

# Padrões GRASP avançados

Matheus Barbosa  
matheus.barbosa@dcx.ufpb.br

## Padrões Grasp básicos

- Information Expert (Especialista de Informação)
- Creator (Criador)
- High Cohesion (Coesão alta)
- Low Coupling (Baixo acoplamento)
- Controller (Controlador)

## Padrões Grasp avançados

- Polymorphism (Polimorfismo)
- Indirection (Indirecionamento)
- Pure Fabrication (Invenção Pura)
- Protected Variations (variações protegidas)

# Polymorphism (Polimorfismo)

## Problema:

- Como lidar com alternativas baseadas no tipo? Como criar componentes de software plugáveis?
- Deseja-se evitar variação condicional (if-then-else): pouco extensível.
- Deseja-se substituir um componente por outro sem afetar o cliente.

**Classe Grafico**

**desenhar(){**

**switch(tipo){**

**case: QUADRADO: // Faz algo**

**case: CIRCULO: // Faz outra coisa**

**case: TRIANGULO: // Faz...**

**}**

**FimClasse**

# Polymorphism (Polimorfismo)

## Solução:

Não use lógica condicional para realizar alternativas diferentes baseadas em tipo. Atribua responsabilidades ao comportamento usando **operações polimórficas**

Herança direta ou herança múltipla (*interfaces*)

```
Classe abstrata Grafico
```

```
    abstrato desenhar();
```

```
FimClasse
```

```
Classe Quadrado estende Grafico
```

```
    desenhar(){ // faz algo }
```

```
FimClasse
```

```
...
```

Cuidado com herança

Mudança na superclasse  
pode afetar todas as  
subclasses



**Subclasses carregam implementações, interfaces não!**

**A extends B**

A herda os  
comportamentos e  
dados privados  
(invariáveis) de B

**A implements B**

A não herda nenhum  
comportamento  
prévio de B

```
interface Grafico
```

```
    desenhar();
```

```
FimInterface
```

```
Classe Quadrado implementa Grafico
```

```
    desenhar(){ // faz algo }
```

```
FimClasse
```

```
...
```

## Pure Fabrication (Invenção Pura)

### Problema:

- Que objeto deve ter a responsabilidade, quando você não quer violar *High Cohesion* e *Low Coupling*, mas as soluções oferecidas por *Expert* não são adequadas?
- Atribuir responsabilidades apenas para classes do domínio conceitual pode levar a situações de maior acoplamento e menos coesão.

## Pure Fabrication (Invenção Pura)

### Solução:

Não repasse essas responsabilidades para classes de domínio. Atribua um conjunto coeso de responsabilidades a classes artificiais. Use sua criatividade, **invente!**

Suponha que necessitemos de suporte para salvar instâncias de Venda em um banco de dados relacional.

Segundo o padrão *Expert*,  
onde devemos colocar essa  
responsabilidade?

Na VENDA!!

Porém, analise as  
seguintes implicações...

**Baixa coesão:** A classe Venda realiza várias operações de banco de dados que não têm relação com o conceito de "vender".

**Alto acoplamento:** A classe depende diretamente da interface do banco de dados, aumentando o acoplamento de forma inadequada.

**Pouca reutilização:** Centralizar a lógica de persistência em Venda gera duplicação de código em outras classes e reduz a reutilização.



## Conceitos de Negócio

Pagamentos

Vendas

Clientes

## Conceitos de Computação

Persistência

Falhas

Erro de acesso.

## Decomposição representacional

Criar classes  
relacionadas a  
conceitos do domínio.

## Decomposição comportamental

Agrupar  
comportamentos ou  
algoritmos.

## Venda

- dataHora : Date

+ getTotal() : BigDecimal

## ArmazenamentoPersistente

+ inserir(v : Venda) : void  
+ atualizar(v : Venda) : void  
+ recuperarVendas() : List

## JDBC

# Indirection (Indirecionamento)

## Problema:

Onde atribuir uma responsabilidade para evitar acoplamento direto entre duas ou mais coisas? Como desacoplar objetos para que seja possível suportar baixo acoplamento e manter elevado o potencial de reuso?

## Indirection (Indirecionamento)

### Solução:

Atribua a responsabilidade a um **objeto intermediário** para mediar as mensagens entre outros componentes ou serviços para que não sejam diretamente acoplados.

O próprio exemplo da ***invenção pura*** desacopla *Venda* de serviços de banco de dados relacional, também é um exemplo de indireção. A classe de armazenamento persistente atua como um **intermediário** entre venda e banco de dados.

