1. Descreva, detalhadamente, o padrão **Injeção de Dependência** (*Dependency Injection*) e explique como esse padrão pode ser usado em uma aplicação que utilize o framework Spring.

2. Discorra, detalhadamente, sobre as diferenças e semelhanças entre os padrões **Repository** (repositórios) e **DAO**, envolvendo os aspectos arquiteturais de cada um desses padrões.

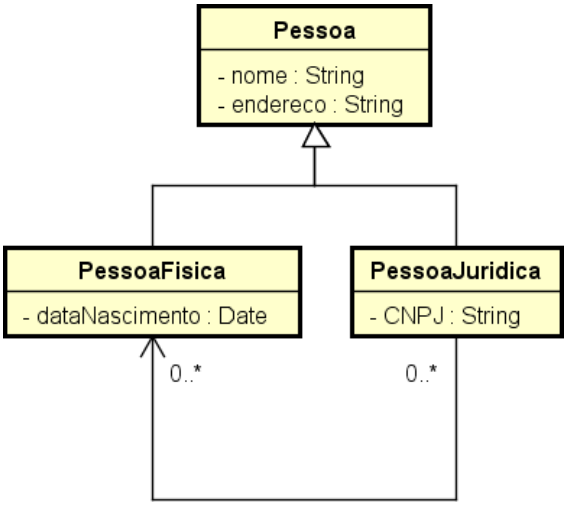
3. Descreva, detalhadamente, o conceito de **DAO genérico.** Informe os passos para utilizar esse padrão na implementação de um repositório chamado AlunoRepositorio, para manipular objetos da classe Aluno (no contexto do SCA). Forneça esboços de código-fonte em Java para ilustrar e exemplificar sua descrição.

4. Considere o diagrama de classes a seguir e forneça:

   (a) a expressão em JPQL que produza a lista de todos os nomes e endereços das pessoas físicas.

   (b) um esboço de implementação da classe **PessoaJuridica** com as anotações adequadas do JPA para esta classe de entidade.

   (c) a implementação de um construtor para a classe **PessoaJuridica** que inicie todos os seus atributos. Considere a existência de uma classe **CNPJ**, com um método estático para validar valores de **CNPJ** e cuja assinatura é a seguinte:
   `Boolean validar (String cnpj)`




5. Considere que, em uma aplicação Java Swing, existem duas classes (ambas subclasses de JFrame): **JPerspectivaTextualVotacao** e **JPerspectivaGraficaVotacao**. A classe **JPerspectivaTextualVotacao** permite visualizar e alterar as quantidades de votos atribuidos a candidatos em uma votação (por simplicidade, consideramos uma quantidade constante de candidatos). A classe **JPerspectivaGraficaVotacao** permite visualizar o resultado da votação através de um gráfico de pizza. Nessa aplicação, quando o usuário modifica a quantidade de votos de algum candidato (no formulário correspondente a um objeto da classe **JPerspectivaTextualVotacao**), o gráfico de pizza apresentado deve ser automaticamente atualizado. Considere que, nessa aplicação, há uma classe de negócio denominada **Votação**, que é responsável por manter as quantidades de votos de cada um dos candidatos e por prover métodos seletores e modificados (get/set) adequados para tal. Considere a classe **Observable** e a interface **Observer**, disponíveis na API da linguagem Java, que implementam os componentes Subject e Observer, respectivamente, do padrão GoF Observer.

   Descreva, detalhadamente, uma solução para a situação descrita acima, através do uso do padrão GoF Observer, e forneça um diagrama de classes correspondente à solução.

CEFET/RJ

PROVA: Arq. e Padrões de SW. DATA: 27/04/15
PROFESSOR:
ALUNO: Luana da Silva Fragoso Nº: TURMA:

1) Vamos considerar duas classes A e B, na qual A depende de B. Ou seja, A possui um objeto B, como no exemplo 1.

```
public class A {
    B b;
    ...
}
```
Exemplo 1

O problema a se implantar na dependência entre as classes é como instanciar a classe B em A. Pelo padrão de Injeção de Dependência, não se deve instanciar a classe B em A (como no exemplo 2), pois isso não traz flexibilidade ao código. Para contornar esse problema, passa-se como parâmetro, no construtor A, por exemplo, um objeto b. Ou seja, a responsabilidade da instanciação passa a ser de outro objeto, e (como no exemplo 3) este apenas passa o objeto b para A. Isso permite uma maior flexibilidade, pois A não conhece B, então a classe B pode ser uma interface, por exemplo, e o código na classe A não precisará ser mexido. Essa solução também diminui o acoplamento, já que A não conhece B. Resumindo, Injeção de Dependência é a passagem de um objeto, que pertence a uma classe B, de que uma classe A depende. Essa passagem pode ser por construtor, método ou utilizando um framework (exemplo: Spring). A anotação @Autowired do framework Spring faz todo esse processo de Injeção internamente. Isso permite que o framework tenha mais controle no código, invocando partes do código apenas quando necessário (quando ocorre dependência). Exemplo 4 mostra a utilização da anotação.

