REDES SOCIAIS



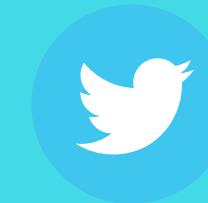
devs-javagirl

Medium

devs-javagirl



Devs-Java-Girl



@devsjavagirl

Slack

devsjavagirl.slack.com



<https://www.youtube.com/channel/UCgoGOLleKmM9ikxQhGhhOhQ>

LET'S TALK ABOUT

DESIGN PATTERNS

O QUE SÃO PADRÕES

São um **repertório** de soluções e princípios que ajudam os desenvolvedores a criar software e que são codificados em um formato estruturado consistindo de:

Nome

Problema que soluciona

Solução para o problema

O **objetivo** dos padrões é codificar conhecimento (*knowing*) existente de forma que possa ser reaplicado em contextos diferentes.

POR QUE APRENDER PADRÕES

APRENDER
COM A
EXPERIÊNCIA
DO OUTRO

PROGRAMAR
BEM COM
ORIENTAÇÃO A
OBJETO

DESENVOLVER
SOFTWARE DE
MELHOR
QUALIDADE

ELEMENTOS DE UM PADRÃO

Nome

Problema que soluciona

Quando aplicar o padrão e em que condições?

Solução para o problema

Descrição de um problema e como usar os elementos disponíveis para solucioná-lo.

Consequências

- * Custos e benefícios de se aplicar o padrão;
- * Impacto na flexibilidade, extensibilidade, portabilidade e eficiência do sistema.

TIPOS:

GRASP

Descrevem os princípios fundamentais da atribuição de **responsabilidades** a objetos, expressas na forma de padrões de projeto.

Exploram os princípios fundamentais de sistemas OO.

GOF

Soluções genéricas para os problemas mais comuns do desenvolvimento de software orientado a objetos.

São 23 padrões de design, soluções documentadas, coletadas de experiências de sucesso da indústria de software.

GRASP

Creator
Information Expert
Low Coupling
High Cohesion
Controller

Polymorphism
Pure Fabrication
Indirection
Protected Variations

PADRÕES

GOF

Factory Method
Abstract Factory
Builder
Prototype
Singleton

Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Chain of Responsibility
Memento
Template Method
Command
Observer
Visitor
Iterator
Mediator
State
Strategy

DESIGN PATTERN:

BUILDER

BUILDER

Faz parte do grupo de padrões titulado como padrões criacionais

É utilizado para construção de objetos complexos fazendo-se uso de uma abordagem passo a passo.

Quando usar?

- Onde uma classe possui o construtor muito grande, e você precisa passar todos os parâmetros mesmo que não necessite de algum.
- Quando necessita criar muitas representações de uma classe.
- Quando seu objeto a ser criado é muito complexo.

BUILDER

Definições

- Builder: uma interface indicando os passos a serem realizados para criar um objeto.
- Concrete Builder: implementa o Builder e repassa os dados ao objeto.
- Director: Recebe um Builder e monta os passos com os dados necessários.
- Product: O objeto final criado.

BUILDER

```
1 package devsjavagirl.designpatterns;
2
3 public class Pedido {
4
5     private Long id;
6     private Double valor;
7     private String descricao;
8
9     public Long getId() {
10         return id;
11     }
12     public void setId(Long id) {
13         this.id = id;
14     }
15     public String getDescricao() {
16         return descricao;
17     }
18     public void setDescricao(String descricao) {
19         this.descricao = descricao;
20     }
21     public Double getValor() {
22         return valor;
23     }
24     public void setValor(Double valor) {
25         this.valor = valor;
26     }
27
28 }
```

```
1 package devsjavagirl.designpatterns;
2
3 public class TesteBuilder {
4
5     public static void main(String[] args) {
6
7         Pedido pedido = new Pedido();
8         pedido.setId(1L);
9         pedido.setDescricao("caixa de chocolate lindt branco");
10        pedido.setValor(24.33D);
11
12    }
13
14 }
```

BUILDER

```
1 package devsjavagirl.designpatterns;
2
3 public class Pedido {
4
5     private Long id;
6     private Double valor;
7     private String descricao;
8
9     public Pedido(Long id, Double valor, String descricao) {
10         this.id = id;
11         this.valor = valor;
12         this.descricao = descricao;
13     }
14
15     public Long getId() {
16         return id;
17     }
18     public String getDescricao() {
19         return descricao;
20     }
21     public Double getValor() {
22         return valor;
23     }
24
25 }
```

```
1 package devsjavagirl.designpatterns;
2
3 public class TesteBuilder {
4
5     public static void main(String[] args) {
6
7         Pedido pedido = new Pedido(1L, 24.33D, "caixa de chocolate lindt branco");
8
9     }
10 }
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

BUILDER

```
1 package devsjavagirl.designpatterns;
2
3 public interface PedidoBuilder {
4
5     public void setId(Long id);
6     public void setValor(Double valor);
7     public void setDescricao(String descricao);
8     public Pedido get();
9
10 }
```