*"Não existe problema na computação que não possa ser resolvido com um nível extra de indireção"*

- David Wheeler

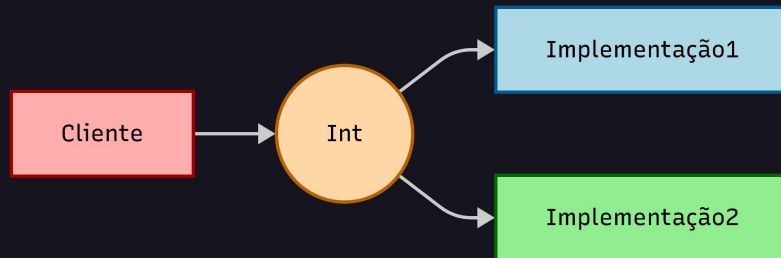
"... exceto problema de muitas camadas de indireção"

- Kevlin Henney

# Injeção de dependência: dependências devem ser injetadas em um objeto dependente

```
class Cliente {  
    Int int;  
  
    Cliente(){  
        int = new Implementacao1();  
    }  
}
```

dependência criada internamente



```
c = new Cliente();  
c.m();  
c.n();
```



Dependência criada  
externamente

Alterando  
dependência em  
tempo de execução

Trabalhando com  
dependência  
alternativa

```
class Cliente {  
    Int int;  
  
    void setInt(Int i){  
        this.int = i;  
    }  
}
```

```
c = new Cliente();  
c.setInt(new Implementacao1())  
c.m();  
c.n();
```

```
c = new Cliente();  
c.setInt(new Implementacao1());  
c.m();  
...  
c = new Cliente();  
c.setInt(new Implementacao2());  
c.m();
```

# Protected Variations (variações protegidas)

## Problema:

Como projetar objetos, subsistema e sistemas para que as variações ou instabilidades nesses elementos não tenha um impacto indesejável nos outros elementos?

## Protected Variations (variações protegidas)

### Solução:

- Identificar pontos de variação ou instabilidade potenciais.
- Atribuir responsabilidades para criar uma interface estável em volta desses pontos.

Imagina um sistema de PDV com suporte a vários fornecedores de emissão de nota fiscal

Variações protegidas (Manter separado e protegido):

- Protocolo dos fornecedores
- Inclusão de novo fornecedor
- Nova regra de legislação

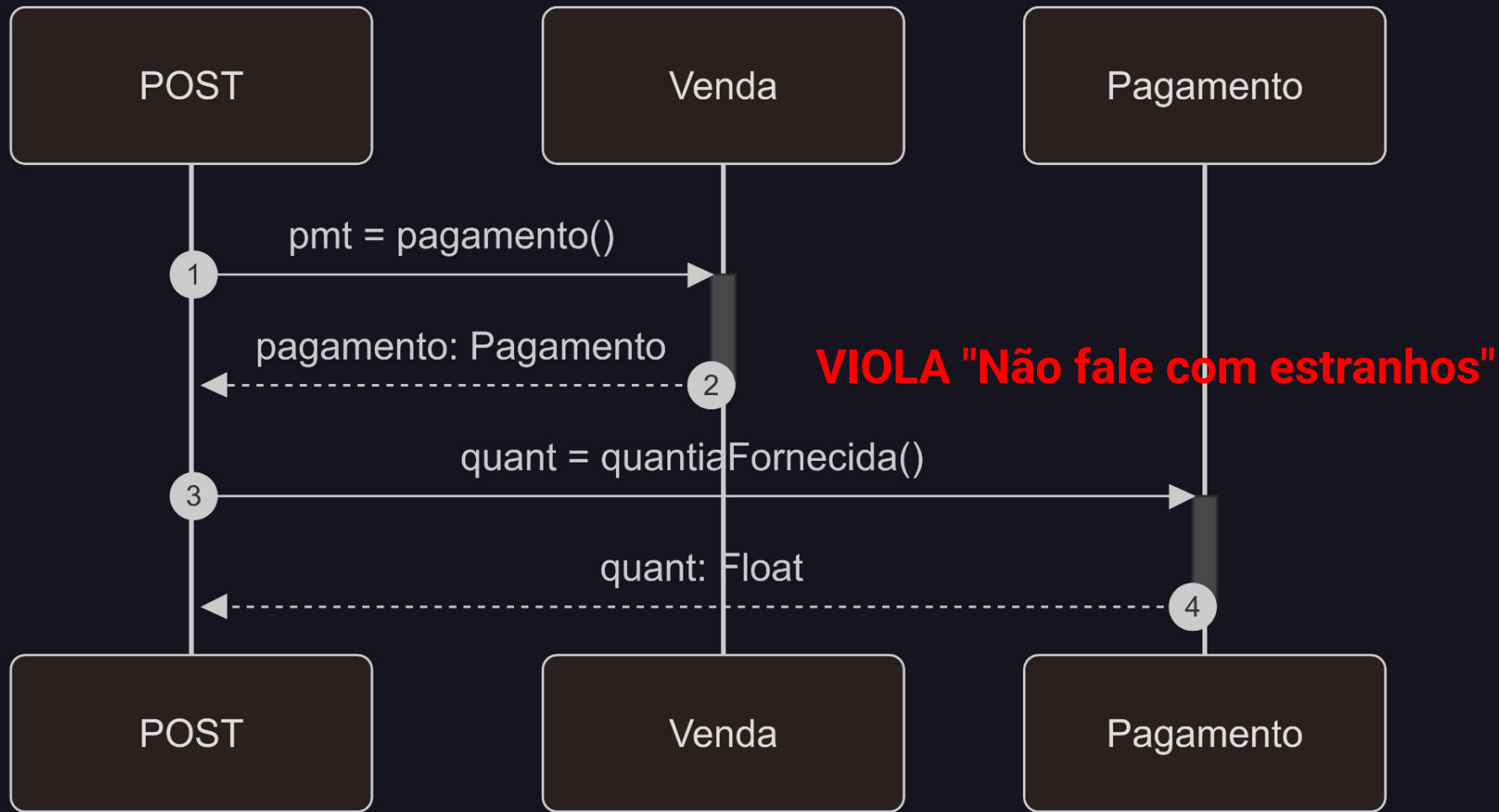
Usado em conjunto com  
Polimorfismo e Indireção

