Aula 05

Padrões de Projeto

Análise e Projeto Arquitetural de Software

Prof. Thiago

Pós-Graduação em Gestão de Sistemas de Informação

Objetivos

- Após esta aula, você deverá ser capaz de:
  - Conceituar padrão de projeto;
  - Aplicar os principais padrões de projeto GoF

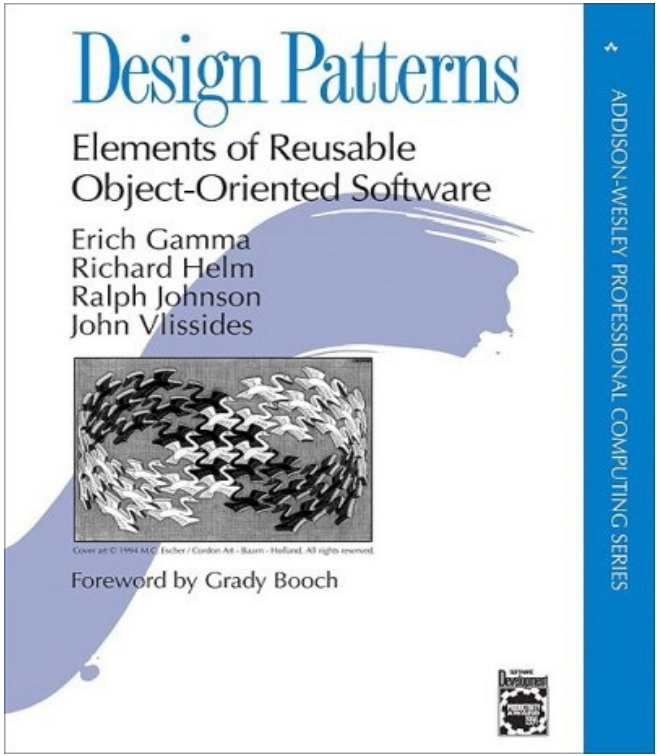
Conceito de padrão

Padrão

Descrição de um problema que ocorre muitas vezes, e de uma solução genérica que pode ser aplicada toda vez que o problema ocorre

Padrões em computação

- O livro *Design Patterns*, de Gamma, Johnson, Vlissides e Helm, publicado em 1995, trouxe o conceito de padrões para a arquitetura de software
- O impacto foi enorme na comunidade de desenvolvimento, e o livro vendeu milhares de cópias
- Os padrões definidos ficaram conhecidos como **padrões GoF** (de Gang of Four)



Padrões GoF

Os padrões GoF são organizados em três categorias:

- **Criacional**
  - Como tratar da criação de objetos problemáticos
- **Estrutural**
  - Como construir estruturas flexíveis
- **Comportamental**
  - Como criar comportamentos especiais

Padrões GoF

Criacional	Estrutural	Comportamental
Factory Method*	Adapter	Interpreter*
Abstract Factory	Bridge	Template Method*
Builder	**Composite**	Chain of Responsibilty
Prototype	Decorator	Command
Singleton	**Façade**	Iterator
	Proxy	Mediator
		Memento
		Flyweight
		Observer
		State
		Strategy
		Visitor

Padrões GoF

- Vamos estudar hoje o uso de alguns padrões GoF de ampla utilização:
  - **Singleton** (objeto unitário)
  - **Abstract Factory** (fábrica abstrata)
  - **Strategy** (estratégia)
  - **Composite** (composição)
  - **Façade** (fachada)

Singleton

- **O Problema**

Alguns objetos têm responsabilidades relacionadas ao controle de outros objetos (Ex.: instanciar, destruir, agregar e fazer busca)

Queremos acessar esses objetos facilmente, de qualquer ponto, e ainda garantir que haja apenas uma instância de cada objeto.

