Estruturas de Dados Utilizadas no Projeto

Integrantes:

- Bruno da Silva Ribeiro
- Isadora Teixeira Santoma
- Nicolas Soeiro
- Ricardo Liyudi Tetsuya
- Stephany Aliyah Guimarães Eurípedes de Paula

1. Introdução

O objetivo deste relatório é descrever as principais estruturas de dados e conceitos de Programação Orientada a Objetos (POO) utilizados no desenvolvimento do projeto. A aplicação foi desenvolvida com o Node.js como ferramenta de backend, utilizando classes, heranças, abstração e arrays para organizar e manipular os dados dos usuários, viagens e avaliações. A análise demonstra exemplos práticos extraídos do código-fonte e explicando suas aplicações.

2. Aplicação dos Conceitos de Programação Orientada a Objetos (POO)

2.1 Classes

Uma classe é a estrutura fundamental da Programação Orientada a Objetos (POO). Ela define um molde para criar objetos, agrupando atributos (dados) e métodos (funções) que representam o comportamento e as características de uma entidade do mundo real.

As seguintes classes representam as entidades principais do sistema:
- Usuario
- UsuarioMotorista (herda de Usuario)
- UsuarioPassageiro (herda de Usuario)
- Viagem
- Avaliacao
- RelataPerigo
- ZonaPerigo
- Alerta

Cada classe possui atributos e comportamentos próprios, permitindo representar informações reais utilizadas na aplicação. Por exemplo, a classe **Usuario** possui atributos como **nome, email** e **senha**, além de métodos para acessar e manipular esses dados. O uso de classes facilita a organização do sistema, melhora a legibilidade e promove a reutilização de código.

2.2. Classe Abstrata

Uma classe abstrata serve como modelo base para outras classes. Ela não pode ser instanciada diretamente e pode conter métodos abstratos (sem implementação) e métodos comuns.

A classe **Usuario** pode ser implementada como uma classe abstrata, pois define atributos e métodos comuns que são compartilhados entre **UsuarioMotorista** e **UsuarioPassageiro**, mas não é instanciada diretamente. Isso promove a reutilização e polimorfismo.

Exemplo: Classe Usuario

```
public abstract class Usuario {
  private int id;
  private String nome;
  private String email;
  public Usuario(int id, String nome, String email) {
    this.id = id;
    this.nome = nome;
    this.email = email;
  }
  public abstract void exibirTipoUsuario();
  public String getNome() {
    return nome;
  }
  public String getEmail() {
    return email;
  }
}
```

2.3 Herança

A herança é um dos pilares da POO e permite que uma classe (subclasse) herde atributos e métodos de outra (superclasse). Isso promove a reutilização de código e facilita a criação de estruturas mais organizadas

Exemplo: UsuarioMotorista e **UsuarioPassageiro**

```
public class UsuarioMotorista extends Usuario {
  private String placaVeiculo;
  public UsuarioMotorista(int id, String nome, String email, String placaVeiculo) {
    super(id, nome, email);
    this.placaVeiculo = placaVeiculo;
  }
  @Override
  public void exibirTipoUsuario() {
    System.out.println("Usuário do tipo: Motorista");
  }
}

public class UsuarioPassageiro extends Usuario {
  private String cartaoCadastrado;
  public UsuarioPassageiro(int id, String nome, String email, String cartaoCadastrado) {
    super(id, nome, email);
    this.cartaoCadastrado = cartaoCadastrado;
  }
  @Override
  public void exibirTipoUsuario() {
```

```
    System.out.println("Usuário do tipo: Passageiro");
  }
}
```

2.4 Polimorfismo

O termo polimorfismo significa "muitas formas" e permite que um mesmo método se comporte de diferentes maneiras, dependendo do objeto que o invoca.
No projeto, é possível tratar tanto **UsuarioMotorista** quanto **UsuarioPassageiro** como objetos do tipo genérico **Usuario**, permitindo que métodos como **exibirTipoUsuario()** sejam invocados sem precisar saber o tipo específico da instância.

