Caique Augusto de Aquino Braga
Programação Orientada a Objetos 2 - IFMG Campus Ouro Branco
24 de Novembro de 2024
Lista 1

Questão 1 - Quais são os padrões GRASP (ou padrões de responsabilidade geral)? Explique de maneira sucinta, cada um deles.

1. Creator

Exemplo:
Um **Pedido** é responsável por criar seus próprios **ItensPedido**, porque ele possui os dados necessários para isso.

```
class ItemPedido {
    private String produto;
    private int quantidade;

    public ItemPedido(String produto, int quantidade) {
        this.produto = produto;
        this.quantidade = quantidade;
    }
}

class Pedido {
    private List<ItemPedido> itens = new ArrayList<>();
    public void adicionarItem(String produto, int quantidade) {
        // Pedido é o "Creator" de ItemPedido
        ItemPedido item = new ItemPedido(produto, quantidade);
        itens.add(item);
    }
}
```

---

2. Information Expert

Exemplo:
A classe **Carrinho** calcula o total do pedido porque possui as informações necessárias.

```
class ItemCarrinho {
    private double preco;
    private int quantidade;

    public ItemCarrinho(double preco, int quantidade) {
        this.preco = preco;
        this.quantidade = quantidade;
    }

    public double getSubtotal() {
        return preco * quantidade;
    }
}
```

```
class Carrinho {
    private List<ItemCarrinho> itens = new ArrayList<>();

    public void adicionarItem(double preco, int quantidade) {
        itens.add(new ItemCarrinho(preco, quantidade));
    }

    public double calcularTotal() {
        // Carrinho é o "Expert" que sabe calcular o total
        return itens.stream().mapToDouble(ItemCarrinho::getSubtotal).sum();
    }
}
```

---

3. Controller

Exemplo:
Um **Controlador** gerencia as interações entre a interface e a lógica de negócio.

```
class PedidoController {
    private Pedido pedido = new Pedido();

    public void processarNovoItem(String produto, int quantidade) {
        // O Controller gerencia o fluxo
        pedido.adicionarItem(produto, quantidade);
    }

    public Pedido getPedido() {
        return pedido;
    }
}
```

---

4. Low Coupling

Exemplo:
Usar interfaces reduz o acoplamento entre a lógica de negócio e a camada de persistência.

```
interface RepositorioPedido {
    void salvar(Pedido pedido);
}

class RepositorioPedidoSQL implements RepositorioPedido {
    @Override
    public void salvar(Pedido pedido) {
        // Código para salvar o pedido no banco de dados
    }
}

class PedidoService {
    private RepositorioPedido repositorio;

    public PedidoService(RepositorioPedido repositorio) {
        this.repositorio = repositorio; // Inversão de dependência
    }
```

```
    public void salvarPedido(Pedido pedido) {
        repositorio.salvar(pedido);
    }
}
```

---

5. High Cohesion

Exemplo:
Uma classe bem coesa se concentra em uma única responsabilidade.

```
class ValidadorPedido {
    public boolean validar(Pedido pedido) {
        // Lógica para validar um pedido
        return !pedido.getItens().isEmpty();
    }
}
```

---

6. Polymorphism

Exemplo:
Uma classe usa polimorfismo para tratar comportamentos diferentes.

```
abstract class Pagamento {
    public abstract void processarPagamento();
}

class PagamentoCartao extends Pagamento {
    @Override
    public void processarPagamento() {
        System.out.println("Pagamento por cartão processado.");
    }
}

class PagamentoBoleto extends Pagamento {
    @Override
    public void processarPagamento() {
        System.out.println("Pagamento por boleto processado.");
    }
}

class Pedido {
    public void pagar(Pagamento pagamento) {
        pagamento.processarPagamento(); // Uso do polimorfismo
    }
}
```

---

7. Pure Fabrication

Exemplo:
Criar uma classe que encapsula lógica auxiliar, como persistência.

```
class Logger {
    public void log(String mensagem) {
        System.out.println("LOG: " + mensagem);
    }
}
```

---

8. Indirection

Exemplo:
Uma camada intermediária para gerenciar dependências.

```
class MediadorPagamento {
    private Pagamento pagamento;

    public MediadorPagamento(Pagamento pagamento) {
        this.pagamento = pagamento;
    }

    public void executarPagamento() {
        pagamento.processarPagamento();
    }
}
```