1

log4j.xml já esteja configurado corretamente.

Exemplo ②:

```
public class A {
    B b = new B();
    ...
}
```

Exemplo ③:

```
public class A {
    B b;
    public A(B b) {
        this.b = b;
    }
```

Exemplo ①:

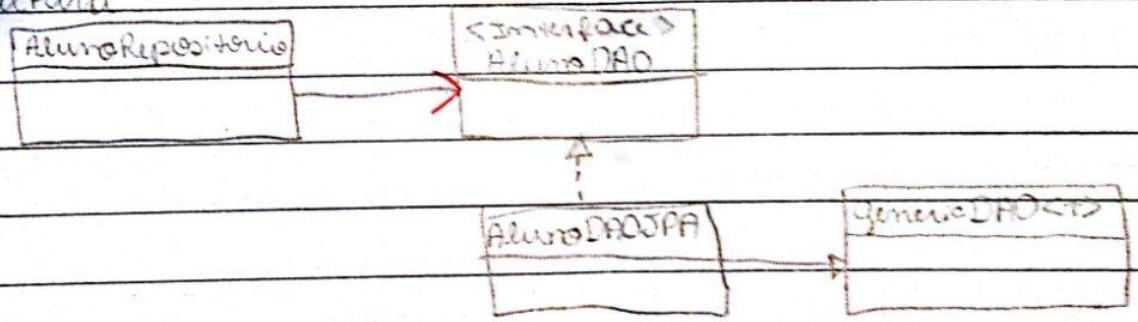
```
public class A {
    @Autowired
    B b;
    ...
}
```

2) O Repositório está na camada de domínio, diferente do DAO que está na camada de infraestrutura. A existência do repositório é importante, pois, quando se precisa acessar um banco de dados, ele fica responsável pelas regras de negócios envolvidas e delega a tarefa de acesso para o DAO. Com isso, isola-se melhor essas duas camadas permitindo melhor flexibilidade, manutenabilidade e etc. A semelhança é que ambos procuram as mesmas informações de um objeto. Por exemplo, quero pegar informações sobre um usuário u, tendo o repositório, eu vou mandar mensagem para o UsuárioRepositório pedindo para me dar informações sobre o usuário u. Em seguida, o repositório irá delegar para o DAO, que então vai me devolver tais informações. Sem utilizar o repositório, eu poderia mandar mensagem diretamente para o DAO, e ele, então, iria me devolver as mesmas informações. Logo, para um objeto que manda mensagem para o repositório ou para o dao, se espera a mesma funcionalidade nos dois. Porém, vale lembrar que o repositório é importante para isolar as regras de negócio (do sistema) do acesso ao banco de dados

PROVA: Arq. e Padrões de SW DATA 27/04/15
PROFESSOR
ALUNO: Luana da Silva Fragoso Nº TURMA

3) O DAO genérico tem como função agrupar todos os métodos comuns encontrados nos DAO's, o CRUD principalmente. Assim, com o uso do DAO genérico se evita várias repetições de código nos DAO's. Em Java, o GenericDAO possui o conceito de Generics (GenericDAO<T>). Com isso, o conceito de DAO genérica é possível, pois é só passar no generics (T) o tipo de classe que você quer manipular, exemplo: GenericDAO<Aluno>, nesse DAO genérico pois Aluno vai ter todos os métodos base (CRUD, por exemplo) para a manipulação de alunos.

Estrutura




Para utilizar o GenericDAO, deve-se ter uma amarração entre o AlunoRepositório e o AlunoDAO para poder fazer a manipulação de aluno no Repositório (camada de domínio). O DAOJPA deve então implementar a interface AlunoDAO, isso faz com que ocorra o isolamento da tecnologia (JPA, no caso) da camada de domínio. Como a implementação do acesso ao banco fica no DAOJPA, então essa classe extende a GenericDAO, para assim, evitar escrever códigos básicos (exemplo, CRUD).