Singleton

- **O Problema**

Por exemplo, “catálogos” que agregam objetos para busca, ou Factory’s de objetos são típicos exemplos de instâncias únicas

Idéias ruins:

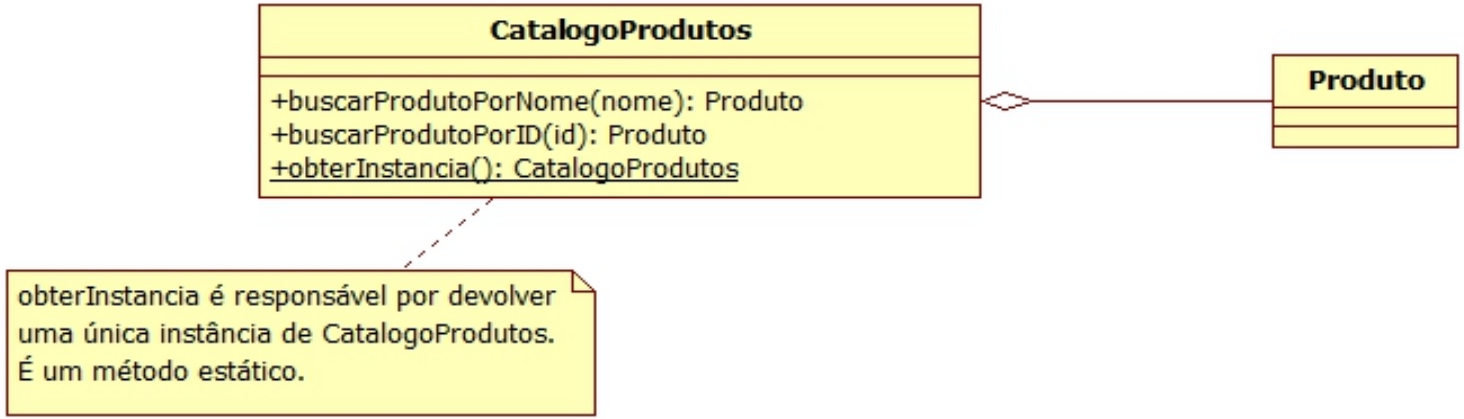
- Variáveis globais;
- Classe cheia de métodos estáticos (vira programação estruturada dentro da OO)

Singleton

A Solução

- A própria classe, com métodos e atributos regulares, é responsável pela criação da sua própria instância

Singleton


CatalogoProdutos
+buscarProdutoPorNome(nome): Produto
+buscarProdutoPorID(id): Produto
+obterInstancia(): CatalogoProdutos
Produto
obterInstancia é responsável por devolver uma única instância de CatalogoProdutos. É um método estático.

Singleton

- Implementando um Singleton:

```
public class CatalogoProdutos {
   private static CatalogoProdutos _instancia;

   public static CatalogoProdutos obterInstancia() {
        if (_instancia == null)
                _instancia = new CatalogoProdutos();
        return _instancia;
   }

   private CatalogoProdutos() { }
   // Seguem os demais métodos
}
```

Abstract Factory

- **O Problema**

  Queremos instanciar objetos de classes (implementações) diferentes, mas que tem a mesma funcionalidade, dependendo da situação.

  Queremos, também, que a troca do tipo de objeto utilizado seja o mais transparente possível para a aplicação.

Abstract Factory

- **O Problema**

Por exemplo, considere uma classe que efetua acesso a um SGBD específico, ou uma classe que implementa um widget de interface, cuja implementação varia de acordo com o ambiente (Windows, MOTIF, MacOS)