```
1 package devsjavagirl.designpatterns;
2
3 public class PedidoConcrete implements PedidoBuilder {
4
5     private Pedido pedido;
6
7     public void setId(Long id) {
8         this.pedido.setId(id);
9     }
10
11     public void setValor(Double valor) {
12         this.pedido.setValor(valor);
13     }
14
15     public void setDescricao(String descricao) {
16         this.pedido.setDescricao(descricao);
17     }
18
19     public Pedido get() {
20         return pedido;
21     }
22
23 }
```

BUILDER

```
1 package devsjavagirl.designpatterns;
2
3 public class PedidoDirector {
4
5     public Pedido criarChocolateLindt(PedidoBuilder builder) {
6
7         builder.setId(1L);
8         builder.setValor(24.33D);
9         builder.setDescricao("caixa de chocolate lindt branco");
10
11     return builder.get();
12 }
13
14 }
```

```
1 package devsjavagirl.designpatterns;
2
3 public class TesteBuilder {
4
5     public static void main(String[] args) {
6
7         Pedido pedido = new PedidoDirector().criarChocolateLindt( new PedidoConcrete() );
8
9     }
10
11 }
```

BUILDER

Prós

- Possibilita criação passo por passo.
- Isola complexidade de construção.
- Reaproveitamento de código.

Contras

- Cria uma complexidade a mais na aplicação com a elaboração das classes Builders.

Dicas:

1. Dá para criar usando Interface Fluente.
2. Framework Lombok ajuda muito no Builder.

DESIGN PATTERN:

STRATEGY

STRATEGY

Faz parte do grupo de padrões titulado como padrões estrutural.

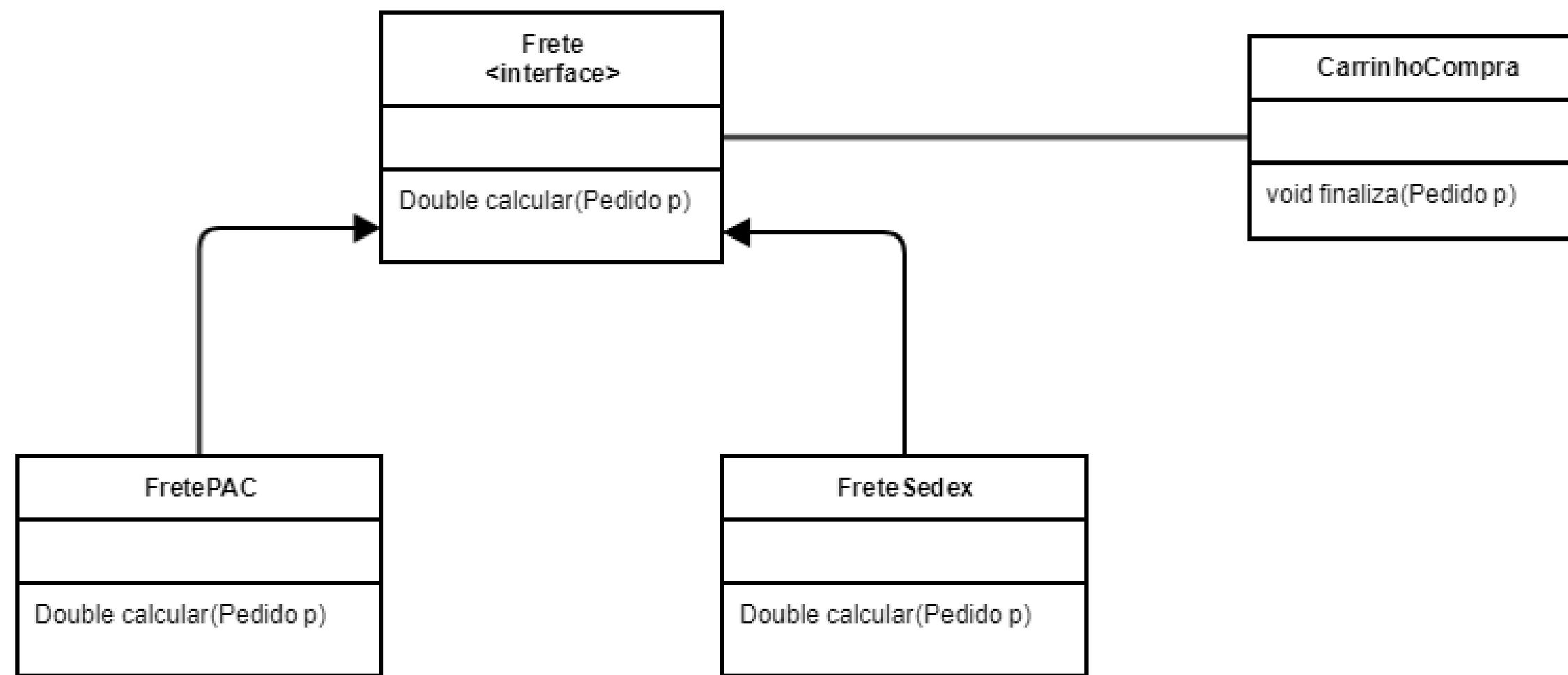
Permite que a regra de negocio varie conforme a implementação passada ao objeto.

Devemos utilizar os conceitos de OO chamados composição, interface e reescrita de métodos.

Quando usar?

- Quando sua regra esta espalhada pelo código. Este padrão permite ainda adicionar uma regra nova sem impactar nas existentes.
- Quando não queremos expor as regras implementadas. O Strategy isola as regras internas e suas dependências.
- Quando temos muitas condicionais para indicar uma determinada regra. Com o Strategy, podemos criar algoritmos sem as condicionais.

STRATEGY



STRATEGY

