

Atividade Etapa 01 – Parte 01

Princípios de Design Orientando a Objetos

Questões teóricas

1. Veja o trecho de código abaixo:

```
if(obj == "tipo1") { /* faz alguma coisa */ }  
else if (obj == "tipo2") { /* faz outra coisa */ }  
else if (obj == "tipo3") { /* e assim sucessivamente! */ }  
// continua com outros else if
```

Você já encontrou algum código, cheio de `ifs` iguais aos mostrados acima?
Qual o problema com esse tipo de `if`, e como fazer para evitá-los?

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?
3. Coesão:
- a. Como fazer para detectar classes não coesas?
 - b. Como fazer para que uma classe se torne coesa?
4. Por que acoplamento é tão indesejado em projetos orientados a objetos?
5. Como o princípio do OCP nos ajuda a escrever classes mais flexíveis?
6. O que é o DIP? E qual a vantagem de sempre depender de classes estáveis?
7. Como o OCP e o DIP se relacionam?

8. O que é a Lei de Demeter? O que o desenvolvedor ganha quando a segue?
9. Qual a alternativa para se reaproveitar comportamento, sem fazer uso de herança?
10. 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:

- Criador
- Especialista na Informação
- Acoplamento Baixo
- Controlador
- Coesão Alta

Questões Práticas

OBS: Para essas questões deverá ser implementado o código refatorado

11. Imagine que, no sistema financeiro de uma empresa, o código que gera o pagamento dos funcionários é o seguinte:

```
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());
    }
}
```

Dica: Repare que as três classes - Chefe, Funcionario e Estagiario - possuem um método para depositar o pagamento, um para calcular o valor base do pagamento e um para calcular os extras. Será que não dá para extrair essa estrutura comum para uma interface?

12. Muitas pessoas optam por investir o dinheiro das suas contas bancárias. Existem diversos tipos de investimentos, desde investimentos conservadores até mais arrojados. Independente do investimento escolhido, o titular da conta recebe apenas 75% do lucro do investimento, pois 25% é imposto.

Implemente um mecanismo que invista o valor do saldo dela em um dos vários tipos de investimento e, dado o retorno desse investimento, 75% do valor é adicionado no saldo da conta.

Crie a classe **RealizadorDeInvestimentos** que recebe uma estratégia de investimento, a executa sobre uma conta bancária, e adiciona o resultado seguindo a regra acima no saldo da conta.

Os possíveis tipos de investimento são:

- CONSERVADOR, que sempre retorna 0.8% do valor investido;
- MODERADO, que tem 50% de chances de retornar 2.5%, e 50% de chances de retornar 0.7%;
- ARROJADO, que tem 20% de chances de retornar 5%, 30% de chances de retornar 3%, e 50% de chances de retornar 0.6%.

Para simular as chances utilize a classe `Random` do pacote `java.util`. Segue um exemplo: para verificar se a chance é maior que 30% use:

```
Random random = new Random()

boolean escolhido = random.nextDouble() > 0.30;
```

13. Seja o código abaixo, que mostra uma possível modelagem de um sistema financeiro. Temos uma classe `Movimentacao` que tem alguns métodos para calcular impostos. A classe `Pagamento` estende a classe `Movimentacao`, customizando parte do comportamento. A classe `Deposito` também estende a classe `Movimentacao`.

```
public class Movimentacao {
    private double valor;
    private Conta conta;
    private Calendar data;

    public double getEncargos() {
        return valor * 0.01;
    }
    // getters e setters
}

public class Pagamento extends Movimentacao {
    private String favorecido;
    private String formaDePagamento;
    // getters e setters
}

public class Deposito extends Movimentacao {
    private String numeroEnvelope;
```

```
public double getEncargos() {  
    throw  
        new RuntimeException("Depositos não sofrem encargos");  
}  
// getters e setters  
}
```

Você vê algum problema com essa abordagem? Comente e apresente uma solução que usa composição em vez de herança para resolver o esse problema.