Código em Java:

```
public class AlunoRepositorio {
    @Autowired
    AlunoDAO dao;
    ...
    public Aluno getAluno(String cpf) {
        return dao.getAluno(cpf)
    }
    ...
}

public interface AlunoDAO {
    Aluno getAluno(String cpf);
    ...
}

public class AlunoDAOJPA implements AlunoDAO extends
                                    GenericDAO<Aluno> {
    public Aluno getAluno(String cpf) {
        ...
        super.getAluno(query, parameters);
    }
}

public class GenericDAO<T> {
    ...
    public Aluno getAluno(String query, Object... parameters) {
        return aluno;
    }
}
```

4) a) SELECT p.nome, p.endereco FROM PessoaFisica p

b) @Entity
public class PessoaJuridica extends Pessoa {
private string cnpj;
@ManyToMany
~~@JoinTable(name = "P_JUR_FIS", joinColumns = {@JoinColumn(name = "JUR_ID", referencedColumnName = "ID")}, inverseJoinColumns = {@JoinColumn(name = "FIS_ID", referencedColumnName = "ID")})~~
private List<PessoaFisica> pf;

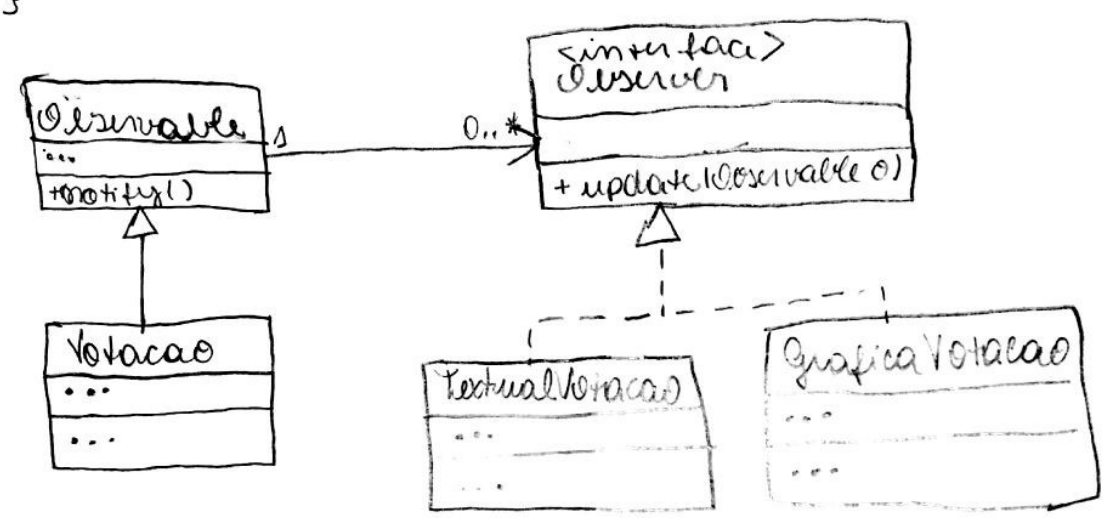
}

4) a) SELECT p.nome, p.endereco FROM PessoaFisica p

b) @Entity
public class PessoaJuridica extends Pessoa {
private string cnpj;
@ManyToMany
@JoinTable(name="PJ_PF", joinColumns={@JoinColumns(name="JUR_ID", referencedColumn="ID")}, inverseColumns={@JoinColumns(name="FIS_ID", referencedColumn="ID")})
private Set<PessoaFisica> pf;

4)c)
```
public class PessoaJuridica extends Pessoa {
    private String cnpj;
    ...
    public PessoaJuridica(String nome, String end, String cnpj) {
        // A validação do nome e do endereço ocorre na classe Pessoa
        super(nome, end);
        if (!Cnpj.validar(cnpj)) {
            throws new IllegalArgumentException("Cnpj invalido");
        }
        this.cnpj = cnpj;
    }
    ...
}
```

5)




A classe Votação extende Observable, porque é nela que ocorre alterações da quantidade de votos. Assim, quando ocorrer uma alteração, logo em seguida, a classe Votação deve chamar o método notify de sua classe pai (Observable). Essa classe possui a lista de todos os observadores da classe Votação (devido a associação). Os observadores são adicionados em tempo de execução. Quando o objeto Votação tem alguma atração, o método notify (dessa lista) é chamado. Ele vai executar o update de cada Observer da lista. Como Observer é uma interface, cada classe que a implementa vai implementar o método update (que tem como parâmetro a classe Observable) de uma forma específica. Mas é pelo seu parâmetro que eles possuem as informações da classe Votacao (no caso do exemplo) e, assim, podem atualizar seus dados também.