## **Documento PDF com Descrição, Justificativas e Evidências**

### **Relatório de Projeto: Cidademais - Plataforma de Crowdsourcing**

**Disciplina:** Programação Orientada a Objetos

**Professora:** Talita Vieira Ribeiro

**Alunos:** Bryan Amorim, Pedro Macedo, Gustavo Salvador

### **1. Introdução**

Este documento apresenta a implementação do projeto "Cidademais", uma aplicação Java desenvolvida no contexto da disciplina de Programação Orientada a Objetos, com o objetivo de consolidar os conhecimentos adquiridos sobre os pilares da POO, uso de classes abstratas e interfaces, relacionamentos entre classes, manipulação de coleções e persistência de dados em banco relacional utilizando JDBC e o padrão DAO.

O "Cidademais" é uma plataforma de *crowdsourcing* que permite aos cidadãos reportar e contribuir com problemas urbanos, como buracos em vias, postes de luz queimados, entre outros. A aplicação gerencia informações de usuários (cidadãos), localização dos problemas, detalhes dos problemas, comentários e as relações de contribuição entre cidadãos e problemas.

### **2. Requisitos Técnicos e Implementação**

#### **2.1 Pilares da Orientação a Objetos**

O projeto "Cidademais" demonstra a aplicação robusta dos quatro pilares da Programação Orientada a Objetos, conforme evidenciado na modelagem e implementação das suas classes:

* **Abstração:**
  + **Descrição:** A abstração foi utilizada para modelar as entidades do domínio, focando nos conceitos essenciais e ignorando detalhes irrelevantes para o contexto da aplicação. Classes como Usuario, Cidadao, Problema, Comentario e Localizacao representam, de forma clara e concisa, os elementos fundamentais do sistema de *crowdsourcing*. Por exemplo, a classe Problema abstrai as características comuns a qualquer problema que um cidadão possa reportar, como título, descrição, status e localização.
  + **Justificativa/Evidência:** A criação da classe abstrata Usuario (conforme Usuario.java) é um exemplo primário de abstração, definindo características comuns (ID, nome) para qualquer tipo de usuário, mesmo que o sistema atualmente só possua Cidadao. Isso prepara a arquitetura para futuras extensões, como a inclusão de um tipo de usuário Administrador ou FuncionárioPúblico, sem a necessidade de grandes refatorações.
* **Encapsulamento:**
  + **Descrição:** O encapsulamento foi rigorosamente aplicado em todas as classes, protegendo o estado interno dos objetos e expondo apenas as interfaces necessárias para interação. Todos os atributos das classes são declarados com o modificador de acesso private, e seu acesso ou modificação é feito exclusivamente através de métodos public (getters e setters).
  + **Justificativa/Evidência:** Observa-se em todas as classes de modelo (e.g., Cidadao.java, Problema.java, Localizacao.java, Comentario.java) que os atributos são private. Por exemplo, em Cidadao.java, problemasReportados é privado, e a única forma de adicionar um problema é através do método adicionarProblemaReportado(Problema problema), que pode conter lógicas de validação interna (como verificar se o problema já existe na lista antes de adicionar). Isso evita manipulações diretas e inconsistentes dos dados.
* **Herança:**
  + **Descrição:** A herança foi empregada para estabelecer uma hierarquia entre classes, permitindo a reutilização de código e a especialização de comportamentos. A classe Cidadao herda características e comportamentos básicos da classe abstrata Usuario.
  + **Justificativa/Evidência:** A declaração public class Cidadao extends Usuario em Cidadao.java é a evidência clara do uso de herança. Cidadao herda id e nome de Usuario, evitando duplicação de código. Além disso, Cidadao sobrescreve métodos abstratos de Usuario, especializando a forma como um "cidadão" é representado.
* **Polimorfismo:**
  + **Descrição:** O polimorfismo foi aplicado através da sobrescrita de métodos abstratos e da implementação de interfaces, permitindo que objetos de diferentes classes sejam tratados de forma uniforme quando se referem a um tipo comum.
  + **Justificativa/Evidência:**
    - A classe Usuario define os métodos abstratos getTipoUsuario() e exibirResumo(). A classe Cidadao sobrescreve esses métodos (e.g., Cidadao.java - getTipoUsuario() retorna "Cidadão"), demonstrando polimorfismo por herança. Isso permite que uma lista de Usuarios possa conter instâncias de Cidadao e cada uma se comporte de maneira específica ao chamar esses métodos.
    - A interface Exibivel (conforme Exibivel.java) define o método exibirDetalhes(). As classes Problema (Problema.java) e Comentario (Comentario.java) implementam essa interface e fornecem suas próprias versões do método exibirDetalhes(), adaptadas às suas respectivas necessidades. Isso permite que, por exemplo, uma lista de objetos Exibivel possa ser iterada e cada objeto exiba seus detalhes de forma particular, sem que o código cliente precise saber o tipo concreto do objeto.