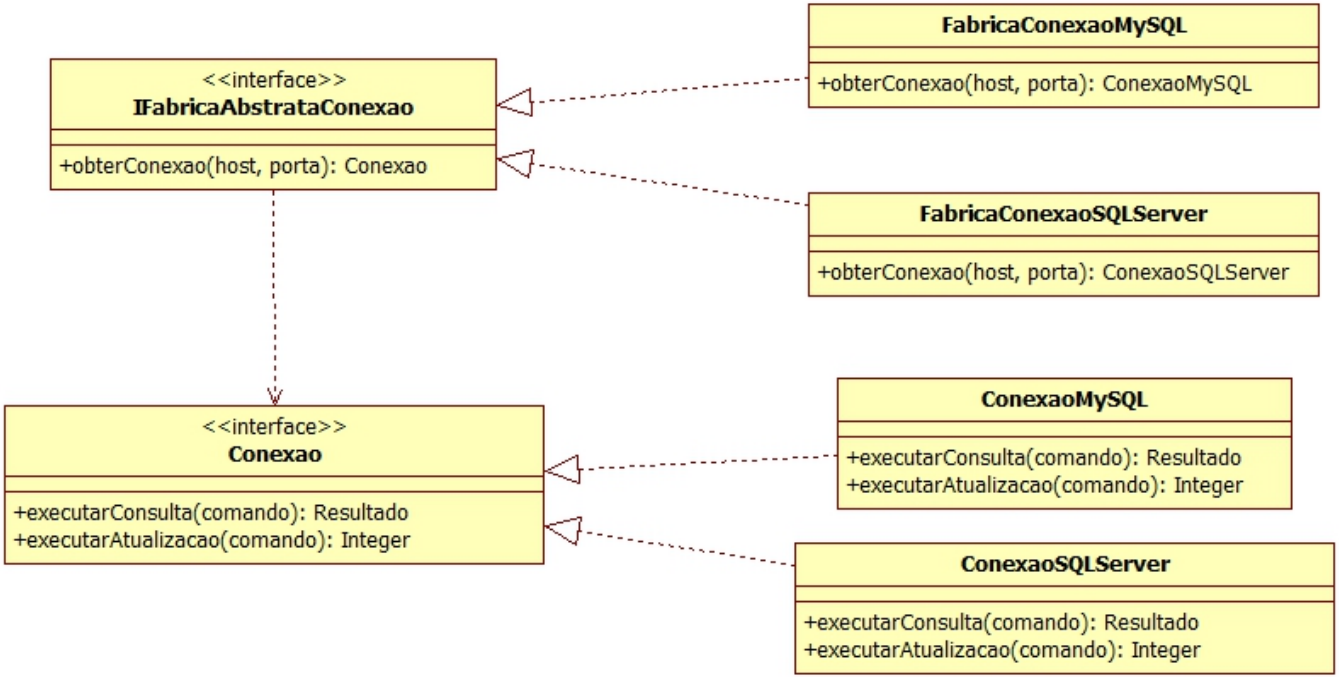
Abstract Factory

- **A solução**
  - A Abstract Factory define uma **interface**, implementada pelas *Concrete Factory*'s, que por sua vez são responsáveis por instanciar as classes concretas

Abstract Factory

- **Exemplo**
  - Um conjunto de classes de acesso a banco de dados deve ser desacoplado do SGBD utilizado
  - Criamos uma interface `Conexao`, que generaliza as operações de consulta e atualização implementadas
  - A interface `IFabricaAbstrataConexao` define a operação de instanciar objetos de classes que implementem a interface `Conexao`

Abstract Factory



Abstract Factory

- Em Java, a Abstract Factory seria instanciada assim:

```
IFabricaAbstrataConexao fabConexao = new
  FabricaConexaoMySQL();
```

ou

```
IFabricaAbstrataConexao fabConexao = new
  FabricaConexaoSQLServer();
```

- Esse é o único ponto de acoplamento do código com a implementação real do acesso ao SGBD

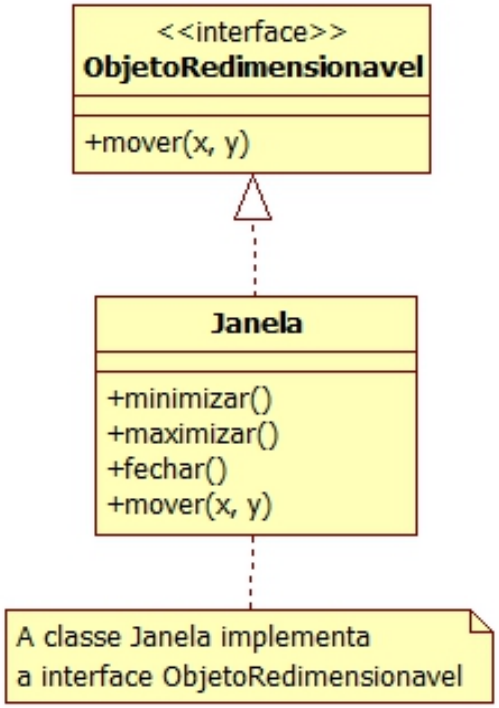
Abstract Factory

- Desse ponto em diante, todas as operações são realizadas em termos das interfaces definidas:

```
Conexao c =
  fabConexao.obterConexao("localhost",
  3128);
```

Uma palavra sobre interfaces

- Uma **interface** é a definição da assinatura de um conjunto de métodos
- Uma classe pode implementar uma ou mais interfaces
- A notação mais usual é o estereótipo `<<interface>>` no Diagrama de Classes



Uma palavra sobre interfaces

- Interfaces são usadas para
  - Permitir que várias classes implementem um comportamento em comum sem depender de herança
  - Permitir que a implementação de uma operação mude de forma desacoplada de quem usa a operação

- Interfaces não devem ser usadas para
  - Especificar um comportamento que hoje apenas uma classe implementa, porque *talvez algum dia* outra classe o implemente

Strategy

- **O Problema**

Precisamos de vários algoritmos diferentes para processar dados em uma classe.