---

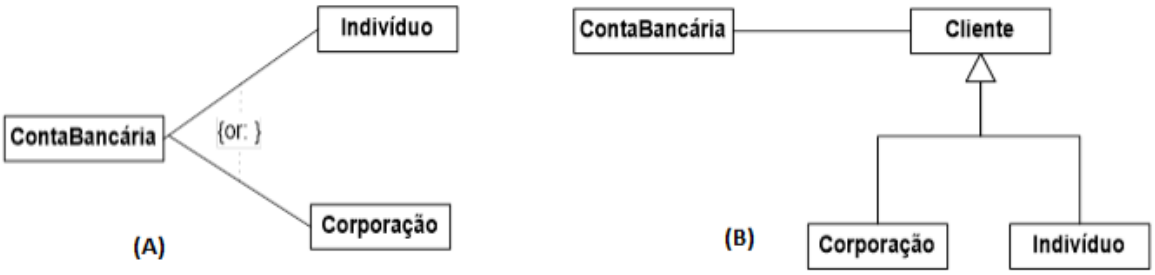
9. Protected Variations

Exemplo:
Proteger variações usando uma abstração.

```
interface Repositorio {
    void salvar(Object objeto);
}

class RepositorioArquivo implements Repositorio {
    @Override
    public void salvar(Object objeto) {
        System.out.println("Objeto salvo no arquivo.");
    }
}

class RepositorioBanco implements Repositorio {
    @Override
    public void salvar(Object objeto) {
        System.out.println("Objeto salvo no banco de dados.");
    }
}

class ServicoCadastro {
    private Repositorio repositorio;

    public ServicoCadastro(Repositorio repositorio) {
        this.repositorio = repositorio; // Proteção contra variações
    }

    public void cadastrar(Object objeto) {
        repositorio.salvar(objeto);
```

```
    }
}
```

Questão 2 - Considere os diagramas de classes de análise fornecidos na Figura 1, nos itens (a) e (b) abaixo, ambos de acordo com a notação da UML. Esses diagramas desejam representar o fato de que uma conta bancária pode ´ estar associada a uma pessoa, que pode ser ou uma pessoa física (representada pela classe Indivíduo), ou uma pessoa jurídica (representada pela classe Corporac¸ao). Uma dessas duas soluções é melhor que a outra? Se sim, qual delas e em que sentido? Justifique sua resposta considerando alguns dos padrões GRASP.



Figuur 1: Diagramas com as duas soluções.

Observações:

Diagrama (A):

- Uma **ContaBancária** se associa diretamente a uma entidade que pode ser um **Indivíduo** ou uma **Corporação**. Isso é representado pela relação "or" (ou).

Diagrama (B):

- Uma **ContaBancária** se associa a um **Cliente**, que é uma classe genérica. **Cliente** é uma superclasse para as subclasses **Indivíduo** e **Corporação**, representando polimorfismo.

Comparações utilizando o padrão GRASP

1. Information Expert (Especialista em Informação):

- **(A):** A **ContaBancária** diretamente associa-se a duas classes específicas (**Indivíduo** ou **Corporação**), mas não utiliza uma estrutura centralizada para gerenciar clientes.
- **(B):** A superclasse **Cliente** centraliza as informações e responsabilidades de características comuns (como "nome", "endereço", etc.), sendo mais coesa e um ponto único de acesso.

Vantagem do (B): O padrão "Expert" é melhor aplicado porque as responsabilidades de cliente (comuns a indivíduos e corporações) são centralizadas na superclasse **Cliente**.

2. Low Coupling (Baixo Acoplamento):

- **(A):** A **ContaBancária** precisa lidar com múltiplas dependências (duas classes distintas: **Indivíduo** e **Corporação**), aumentando o acoplamento.
- **(B):** A **ContaBancária** interage apenas com a classe genérica **Cliente**, reduzindo dependências diretas.

Vantagem do (B): O diagrama (B) promove menor acoplamento, pois abstrai as particularidades de **Indivíduo** e **Corporação** para a superclasse **Cliente**.

3. Protected Variations (Proteção contra Variações):

- **(A):** Adicionar novas categorias de clientes (como "ONGs") exigiria mudanças significativas no diagrama, afetando diretamente **ContaBancária**.
- **(B):** Novas categorias podem ser adicionadas como subclasses de **Cliente** sem impactar a estrutura de **ContaBancária**.

Vantagem do (B): O padrão "Protected Variations" é melhor aplicado, protegendo **ContaBancária** de mudanças futuras.

Conclusão

A solução do **Diagrama (B)** é superior em termos de design orientado a objetos. Ela utiliza **polimorfismo**, promove **baixo acoplamento (relação)**, centraliza responsabilidades seguindo o padrão **Information Expert** e protege contra futuras variações (**Protected Variations**). Essa abordagem é mais flexível, extensível e coesa, enquanto o **Diagrama (A)** apresenta maior acoplamento e complexidade na interação direta com classes específicas.

Questão 3 - responsabilidade correta relacionada ao padrão **GRASP Creator (Criador)** é:

O padrão **Creator** sugere que a responsabilidade de criar um objeto deve ser atribuída a uma classe que satisfaça pelo menos um dos seguintes critérios:

1. **Possui** o objeto que será criado.
   Ex.: Uma classe **Pedido** cria instâncias de **ItemPedido**, porque gerencia os itens.
2. **Contém** os dados necessários para inicializar o objeto.
   Ex.: Uma classe **Fatura** pode criar uma **LinhaFatura**, pois ela possui os valores para configurar a linha.

3. **Usa** o objeto criado.
   Ex.: Um **CarrinhoDeCompras** cria instâncias de **Produto**, porque manipula e mantém uma lista deles.
4. **É responsável por registrar ou armazenar** o objeto.
   Ex.: Um **GerenciadorDeConexoes** cria instâncias de **Conexao**, pois ele mantém controle sobre elas.
5. **Está logicamente associada** ao objeto criado.
   Ex.: Uma classe **Cliente** cria instâncias de **Endereço**, pois um cliente possui um ou mais endereços.