# Documentação do Projeto: Mapa de Doenças

## 1. Introdução

Este projeto visa consolidar os conhecimentos de Programação Orientada a Objetos (POO) através da criação de uma aplicação prática no tema de crowdsourcing, focando na persistência de dados em um banco de dados relacional. A aplicação permite o registro e consulta de informações sobre doenças, sintomas e locais, além de gerenciar relatos de usuários.

## 2. Requisitos Técnicos e Implementação

### 2.1 Pilares da Orientação a Objetos

O projeto aplica os quatro pilares da Programação Orientada a Objetos:

- **Abstração**: Entidades como `Doenca`, `Sintoma`, `Local`, `Usuario` e `Relato` foram modeladas para representar conceitos do domínio do problema. Classes abstratas e interfaces são utilizadas para definir comportamentos comuns.
- **Encapsulamento**: O uso de modificadores de acesso (`private`, `public`, `protected`, `package-private`) é aplicado apropriadamente para proteger o estado interno dos objetos e controlar o acesso aos seus atributos e métodos. Por exemplo, os construtores das classes modelo (`Doenca`, `Local`, `Sintoma`, `Usuario`, `Relato`) são privados, e a criação de instâncias é feita por meio de métodos fábrica (`criarNovaDoenca`, `reconstruir`, `criarNovoSintoma`, `criar`, etc.), garantindo a integridade dos objetos. Métodos como `atribuirId` são `package-private`, permitindo que apenas classes do mesmo pacote (como as implementações DAO) modifiquem o ID após a persistência.
- **Herança**: A classe abstrata `Doenca` é estendida por `DoencaGrave`, `DoencaLeve` e `DoencaModerada`, demonstrando uma hierarquia de classes com reutilização de código e especialização do comportamento (`getGrauDeRisco()`).
- **Polimorfismo**: O polimorfismo é evidenciado na sobrescrita do método `getGrauDeRisco()` nas subclasses de `Doenca` (`DoencaGrave`, `DoencaLeve`, `DoencaModerada`). Além disso, o método `adicionarRelato` na classe `Usuario` apresenta sobrecarga, aceitando tanto um objeto `Relato` quanto os parâmetros individuais (`Doenca`, `Local`, `Date`).

### 2.2 Classes Abstratas e Interfaces

O projeto utiliza classes abstratas e interfaces conforme os requisitos:

- **Classe Abstrata**: A classe `Doenca` é abstrata e possui um método abstrato `getGrauDeRisco()`, que é implementado pelas suas subclasses concretas (`DoencaLeve`, `DoencaModerada`, `DoencaGrave`).
- **Interface**: A interface `IdentificavelPorNome` é implementada por `Doenca`, `Sintoma` e `Local`, definindo um contrato para classes que podem ser identificadas por um nome.

## 2.3 Relacionamentos entre Classes

O projeto demonstra diversidade de cardinalidades e direcionamento nos relacionamentos:

- **1:N (Um para Muitos)**:
  - Um Usuario pode ter vários Relatos. A classe Usuario possui uma List<Relato>.
  - Uma Doenca pode ter muitos Sintomas, e um Sintoma pode estar associado a muitas Doencas (representado por uma relação N:N no banco de dados e gerenciado pelas DAOs). A classe Doenca possui um Set<Sintoma> e a classe Sintoma possui um Set<Doenca>.
- **Unidirecional e Bidirecional**:
  - O relacionamento entre Usuario e Relato é unidirecional (Usuario conhece seus Relatos).
  - O relacionamento entre Doenca e Sintoma é bidirecional, com ambos os objetos tendo referências um ao outro (Doenca tem uma lista de Sintomas e Sintoma tem uma lista de Doencas), garantindo a consistência dos dados quando uma associação é feita. A adição de sintomas a uma doença, por exemplo, chama um método interno no Sintoma para adicionar a doença associada.
- **Composição/Agregação**:
  - A classe Relato demonstra composição ou agregação, pois um Relato é composto por um Usuario, uma Doenca e um Local. No construtor de Relato, os objetos Usuario, Doenca e Local são recebidos como parâmetros.