Os algoritmos podem ser trocados de tempos em tempos e essa troca não deve causar impacto ao sistema.

Strategy

- **A solução**

Os algoritmos são desacoplados da classe principal, criando classes “especializadas” em processar a informação seguindo um determinado algoritmo.

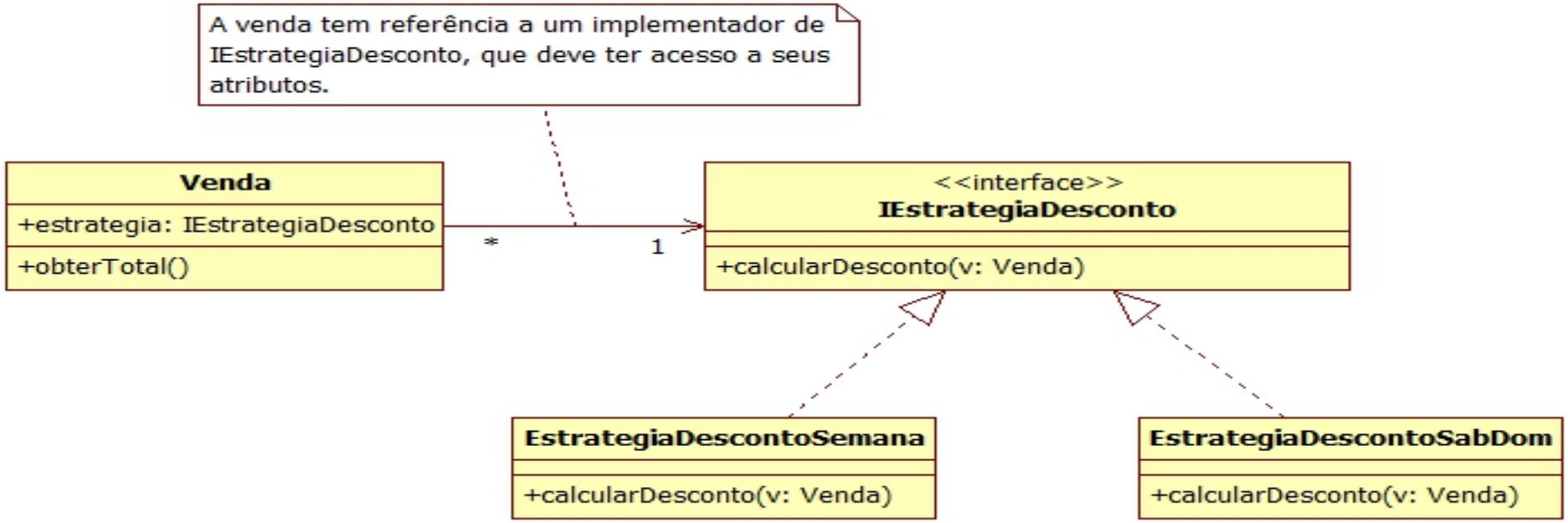
A classe sobre a qual os algoritmos são executados é chamada **classe de contexto**; ela deve fornecer visibilidade dos seus dados para a classe da estratégia (por exemplo, através de *getters* e *setters*)

Strategy

- **A solução**

Considere um sistema de vendas onde a política de desconto muda. De segunda a sexta, é concedido 10% de desconto em todas as vendas. Aos sábados e domingos, na compra de 10 produtos ou mais, é oferecido um produto grátis adicional de valor inferior a 10% do total da compra.

Strategy


A venda tem referência a um implementador de IEstrategiaDesconto, que deve ter acesso a seus atributos.
Venda
+estrategia: IEstrategiaDesconto
+obterTotal()
*
1
<<interface>>
IEstrategiaDesconto
+calcularDesconto(v: Venda)
EstrategiaDescontoSemana
+calcularDesconto(v: Venda)
EstrategiaDescontoSabDom
+calcularDesconto(v: Venda)

Composite

- **O problema**

Queremos reutilizar vários objetos diferentes, mas com tarefas similares, fazendo com que eles trabalhem juntos para resolver o mesmo problema.

