CONCURSO PÚBLICO
MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO DE JANEIRO

EDITAL Nº 006/2022

PADRÃO DE RESPOSTAS DA PROVA DISCURSIVA REALIZADA DOMINGO, 15 DE MAIO DE 2022.
PRAZO PARA RECURSO CONTRA O PADRÃO DE RESPOSTAS: 16 E 17 DE MAIO DE 2022, NO ENDEREÇO ELETRÔNICO:
http://www.selecon.org.br

PADRÃO DE RESPOSTAS PRELIMINAR

SJM-01

INFORMÁTICA
Engenharia de Software; Desenvolvimento de Sistemas; Programação

Nº DA QUESTÃO	Espera-se que o candidato(a) desenvolva os aspectos/conteúdos propostos a seguir.
1	O candidato deverá desenvolver o(s) conteúdo(s) com base nos seguintes aspectos: A) Identificar todos os princípios **(3,0 pontos)** B) Conceitualizar corretamente todos os princípios **(3,0 pontos)** C) Exemplos corretos **(4,0 pontos)** **(Retirado de:** ANICHE, M. Orientação a Objetos e SOLID para Ninjas. Projetando classes flexíveis. [s.l.] Casa do Código, 2015.) S: Single Responsiblity Principle (Princípio da responsabilidade única)

O princípio da responsabilidade única determina que uma classe deve um e apenas um objetivo, ou seja, ela deve possuir apenas uma função ou funções similares.

```
//início de código
class CalculadoraDeSalario {
    public double calcula(Funcionario funcionario) {
        if (DESENVOLVEDOR.equals(funcionario.getCargo())) {
            return dezOuVintePorcento(funcionario);
        }
        if (DBA.equals(funcionario.getCargo()) ||
            TESTER.equals(funcionario.getCargo())) {
            return quinzeOuVinteCincoPorcento(funcionario);
        }
        throw new RuntimeException("funcionario invalido");
    }
}
//fim de código
```

Tente generalizar esse exemplo de código. Códigos como esse são bastante comuns: é normal olhar para uma característica do objeto e, de acordo com ela, tomar alguma decisão. Repare que existem apenas 3 cargos diferentes (desenvolvedor, DBA e tester) com regras similares. Mas em um sistema real, essa quantidade seria grande. Ou seja, essa classe tem tudo para ser uma daquelas classes gigantescas, cheias de if e else, comque estamos acostumados. Ela não tem nada de coesa.
Imagine só essa classe com 15, 20, 30 cargos diferentes. A sequência de ifs seria um pesadelo. Além disso, cada cargo teria sua implementação de cálculo diferente, ou seja, mais algumas dezenas de métodos privados. Agora tente complicar um pouco essas regras de cálculo. Um caos.
Classes não coesas ainda têm outro problema: a chance de terem defeitos é enorme. Talvez nesse exemplo seja difícil de entender o argumento, mas como essas muitas regras estão uma perto da outra, é fácil fazer com que uma regra influencie a outra, ou que um defeito em uma seja propagado para a outra. Entender e manter esse arquivo não é uma tarefa fácil ou divertida de
ser feita.

O: Open-Closed Principle (Princípio Aberto-Fechado)
O princípio aberto-fechado determina que as classes de um sistema devem ser abertas para extensões e fechadas para modificações, ou seja, outras classes podem requisitar os métodos públicos que classe possui, porém, não podem alterá-los.

```
//Início de código
```

```
public class CalculadoraDePrecos {
    public double calcula(Compra produto) {
        TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();
        Frete correios = new Frete();
        double desconto =
            tabela.descontoPara(produto.getValor());
        double frete = correios.para(produto.getCidade());
        return produto.getValor() * (1 - desconto) + frete;
    }
}
public class TabelaDePrecoPadrao {
    public double descontoPara(double valor) {
        if (valor > 5000) return 0.03;
        if (valor > 1000) return 0.05;
        return 0;
    }
}
public class Frete {
    public double para(String cidade) {
        if ("SAO PAULO".equals(cidade.toUpperCase())) {
            return 15;
        }
        return 30;
    }
}
//Fim de código
```

Imagine que o sistema é mais complicado que isso. Não existe apenas uma única regra de cálculo de desconto, mas várias; e também não existe apenas uma única regra de frete, existem várias. Uma maneira (infelizmente) comum de vermos código por aí é resolvendo isso por meio de ifs. Ou seja, o código decide se é a regra A ou B que deve ser executada. Se esse número de regras for razoavelmente grande, a ideia não é boa: esse código ficará complicadíssimo, cheio de ifs, e difícil de ser mantido; testar fica cada vez mais difícil, afinal a quantidade de caminhos a serem testados é grande; a classe deixará de ser coesa, pois conterá muitas diferentes regras (e você já sabe os problemas que isso implica).

