Departamento de Computação – DCOMP - IFMA

ALUNO: Artur Costa Tourinho

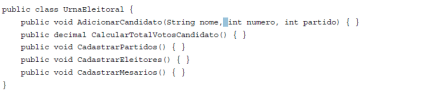
PROFESSOR: João Carlos Pinheiro

Atividade Etapa 01 – Parte 01

Princípios de Design Orientado a Objetos

Questões teóricas (4 pontos)

**1. Os princípios SOLID reúnem cinco boas práticas para projetos Orientados a Objetos-OO. Considere a classe UrnaEleitoral.**



*Com base no princípio SOLID e nas boas práticas para projetos OO quais são os problemas dessa classe. Sugira melhorias neste código.*

Um dos princípios SOLID é o Open-Closed Principle(OCP) que descreve que o código deve estar aberto para extensão mas fechado para modificação, no exemplo acima não há espaço para extender, somente modificar cada uma das classes.

Além disso o ISP (Interface Segregation Principle), também nos mostra que é melhor criar interfaces mais específicas, o que não é o caso da classe UrnaEleitoral que chama todas as outras classes em sua composição. Poderia ser criado a interface Cadastro, já que a ação de cadastrar é uma ação que se repete em todas as funções presentes em UrnaEleitoral, o que consequentemente também contribuiria ao OCP, pois o código estaria mais aberto as extensões.

Ex:

public interface Cadastramento{

public void CadastrarPartido():

}

public Class Solid{

private CadastrarPartido solid;

public void setSolid(CadastrarPartido solid){

this.solid = solid

}

public void executaSolid(){

solid.CadastrarPartido ();

}

}

Class CadastraPartido implementes Cadastramento{

public void CadastrarPartido();

}

**2. Para reaproveitarmos comportamento com composição, precisamos escrever métodos que queremos reaproveitar, criando os métodos e chamando os correspondentes na instância. Esses métodos que só delegam o trabalho para a instância no atributo são conhecidos como *delegate methods*. Quais vantagens e desvantagens essa abordagem apresenta sobre herança?**

A herança favorece o uso de elementos comuns entre uma superclasse e uma subclasse, entretanto aumenta consideravelmente o acoplamento do código de forma que uma classe pode ter que ser alterada em decorrência da outra.

No método de delegação é possível usar um objeto de outra classe como uma variável, assim utilizando suas funcionalidades mantendo o baixo acoplamento, não estando sujeito a todas as alterações e funções(por vezes não necessárias) de uma superclasse.

**3. Por que acoplamento é tão indesejado em projetos orientados a objetos?**

O acoplamento faz com que duas classes diferentes estejam diretamente ligadas, assim uma modificação feita em uma classe refletirá na outra, o que atrapalha a coesão, funcionalidade e reuso do código.

**4. Como o princípio do OCP nos ajuda a escrever classes mais flexíveis?**

O open closed principal descreve que entidades de software devem ser abertos para extensão e fechados para modificação, assim deve-se desenvolver classes coesas onde é possível alterar o comportamento de uma classe, sem necessidade de alterar o seu código fonte.

**5. O que é o DIP? E qual a vantagem de sempre depender de classes estáveis?**

O princípio de Inversão de dependência descreve que abstrações devem depender de abstrações e implementações devem depender de abstrações, assim diminuindo o risco de acoplamento, consequentemente a propagação de problemas.

As abstrações são a base das classes estáveis que tendem a mudar pouco, assim, não propagando essas transformações às classes dependentes.

**6. Como o OCP e o princípio de Liskov se relacionam?**

A relação entre OCP e liskov nos mostra que devemos tomar certos cuidados ao fazer uso da herança, de forma que ao usarmos herança as pré-condições não podem ser restringidas e as pós-condições não podem ser expandidas, assim, a subclasse deve respeitar as condições da superclasse, não sendo necessário a alteração na superclasse quando a subclasse for alterada.

**7. O que é a Lei de Demeter? O que o desenvolvedor ganha quando a segue?**

A lei de Demeter nos mostra que devemos evitar o encadeamento de chamada de métodos.

**8. Qual a alternativa para se reaproveitar comportamento, sem fazer uso de herança?**

A alternativa à herança é a composição, onde ao compor um objeto, você não precisa se preocupar com as pré e pós condições, pois o relacionamento se dá entre os objetos e não entre as classes, apresentando menor acoplamento e maior encapsulamento.

**9. Para cada um dos seguintes princípios de design GRASP, pesquise e dê exemplos usando trechos de código, de como seria (1) ANTES e (2) APÓS a aplicação de cada um desses princípios.**

Princípios de Design OO a serem considerados:

**Código Antes:** Foi usado o mesmo código apresentado na questão 11.

public class PagadorDeFuncionario {

public void pagaChefe(Chefe chefe) {

chefe.depositaNaConta(chefe.getSalarioBase() +

chefe.getBonificacoes());

}

public void pagaFuncionario(Funcionario funcionario) {

funcionario.depositaNaConta(funcionario.getSalario() +

funcionario.getBonus());

}

public void pagaEstagiario(Estagiario estagiario) {

estagiario.paga( estagiario.getBolsa() +

estagiario.getAuxilios());

}

}

**● Criador**

Criação de Classe PagadorDeFuncionario

public class PagadorDeFuncionario {

public static void main(String[] args) {

Contexto context = new Contexto();

int escolha;

System.out.println("Voce Deseja?\n1 - Pagar Chefe\n2 - Pagar Funcionario\n3 - Pagar Estagiario\n");

Scanner scanner = new Scanner(System.in);

escolha = scanner.nextInt();

if (escolha == 1){

context.setStrategy(new PagaChefe());

}else if(escolha == 2){

context.setStrategy(new PagaFuncionario());

}else if(escolha == 3){

context.setStrategy(new PagaEstagiario());

}else{

return;

}

**● Especialista na Informação**

Criação da Classe Contexto, que delimita o domínio da função Pagar

public class Contexto {

private PagadorStrategy strategy;

public void setStrategy(PagadorStrategy strategy) {

this.strategy = strategy;

}

public void executaStrategy(){

strategy.pagar();

}

}

**● Acoplamento Baixo**

No exemplo inicial a classe depende de objetos concretos, já no caso refatorado, o código parte da abstração, da idéia comum do pagar.

**● Controlador**

Definição de Interface para para manipular a função Pagar:

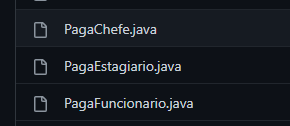
public interface PagadorStrategy {

public void pagar();

}

**● Coesão Alta**

Cada Classe é focada apenas em sua função específica:



Exemplo de PagaEstagiario:

public class PagaEstagiario implements PagadorStrategy {

public void pagar() {

System.out.println("Pagando Estagiario");

}

}