Além disso, queremos que seja indiferente para a classe usuária dos nossos objetos o fato dela estar usando um objeto só ou um conjunto deles

Composite

- **O problema**

Esse tipo de reuso pode acontecer com:

- Compiladores (ao interpretar um comando ou conjuntos de comandos);
- Interfaces gráficas (ao manipular *widgets* ou agrupamentos de *widgets*);
- Sistemas comerciais (ao tratar regras de negócio ou composições de regras de negócio);

Composite

- **A solução**

Objetos compostos ou atômicos devem implementar a mesma interface, de modo que intercambiar o seu uso seja transparente

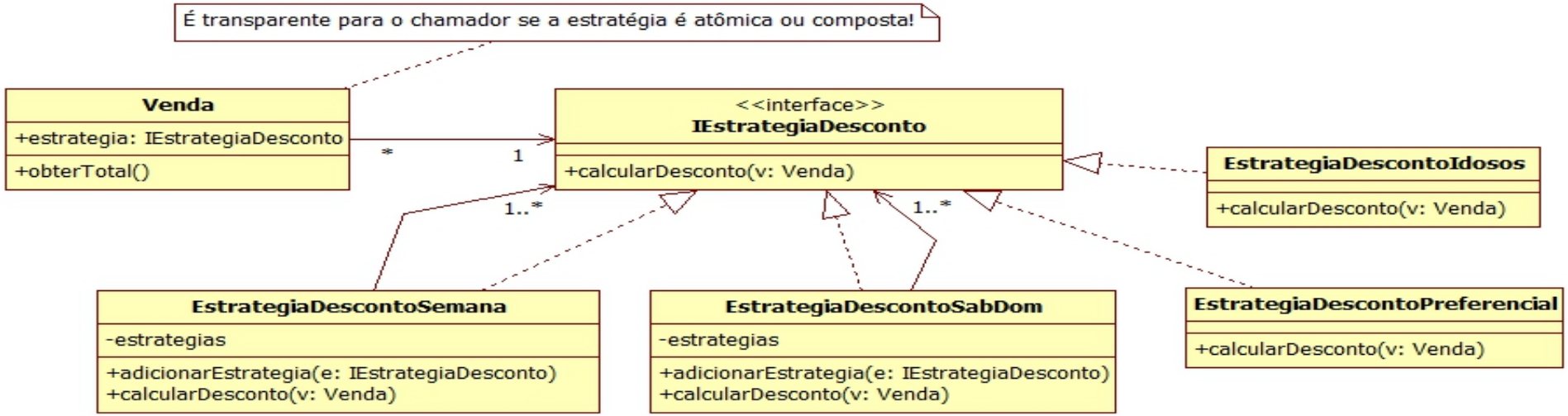
Composite

- **A solução**

Na loja do exemplo anterior, agora temos outras estratégias: descontos para idosos e descontos para clientes preferenciais.

Cada desconto se aplica cumulativamente com os já vigentes de segunda a sexta ou aos domingos

Composite


É transparente para o chamador se a estratégia é atômica ou composta!
Venda
+estrategia: IEstrategiaDesconto
+obterTotal()
*
1
<<interface>>
IEstrategiaDesconto
+calcularDesconto(v: Venda)
1..*
1..*
EstrategiaDescontoIdosos
+calcularDesconto(v: Venda)
EstrategiaDescontoSemana
-estrategias
+adicionarEstrategia(e: IEstrategiaDesconto)
+calcularDesconto(v: Venda)
EstrategiaDescontoSabDom
-estrategias
+adicionarEstrategia(e: IEstrategiaDesconto)
+calcularDesconto(v: Venda)
EstrategiaDescontoPreferencial
+calcularDesconto(v: Venda)

Façade

- **O problema**

  Estamos projetando um conjunto de classes, formando um subsistema, ou componente de software.

  A estratégia de modelagem interna do componente não é bem definida ainda, mas é necessário definir uma interface comum das futuras classes do componente com o resto do sistema.

Façade

- **A solução**

Defina uma classe para representar o único ponto de contato com o subsistema em questão. Essa classe (possivelmente um Singleton) tem uma única interface unificada que chama os métodos das classes do subsistema.

Ela atua como a **fachada** das operações do subsistema.