L: Liskov Substitution Principle (Princípio da substituição de Liskov)
O princípio da substituição de Liskov determina que uma classe derivada deve ser substituível por sua classe base.

```
//Início de código
public class ContaComum {
    protected double saldo;
```

```
    public ContaComum() {
        this.saldo = 0;
    }
    public void deposita(double valor) {
        if (valor <= 0)
            throw new ValorInvalidoException();
        this.saldo += valor;
    }
    public double getSaldo() {
        return saldo;
    }

    public void rende() {
        this.saldo *= 1.1;
    }
}
//Fim de código
```

Ela representa, de maneira simplificada, uma conta em um banco. A classe possui operações simples como deposita() e rende(). Mas, como sempre, o sistema precisa crescer. Imagine agora a classe ContaDeEstudante, que é exatamente igual a uma conta, com a diferença de que ela não “rende”. Usando herança, a implementação seria algo parecido com a que segue, onde o método rende() lança uma exceção:

```
//Início de código
public class ContaDeEstudante extends ContaComum {
    public void rende() {
        throw new ContaNaoRendeException();
    }
}
//Fim de código
```

É difícil enxergar o problema dessa simples sobrescrita. Para isso, imagine um código que faz uso de ambas ContaComum e ContaDeEstudante:

```
//Início de código
public class ProcessadorDeInvestimentos {
    public static void main(String[] args) {
        for (ContaComum conta: contasDoBanco()) {
            conta.rende();
            System.out.println("Novo Saldo:");
            System.out.println(conta.getSaldo());
```

```
        }
    }
}
//Fim de código
```

O método contasDoBanco() retorna uma lista com diferentes contas. Não sabemos exatamente quais estão lá dentro, mas, dado o polimorfismo, podemos tratar todas elas pela referência da classe pai. Agora o problema: qual o comportamento da aplicação? Não sabemos. Afinal, se houver alguma conta de estudante nesse código, a execução do programa parará, pois uma exceção será lançada. Pense que o sistema possui diversos desses loops e classes que interagem bem com ContaComum (e, por consequência, interagem com qualquer filho dela também). A nova classe ContaDeEstudante pode fazer essas classes pararem de funcionar também. Por que isso aconteceu? Porque a classe filha quebrou o contrato (em Java, informal) definido pela classe pai: o método rende() na classe pai não lança exceção. Ou seja, as classes clientes não vão esperar que isso aconteça, e não vão tratar essa possibilidade. Note que as classes filhas precisam respeitar os contratos definidos pela classe pai. Mudar esses contratos pode ser perigoso.

I: Interface Segregation Principle (Princípio da Segregação da Interface)
O princípio da segregação da interface determina que as classes não devem ser forçadas a implementar métodos que não irão utilizar.

Em um jogo hipotético, temos a interface Ave, com os métodos setLocalizacao(long, lat), setAltitude (alt) e renderizar(). A classe Papagaio implementa esta interface de forma adequada. No entanto, se tivermos a classe Pinguim, ela implementará a classe setAltitude, sem a real necessidade.

D: Dependency Inversion Principle (Princípio da Inversão da Dependência)
O princípio da Inversão de Dependência determina que uma classe deve depender de abstrações e não de implementações.

```
//Início de código
public class GeradorDeNotaFiscal {
    private final EnviadorDeEmail email;
    private final NotaFiscalDao dao;
    public GeradorDeNotaFiscal(EnviadorDeEmail email,
        NotaFiscalDao dao) {
        this.email = email;
        this.dao = dao;
    }
    public NotaFiscal gera(Fatura fatura) {
        double valor = fatura.getValorMensal();
        NotaFiscal nf = new NotaFiscal(
```