Alta flexibilidade pode ser custosa HOJE

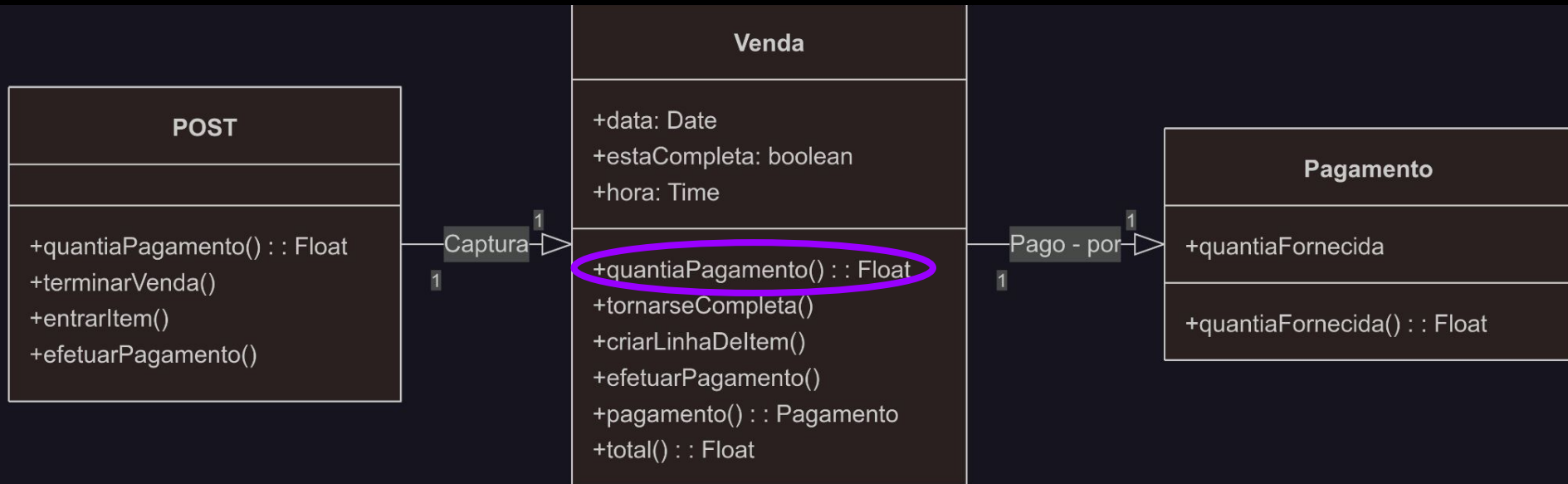
Inflexibilidade pode ser custosa AMANHÃ

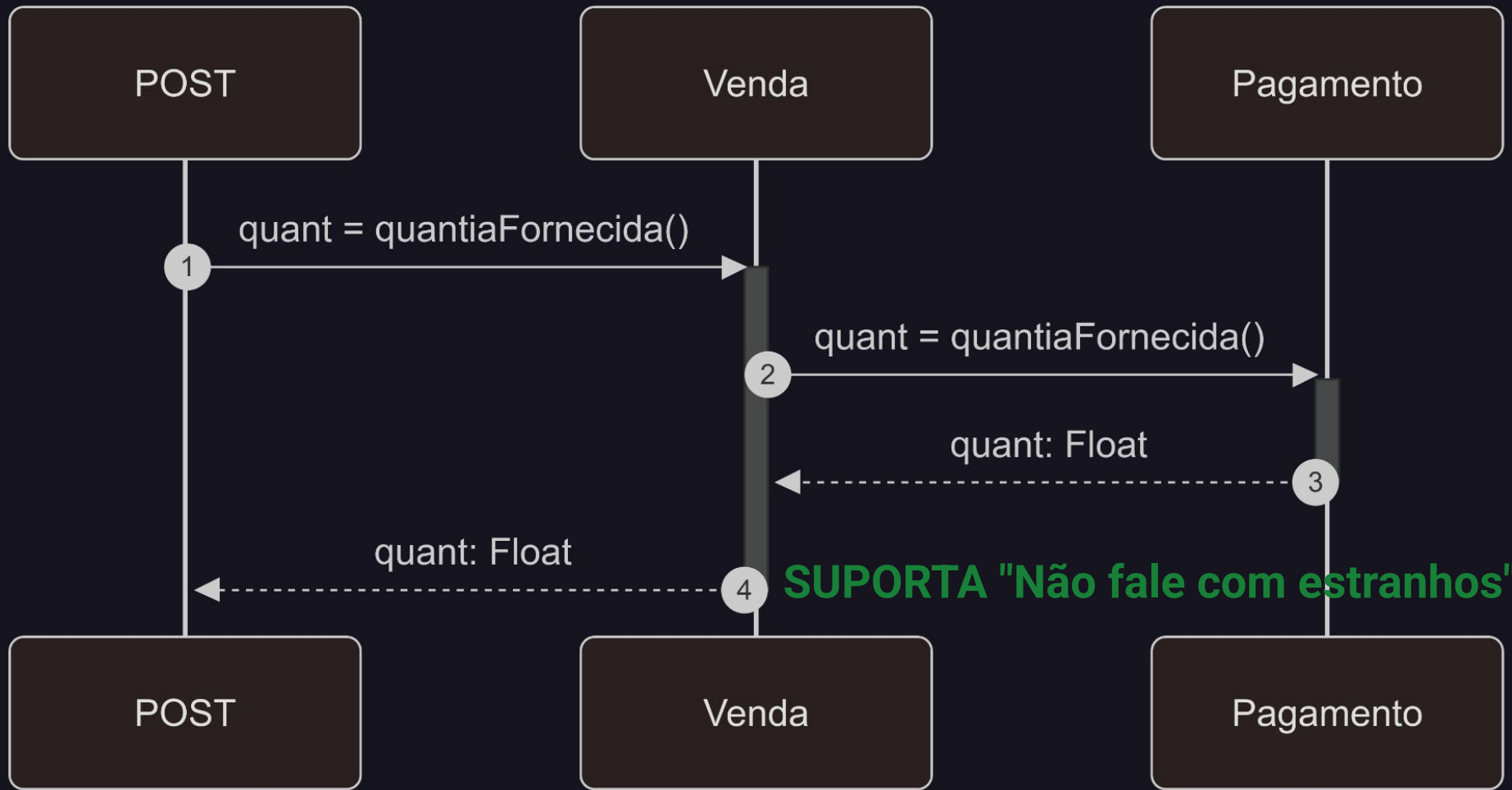
# Don't Talk to Strangers (Não fale com estranhos)















```
class Pagamento {  
    public void processar(String tipo, double valor) {  
        if (tipo.equals("CARTAO")) {  
            System.out.println("Pagando no cartão R$" + valor);  
        } else if (tipo.equals("PIX")) {  
            System.out.println("Pagando via Pix R$" + valor);  
        }  
    }  
}
```




```
interface Pagamento {
    void processar(double valor);
}

class CartaoCredito implements Pagamento {
    public void processar(double valor) {
        System.out.println("Cartão: R$" + valor);
    }
}

class Pix implements Pagamento {
    public void processar(double valor) {
        System.out.println("Pix: R$" + valor);
    }
}
```



```
class Pedido {  
    public void confirmar() {  
        System.out.println("Pedido confirmado");  
        System.out.println("[LOG] Pedido confirmado"); // misturado  
    }  
}
```




```
class Logger {  
    public void registrar(String evento) {  
        System.out.println("[LOG] " + evento);  
    }  
}
```

```
class Pedido {  
    private Logger logger = new Logger();  
  
    public void confirmar() {  
        logger.registrar("Pedido confirmado");  
    }  
}
```



```
class Usuario {  
    public void enviarEmail(String msg) {  
        System.out.println("Enviando e-mail: " + msg);  
    }  
}
```



```
class EnviadorEmail {
    public void enviar(String msg) {
        System.out.println("E-mail enviado: " + msg);
    }
}

class Notificador {
    private EnviadorEmail enviador = new EnviadorEmail();

    public void notificar(String mensagem) {
        enviador.enviar(mensagem);
    }
}
```