## 2.4 Collections

O projeto faz uso adequado de diferentes tipos de Collections:

- **List**: A classe Usuario utiliza ArrayList para armazenar os Relatos de um usuário. As DAOs (DoencaDAOImpl, SintomaDAOImpl, UsuarioDAOImpl) também utilizam ArrayList para retornar listas de objetos.
- **Set**: A classe Doenca utiliza um HashSet para armazenar os Sintomas associados, garantindo a unicidade dos sintomas por doença. Similarmente, a classe Sintoma usa HashSet para as Doencas associadas. Métodos como add e remove são utilizados, e coleções não modificáveis são retornadas (Collections.unmodifiableSet, Collections.unmodifiableList).
- **Map**: A classe GerenciadorDeApelido utiliza um HashMap para armazenar usuários por apelido, garantindo a unicidade e o acesso rápido.

## 2.5 Persistência de Dados

A persistência de dados é implementada com JDBC e o padrão DAO:

- **JDBC**: A classe ConnectionFactory é responsável por estabelecer a conexão com o banco de dados MySQL via JDBC.
- **Padrão DAO**: O padrão Data Access Object (DAO) é utilizado para isolar a lógica de acesso a dados das classes de modelo. Interfaces DAO (DoencaDAO, SintomaDAO, LocalDAO, UsuarioDAO) definem os contratos para as operações CRUD, e suas

implementações (DoencaDAOImpl, SintomaDAOImpl, LocalDAOImpl, UsuarioDAOImpl) contêm a lógica SQL para interagir com o banco de dados.

- **Operações CRUD**: As implementações DAO fornecem métodos para as quatro operações básicas de manipulação de dados (CRUD):
    - **C - Create**: Inserir um novo registro (ex: DoencaDAOImpl.criar(), SintomaDAOImpl.criar(), UsuarioDAOImpl.criar()).
    - **R - Read**: Ler ou recuperar dados (ex: DoencaDAOImpl.buscarPorId(), SintomaDAOImpl.listarTodos(), UsuarioDAOImpl.buscarPorApelido()).
    - **U - Update**: Atualizar informações existentes (ex: DoencaDAOImpl.atualizar(), SintomaDAOImpl.atualizar(), UsuarioDAOImpl.atualizar()).
    - **D - Delete**: Remover registros (ex: DoencaDAOImpl.deletar(), SintomaDAOImpl.deletar(), UsuarioDAOImpl.deletar()).

# 3. Estrutura do Projeto

O projeto segue uma estrutura de pacotes organizada para separar as responsabilidades:

- **dao**: Contém as interfaces DAO (Data Access Object) e suas implementações, responsáveis pela interação com o banco de dados.
    - DoencaDAO.java
    - DoencaDAOImpl.java
    - LocalDAO.java
    - LocalDAOImpl.java (classe vazia no momento, mas esperada a implementação aqui)
    - SintomaDAO.java
    - SintomaDAOImpl.java
    - UsuarioDAO.java
    - UsuarioDAOImpl.java
- **model**: Contém as classes que representam as entidades do domínio da aplicação.
    - Doenca.java (classe abstrata)
    - DoencaGrave.java
    - DoencaLeve.java
    - DoencaModerada.java
    - GerenciadorDeApelido.java
    - IdentificavelPorNome.java (interface)
    - Local.java
    - Relato.java
    - Sintoma.java
    - Usuario.java
- **util**: Contém classes de utilidade, como a ConnectionFactory para conexão com o banco de dados.
    - ConnectionFactory.java

# 4. Instruções de Execução

Para executar o projeto, siga os passos abaixo:

1. **Configuração do Banco de Dados**:
    - Garanta que o MySQL esteja instalado e em execução.
    - Crie um banco de dados chamado `MapaDoenca`.
    - Ajuste as credenciais de conexão no arquivo `src/util/ConnectionFactory.java`, alterando o `PASSWORD` para a sua senha do MySQL.
    - Execute o script SQL de criação do banco de dados (este script não foi fornecido, mas é um requisito para a entrega ). Ele deve criar as tabelas `doencas`, `sintomas`, `doenca_sintoma`, `usuarios`, `locais` e `relatos`.
    -
2. **Compilação e Execução**:
    - O projeto pode ser compilado e executado a partir de um IDE (como IntelliJ IDEA, Eclipse ou NetBeans) ou via linha de comando.
    - **Indicação da Classe com `main`**: A classe principal com o método `main` para execução da aplicação não foi fornecida nos arquivos do GitHub. No entanto, para fins de demonstração, uma classe `Main` (ou similar) no pacote raiz ou em um pacote de `app` seria a responsável por iniciar a aplicação e interagir com as camadas de modelo e DAO.

---