```
1 package devsjavagirl.designpatterns;
2
3 public interface Frete {
4     public Double calcular(Pedido pedido);
5 }
```

```
1 package devsjavagirl.designpatterns;
2
3 public class FretePAC implements Frete {
4
5     public Double calcular(Pedido pedido) {
6         //faz calculo maneiro
7         return 12.66;
8     }
9
10 }
```

```
1 package devsjavagirl.designpatterns;
2
3 public class FreteSedex implements Frete {
4
5     public Double calcular(Pedido pedido) {
6         //faz calculo maneiro
7         return 23.11;
8     }
9
10 }
```

STRATEGY

```
1 package devsjavagirl.designpatterns;
2
3 public class CarrinhoCompra {
4
5     private Frete frete;
6
7     public CarrinhoCompra(Frete frete) {
8         this.frete = frete;
9     }
10
11    public void finalizar(Pedido pedido) {
12
13        Double valorFrete = this.frete.calcular(pedido);
14
15        //outras coisas interessantes
16
17    }
18
19 }
```

STRATEGY

```
1 package devsjavagirl.designpatterns;
2
3 public class TesteStrategy {
4@    public static void main(String[] args) {
5
6        Pedido pedido = new PedidoDirector().criarChocolateLindt( new PedidoConcrete() );
7
8        CarrinhoCompra carrinho = new CarrinhoCompra( new FreteSedex() );
9        carrinho.finalizar(pedido);
10
11    }
12 }
```

```
1 package devsjavagirl.designpatterns;
2
3 public class TesteStrategy {
4@    public static void main(String[] args) {
5
6        Pedido pedido = new PedidoDirector().criarChocolateLindt( new PedidoConcrete() );
7
8        CarrinhoCompra carrinho = new CarrinhoCompra( new FretePAC() );
9        carrinho.finalizar(pedido);
10
11    }
12 }
```

STRATEGY

Prós

- Allows building products step by step.
- Allows using the same code for building different products.
- Isolates the complex construction code from a product's core business logic.

Contras

- Increases overall code complexity by creating multiple additional classes.

DESIGN PATTERN:

SINGLETON

SINGLETON

Faz parte do grupo de padrões titulado como padrões criacionais
É utilizado para criar uma classe onde temos somente uma instancia do objeto na
memória, com acesso a variável de forma global (static).

Quando usar?

- Quando queremos uma única instancia de um objeto na aplicação toda. Por exemplo, conexão com banco de dados.

Dois passos

- Faça um construtor privado, garantindo acesso somente interno ao objeto.
- Crie um método estático para controlar a criação do objeto único.

SINGLETON

```
1 package devsjavagirl.designpatterns;
2
3 public class GravadorBanco {
4
5     private GravadorBanco() {
6         //nope
7     }
8
9     private static GravadorBanco instancia;
10
11    public static GravadorBanco getInstance() {
12        if(instancia == null) {
13            instancia = new GravadorBanco();
14        }
15        return instancia;
16    }
17
18    public void conectarBanco() {
19        //conexao JDBC
20    }
21
22 }
```

```
1 package devsjavagirl.designpatterns;
2
3 public class TesteSingleton {
4     public static void main(String[] args) {
5
6         Pedido pedido = new PedidoDirector().criarChocolateLindt( new PedidoConcrete() );
7
8         CarrinhoCompra carrinho = new CarrinhoCompra( new FretePAC() );
9         carrinho.finalizar(pedido);
10
11        GravadorBanco.getInstance().conectarBanco(); //efetua o new GravadorBanco
12        GravadorBanco.getInstance().conectarBanco(); //so retorna
13        GravadorBanco.getInstance().conectarBanco(); //so retorna
14
15    }
16 }
```

SINGLETON

Prós

- Garante uma única instância da classe.
- Centraliza acesso a variáveis globais.
- Possibilita múltiplas inicializações.

Contras

- Violação do princípio de Separação de Responsabilidade.
- Mascara design de código ruim.
- Requer tratamento para Threads.
- Requer mock para testes unitarios.
- Não possibilita IoC.

B E C A R E F U L . . .

REFERÊNCIAS

<https://refactoring.guru/design-patterns/>

Curso J930: Design Patterns - www.argonavis.com.br

WE HOPE YOU JOIN US

THANK YOU