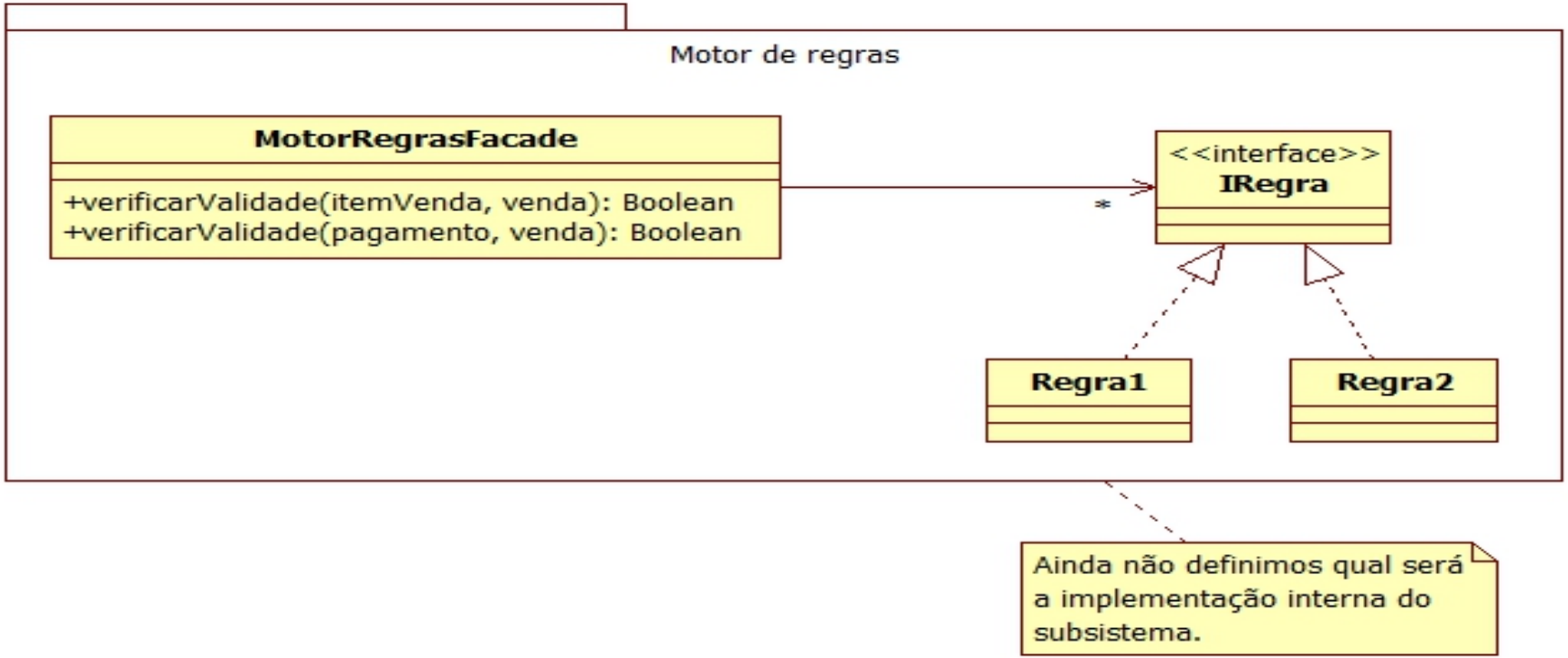
Façade

- **A solução**

  No exemplo anterior, vamos definir um subsistema *Motor de Regras*, responsável por definir se os itens de uma compra são válidos ou não (pode ser impossível entregar um item para um cliente que está muito longe, o custo de um produto pode ser mais baixo no estado de origem do produto, etc.)

  Queremos desacoplar o uso desse subsistema de sua implementação interna.

Façade


Motor de regras
MotorRegrasFacade
+verificarValidade(itemVenda, venda): Boolean
+verificarValidade(pagamento, venda): Boolean
*
<<interface>>
IRegra
Regra1
Regra2
Ainda não definimos qual será a implementação interna do subsistema.

Atividade em equipes

- Escolha um padrão de projeto GoF que poderia se aplicar ao seu projeto (mesmo que seja necessária uma alteração ou ampliação do escopo)

- Prepare um pequeno esboço de diagrama de classes para ilustrar sua decisão de projeto e compartilhar com seus colegas