#### **2.2 Classes Abstratas e Interfaces**

O projeto faz uso estratégico de classes abstratas e interfaces para promover a flexibilidade, a reutilização de código e a definição de contratos.

* **Classe Abstrata:**
  + **Descrição:** A classe Usuario foi projetada como abstrata para servir como uma base comum para diferentes tipos de usuários no sistema. Ela define atributos (id, nome) e métodos concretos (getId, getNome, setNome) que são comuns a todos os usuários. Mais importante, ela introduz métodos abstratos (getTipoUsuario(), exibirResumo()) que devem ser obrigatoriamente implementados por suas subclasses concretas.
  + **Justificativa/Evidência:** Conforme Usuario.java, a classe é declarada public abstract class Usuario. A existência de métodos abstratos garante que qualquer subclasse de Usuario (como Cidadao) será forçada a definir como um usuário exibe seu resumo e qual é o seu tipo, assegurando a consistência na API sem a necessidade de instanciar diretamente um Usuario genérico que não possui um significado concreto na aplicação.
* **Interface:**
  + **Descrição:** A interface Exibivel foi criada para definir um contrato que qualquer objeto que possa ter seus "detalhes exibidos" deve seguir. Isso promove um design mais desacoplado e flexível, permitindo que diferentes tipos de entidades implementem a mesma funcionalidade de exibição de maneira polimórfica.
  + **Justificativa/Evidência:** A interface Exibivel (Exibivel.java) possui apenas um método void exibirDetalhes();. As classes Problema (Problema.java) e Comentario (Comentario.java) ambas implementam Exibivel. Por exemplo, Problema.java implementa exibirDetalhes() mostrando o título, descrição, status e localização do problema, enquanto Comentario.java implementa exibirDetalhes() mostrando o autor, texto e data/hora do comentário. Isso demonstra que diferentes objetos podem ser tratados através da interface Exibivel para realizar uma ação comum.

#### **2.3 Relacionamentos entre Classes**

A modelagem de dados do projeto reflete os relacionamentos complexos entre as entidades do domínio, utilizando diversas cardinalidades:

* **1:N (Um para Muitos) - Bidirecional (Cidadão reporta Problema):**
  + **Descrição:** Um Cidadao pode reportar múltiplos Problemas, e cada Problema é reportado por exatamente um Cidadao. A relação é bidirecional, o que significa que tanto o Cidadao "conhece" os problemas que reportou, quanto o Problema "conhece" quem o reportou.
  + **Evidência:**
    - Em Cidadao.java: private List<Problema> problemasReportados; e o método adicionarProblemaReportado(Problema problema).
    - Em Problema.java: private Cidadao reportante; no construtor e getReportante().
    - A lógica no construtor de Problema (if (reportante != null) { reportante.adicionarProblemaReportado(this); }) garante a consistência da relação bidirecional quando um problema é criado.
* **1:N (Um para Muitos) - Unidirecional (Cidadão faz Comentário):**
  + **Descrição:** Um Cidadao pode fazer múltiplos Comentarios. No entanto, o Comentario conhece seu Cidadao autor, mas o Cidadao não mantém explicitamente uma lista de *todos* os comentários que fez.
  + **Evidência:**
    - Em Comentario.java: private Cidadao autor; e getAutor().
    - Em Cidadao.java: Não há uma lista direta de Comentarios feitos por ele.
* **1:N (Um para Muitos) - Unidirecional (Problema tem Comentários):**
  + **Descrição:** Um Problema pode ter múltiplos Comentarios. A relação é unidirecional do Problema para Comentario.
  + **Evidência:**
    - Em Problema.java: private List<Comentario> comentarios; e o método adicionarComentario(Comentario comentario).
    - Em Comentario.java: Não há referência ao Problema ao qual pertence diretamente no modelo Java, essa associação é gerenciada primariamente pela DAO e a tabela comentarios no banco de dados.
* **1:1 (Um para Um) - Unidirecional (Problema tem Localização):**
  + **Descrição:** Cada Problema está associado a uma única Localizacao exata. Uma Localizacao está associada a somente um Problema. A relação é unidirecional do Problema para Localizacao.
  + **Evidência:**
    - Em Problema.java: private Localizacao localizacao; e getLocalizacao().
    - Em Localizacao.java: Não há referência ao Problema ao qual pertence. A unicidade no banco de dados (id\_problema INT <<FK, UNIQUE>> na tabela localizacoes) garante a cardinalidade 1:1.
* **N:N (Muitos para Muitos) - Unidirecional (Cidadãos contribuem com Problemas):**
  + **Descrição:** Múltiplos Cidadaos podem contribuir para múltiplos Problemas. No modelo de domínio, a classe Cidadao mantém uma lista dos problemas aos quais contribuiu, e a classe Problema mantém uma lista dos cidadãos que contribuíram. Esta relação é materializada no banco de dados através de uma tabela associativa (contribuicoes).
  + **Evidência:**
    - Em Cidadao.java: private List<Problema> problemasContribuidos; e adicionarProblemaContribuido(Problema problema).
    - Em Problema.java: private List<Cidadao> contribuintes; e adicionarContribuinte(Cidadao cidadao).
    - A existência da tabela contribuicoes no script SQL (script\_criacao\_agenda.sql - assumindo que foi adaptado para este projeto, ou que este projeto terá um similar) com chaves primárias compostas (id\_cidadao, id\_problema) é a evidência da implementação desta cardinalidade no banco de dados.