	`` valor, impostoSimplesSobreO(valor)); email.enviaEmail(nf); dao.persiste(nf); return nf; } private double impostoSimplesSobreO(double valor) { return valor * 0.06; } } //Fim de código ``` A classe GeradorDeNotaFiscal é acoplada ao EnviadorDeEmail e NotaFiscalDao. Pense agora o seguinte: hoje, esse código em particular manda e-mail e salva no banco de dados usando um DAO. Imagine que amanhã esse mesmo trecho de código também mandará informações para o SAP, disparará um SMS, consumirá um outro sistema da empresa etc. A classe GeradorDeNotaFiscal vai crescer, e passar a depender de muitas outras. Qual o problema disso? O grande problema do acoplamento é que uma mudança em qualquer uma das classes pode impactar em mudanças na classe principal. Ou seja, se o EnviadorDeEmail parar de funcionar, o problema será propagado para o GeradorDeNotaFiscal. Se o NFDao parar de funcionar, o problema será propagado para o gerador. E assim por diante. Podemos pensar não só em defeitos, mas em problemas de implementação. Se a interface da classe SAP mudar, essa mudança será propagada para o GeradorDeNotaFiscal. Portanto, o problema é: a partir do momento em que uma classe possui muitas dependências, todas elas podem propagar problemas para a classe principal. O reuso dessas classes também fica cada vez mais difícil, afinal, se quisermos reutilizar uma determinada classe em outro lugar, precisaremos levar junto todas suas dependências. Lembre-se também que as dependências de uma classe podem ter suas próprias dependências, gerando uma grande árvore de classes que devem ser levadas junto. Total previsto de linhas para a resposta final do(a) candidato(a): **2 laudas**
2	Para a definição informal, o candidato deverá falar de tabelas e atributos. **(Até 3 pontos)** Para a definição formal, o candidato deverá desenvolver conteúdo que defina o modelo relacional, por meio da descrição dos seguintes aspectos: o domínio como um conjunto de valores atômicos, a existência dos tipos de dados, ou formato, para domínios, os esquemas de relação, formados de nome de relação e lista de atributos, os atributos possuem um domínio, Aridade/grau do esquema de relação, e a relação, ou estado da relação, indicando tupla, intenção e extensão da relação. **(Até 4 pontos)** Além disso, deve dar exemplos de domínios, esquemas de relação e relação (com valores). Para os exemplos. **(até 3 pontos)**

Exemplo de resposta:
Um Banco de Dados Relacional é um repositório de dados composto unicamente por tabelas, destinado a registrar as informações que representam o estado de uma sistemas de informação.
Cada tabela é formada por linhas e colunas. As linhas representam fatos de um mesmo tipo sobre o qual se deseja guardar informação, enquanto as colunas representam atributos a serem registrados sobre esses fatos, ou seja, as informações específicas que são guardadas sobre Elmasri e Navathe (2019). Esses fatos se referem a objetos do mundo real, eventos, momentos, contratos, etc., e são diretamente ligados às instâncias de entidades ou relacionamentos.

Uma célula $a_{i,j}$ da tabela A indica o valor do atributo j para um fato i. Essa maneira de representar o mundo é conhecida como Modelo Relacional. A figura a seguir mostra informalmente uma pequena parte de um banco de dados relacional com três tabelas, criado a partir do modelo ER da Figura, tratando de avaliações de especialistas sobre o custo de recuperação de terrenos com problemas de contaminação.

Cada especialista é descrito por seu nome, em que empresa trabalha e qual sua especialidade. Cada local é descrito por um código, o tipo de terreno e o desenvolvimento da região. Uma terceira tabela indica qual o custo de recuperação de um local de acordo com um especialista. Pela forma como foi construída, cada especialista só pode dar uma avaliação por local, porém todas as combinações de local e especialistas são possíveis, porém apenas algumas são verdadeiras e são registradas na tabela.

Especialista

nome	empresa	especialidade
João	ServAmb	Qualidade de Água
Carlos	EcoConsult	Produtos Químicos
Ana	ServAmb	Vazamentos
...	...	...

Local

site_id	terreno	desenvolvimento
L171	Pantanoso	Baixo
M123	Subúrbio	Moderado
...	...	...

Avaliação

nome	site_id	custos
João	L171	100K
Paulo	M123	20K
Ana	L171	80K
...	...	...

É interessante também possuir uma noção da definição formal de um banco de dados relacional, como a apresentada por Elmasri e Navathe (2019)
O Modelo Relacional usa o conceito de relação matemática como base de sua construção. Ele foi proposto por Codd em 1970.

A descrição formal do modelo é a seguinte.

Um **domínio** *D* é um conjunto de valores atômicos. Normalmente um domínio é especificado de acordo com um **tipo de dados**, que define todos os valores possíveis dos valores desse domínio.

Exemplos de domínios são: datas, números de telefone, nomes de pessoa, quantidade em dinheiro, etc.

Também é comum que seja especificado um formato para cada domínio. Por exemplo, um formato válido para número de telefone pode ser "+999 999 99999-9999", indicando três números para o código do país, três para o código de cidade e nove para o número do telefone. Já um domínio relacionado a dinheiro pode ter seu valor máximo limitado e uma indicação de como será representado.

Sendo assim, para definir um domínio são necessários:
• um nome,
• um tipo de dados,
• um formato e
• outras informações adicionais que auxiliem a interpretação do. domínio

A definição do domínio é arbitrária, mas segue a lógica de um conceito desejado em um contexto específico.