#### **2.4 Collections**

O uso adequado de coleções Java foi fundamental para gerenciar múltiplos objetos e suas relações, garantindo eficiência nas operações de adição, busca e remoção.

* **Uso de List (ArrayList):**
  + **Descrição:** ArrayList é utilizada para armazenar coleções ordenadas de objetos que podem conter elementos duplicados e onde a ordem de inserção é importante.
  + **Justificativa/Evidência:**
    - Em Cidadao.java: private List<Problema> problemasReportados; e private List<Problema> problemasContribuidos;. A escolha de ArrayList é adequada para manter a ordem em que os problemas foram reportados/contribuídos e para permitir fácil iteração.
    - Em Problema.java: private List<Comentario> comentarios; e private List<Cidadao> contribuintes;. Da mesma forma, ArrayList é apropriada para gerenciar as coleções de comentários e contribuintes, mantendo a ordem de adição.
    - **Exemplo de uso (similar ao Evento.java do projeto de exemplo):** No arquivo Evento.java do projeto de exemplo, a lista de pessoas (private ArrayList<Pessoa> pessoas;) e o método addPessoa() demonstram o armazenamento de múltiplos objetos relacionados, similar ao que é feito para comentários e contribuições em Problema.
* **Uso de Map (HashMap):**
  + **Descrição:** HashMap é utilizada para armazenar pares chave-valor, permitindo a recuperação eficiente de um valor a partir de sua chave. É ideal quando a busca por um objeto específico por um identificador único é frequente.
  + **Justificativa/Evidência:**
    - Em Cidadao.java: private Map<Integer, Problema> problemasReportadosPorId;. Esta coleção é utilizada para otimizar as operações de busca e remoção de problemas reportados por ID (removerProblemaReportado(int id)). Em vez de iterar sobre uma lista para encontrar o problema, o Map permite acesso direto em tempo quase constante (O(1)).
    - **Exemplo de uso (comparação com exemplos):** Embora o projeto de exemplo não use Map de forma explícita no modelo como Cidadao faz, o conceito de buscar por um ID é fundamental. PessoaDAO.java e EventoDAO.java implementam buscarPorId, que é uma operação baseada no conceito de chave-valor. O uso de Map no modelo Java de Cidadao complementa essa funcionalidade para a gestão em memória.
* **Uso de Set (HashSet) - *Correção/Aprimoramento*:**
  + **Descrição:** Originalmente, seu README mencionava o uso de Set para tags. Se a classe Problema realmente tiver um Set<String> tags;, isso seria ideal para armazenar elementos únicos e sem ordem específica, como categorias ou palavras-chave.
  + **Justificativa/Evidência (Se implementado):** Se a classe Problema tivesse private Set<String> tags;, seria um exemplo de Set para garantir que não haja tags duplicadas para um problema, como "buraco" e "buraco". Isso otimizaria o armazenamento e a verificação da existência de uma tag específica. *No entanto, com base no Problema.java fornecido, esta funcionalidade não está implementada. Se for um requisito, deve ser adicionado ao código.*
  + **Recomendação:** Avalie se a classe Problema realmente precisa de tags. Se sim, adicione a private Set<String> tags; e os métodos adicionarTag e removerTag e ajuste seu Problema.java. Se não, remova esta seção do README e do PDF.
* **Operações de adição, busca e remoção:**
  + **Descrição:** O projeto implementa métodos específicos para manipular os elementos dentro das coleções, garantindo a integridade dos dados e a lógica de negócio.
  + **Justificativa/Evidência:**
    - adicionarProblemaReportado(Problema problema) em Cidadao.java demonstra adição, incluindo uma verificação para evitar duplicação.
    - removerProblemaReportado(int id) em Cidadao.java demonstra remoção utilizando o Map para eficiência.
    - adicionarComentario(Comentario comentario) em Problema.java demonstra adição.
    - **Comparativo com exemplo:** Métodos como addTelefone() em Pessoa.java e addPessoa() em Evento.java do projeto de exemplo são análogos às operações de adição nas listas do seu projeto.

#### **2.5 Persistência com JDBC + DAO**

A persistência dos dados é gerenciada através de um banco de dados relacional (SQL) e acesso via JDBC, seguindo o padrão Data Access Object (DAO) para desacoplar a lógica de negócio da lógica de persistência.

* **JDBC (Java Database Connectivity):**
  + **Descrição:** O JDBC é a API padrão do Java para conexão e interação com bancos de dados relacionais. Ele é utilizado para estabelecer conexões, executar consultas SQL, atualizar dados e gerenciar transações.
  + **Justificativa/Evidência:**
    - A classe ConnectionFactory (não fornecida diretamente, mas implícita na Main.java e bd.ConnectionFactory import) é responsável por criar e gerenciar as conexões com o banco de dados.
    - Em todas as classes DAO (e.g., CidadaoDAO, ProblemaDAO), observa-se o uso de java.sql.Connection, PreparedStatement e ResultSet para executar operações SQL.
    - **Exemplo de uso (PessoaDAO.java e EventoDAO.java do projeto de exemplo):** Ambos os arquivos demonstram o uso de Connection injetada via construtor (private Connection connection; public PessoaDAO(Connection connection)), PreparedStatement para consultas parametrizadas (pstm.setString(1, pessoa.getNome());), ResultSet para leitura de dados (rst.next(), rst.getInt(1)), e execute() ou executeUpdate() para execução de DML.
* **Padrão DAO (Data Access Object):**
  + **Descrição:** O padrão DAO isola a lógica de persistência de dados do restante da aplicação. Cada DAO é responsável pelas operações CRUD (Create, Read, Update, Delete) de uma entidade específica, garantindo que as regras de negócio não sejam misturadas com os detalhes de acesso ao banco de dados.
  + **Justificativa/Evidência:**
    - A interface BaseDAO (BaseDAO.java) define um contrato genérico para operações de persistência (salvar, buscarPorId, listarTodosLazyLoading, listarTodosEagerLoading, atualizar, excluir).
    - Classes como CidadaoDAO, LocalizacaoDAO, ProblemaDAO, ComentarioDAO e ContribuicaoDAO (inferidas pelos imports em Main.java) implementam essa interface, cada uma lidando com a persistência de sua respectiva entidade de modelo.
    - **Exemplo de uso (PessoaDAO.java e EventoDAO.java do projeto de exemplo):**
      * Ambas as classes implementam BaseDAO.
      * O método salvar(Object objeto) em PessoaDAO e EventoDAO demonstra como um objeto de modelo é transformado em dados para inserção no banco de dados.
      * Os métodos buscarPorId e listarTodosEagerLoading (ou similar) demonstram como os dados do ResultSet são mapeados de volta para objetos de modelo.
      * A separação clara entre a lógica de negócio (modelo.\* classes) e a lógica de persistência (dao.\* classes) é evidente. Por exemplo, a classe Main.java interage com os DAOs para salvar e recuperar objetos, sem se preocupar com os detalhes internos de como a persistência é feita.

#### **2.6 Organização do Código e Documentação**

* **Organização do Código:**
  + **Descrição:** O código-fonte está organizado em pacotes lógicos (modelo, dao, bd) para separar as responsabilidades e melhorar a modularidade.
  + **Justificativa/Evidência:** A estrutura de pacotes é visível em todos os arquivos Java fornecidos e nos imports. Por exemplo, package modelo; em Cidadao.java e package dao; em BaseDAO.java.
* **Documentação:**
  + **Descrição:** O código inclui comentários Javadoc ou inline para explicar a finalidade das classes, métodos e atributos complexos, contribuindo para a legibilidade e manutenção.
  + **Justificativa/Evidência:** [Revise seu código e mencione onde há bons comentários ou onde você planeja adicioná-los. Se não houver, mencione que essa é uma área para aprimoramento.]