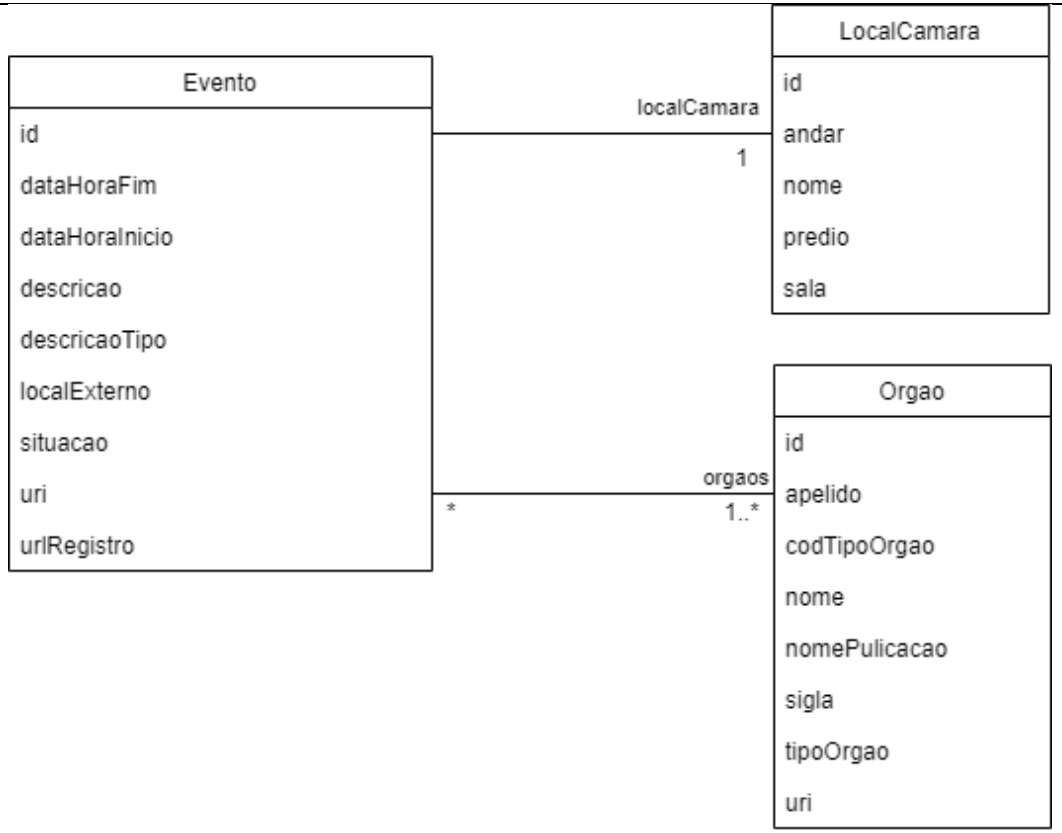
Um **esquema de relação** é denotado por
$R(A1,A2,...,An)$
atributo *Ai* é um papel que algum domínio assume na relação *R*. *D* é chamado o domínio de *Ai*, $D = dom(Ai)$.
O **grau da relação** é o número de atributos *n*. O esquema da relação descreve uma **relação** chamada *R*.

Os esquemas de relação da Figura são:
• Especialista(nome, empresa, especialidade)
• Local(site_id, terreno, desenvolvimento)
• Avaliação(nome, site_id, custos).

Uma **relação** do esquema de relação $R(A1,A2,...,An)$, denotada por $r(R)$ é um conjunto de n-tuplas $r = \{t1,t2,...,tk\}$, onde cada tupla é uma lista ordenada de valores $t =< v1,v2,...,vn >$ onde cada vj, $1 \leq j \leq n \Rightarrow vj \in dom(Aj) \vee vj$ = nulo.

Ou seja, na definição formal, um esquema de relação explica como é organizada a relação, que é a tabela, já preenchida com os dados, da definição informal. Para um mesmo esquema de relação podem existir várias, possivelmente infinitas, relações possíveis. Em um certo instante do tempo, porém, provavelmente apenas uma relação representa corretamente a informação necessária para um sistema de informação.

	Como uma relação não é ordenada, então as tuplas de uma relação não são ordenadas. É importante notar que dentro do Modelo Relacional só existem tabelas. Assim, para consultar a base de dados são feitas operações com tabelas que geram outras tabelas, com o resultado desejado da consulta imaginada. Total previsto de linhas para a resposta final do(a) candidato(a): **2 laudas**
3	O candidato deverá desenvolver o(s) conteúdo(s) com base nos seguintes aspectos: A) Representação correta dos objetos e chaves do modelo em JSON nas classes UML **(5 pontos)** B) Correção sintática e semântica da instrução SQL **(5 pontos)**




```
SELECT e.descricao, l.nome, l.predio, e.urlRegistro, o.sigla
FROM Evento e
INNER JOIN LocalCamara l ON e.localCamaraId = l.id
INNER JOIN EventoOrgao eo ON e.id = eo.EventoId
INNER JOIN Orgao o ON o.id = eo.OrgaoId
```

Total previsto de linhas para a resposta final do(a) candidato(a): **1 lauda**