Exemplo: Classe Sistema

```
public class Sistema {
  public static void main(String[] args) {
    Usuario u1 = new UsuarioMotorista(1, "Carlos", "carlos@email.com", "ABC-1234");
    Usuario u2 = new UsuarioPassageiro(2, "Ana", "ana@email.com", "1234-5678");
    u1.exibirTipoUsuario(); // Motorista
    u2.exibirTipoUsuario(); // Passageiro
  }
}
```

2.5 Encapsulamento

O encapsulamento é a prática de tornar os atributos privados e fornecer métodos públicos para acessá-los. Isso protege os dados contra modificações indevidas e aumenta a segurança da aplicação.

Exemplo: Classe ZonaPerigo

```
public class ZonaPerigo {
  private String nome;
  private int nivelRisco;
  public String getNome() {
    return nome;
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  public int getNivelRisco() {
    return nivelRisco;
  }
  public void setNivelRisco(int nivelRisco) {
    if (nivelRisco >= 0 && nivelRisco <= 10) {
      this.nivelRisco = nivelRisco;
    }
  }
}
```

3. Estrutura de Dados

3.1 Arrays e ArrayList

Um array é uma estrutura de dados que permite armazenar vários valores do mesmo tipo em uma única variável. Ele possui tamanho fixo, ou seja, o número de elementos deve ser definido no momento da criação.

- **Array:** estrutura de tamanho fixo utilizada para armazenar múltiplos elementos do mesmo tipo.
- **ArrayList:** estrutura dinâmica da linguagem Java que permite adicionar e remover elementos com flexibilidade.

Exemplo: Classe Viagem

```
import java.util.ArrayList;
public class Viagem {
  private int id;
  private String origem;
  private String destino;
  private ArrayList<Avaliacao> avaliacoes;
  public Viagem() {
    this.avaliacoes = new ArrayList<>();
  }
  public void adicionarAvaliacao(Avaliacao avaliacao) {
    avaliacoes.add(avaliacao);
  }
  public void listarAvaliacoes() {
    for (Avaliacao a : avaliacoes) {
      System.out.println("Nota: " + a.getNota() + " | Comentário: " + a.getComentario());
    }
  }
  // Getters e setters opcionais para id, origem e destino
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getOrigem() {
    return origem;
  }
  public void setOrigem(String origem) {
    this.origem = origem;
  }
  public String getDestino() {
    return destino;
  }
  public void setDestino(String destino) {
    this.destino = destino;
  }
}
```

3.2 Fila (Queue)

A fila é uma estrutura de dados linear que segue o princípio FIFO (First In, First Out), ou seja, o primeiro elemento inserido é o primeiro a ser removido. Essa estrutura é amplamente utilizada em sistemas que precisam processar eventos na ordem em que ocorrem, garantindo que os elementos sejam tratados em sequência cronológica.

Exemplo: Classe AlertaManager

```
import java.util.LinkedList;
import java.util.Queue;
public class AlertaManager {
  private Queue<Alerta> filaDeAlertas;
  public AlertaManager() {
    this.filaDeAlertas = new LinkedList<>();
  }
  public void novoAlerta(Alerta alerta) {
    filaDeAlertas.add(alerta);
    System.out.println("Novo alerta adicionado: " + alerta.getMensagem());
  }
  public void processarAlertas() {
    while (!filaDeAlertas.isEmpty()) {
      Alerta alerta = filaDeAlertas.poll(); // Remove o primeiro
      System.out.println("Processando alerta: " + alerta.getMensagem());
    }
  }
}
```

3.3 Pilha (Stack)

A pilha é uma estrutura de dados baseada no princípio LIFO (Last In, First Out), ou seja, o último elemento inserido é o primeiro a ser removido. Essa estrutura é muito útil para funcionalidades onde há necessidade de voltar à ação anterior,

Exemplo: HistoricoZona

```
package aula;
import java.util.Stack;
public class HistoricoZona {
  private Stack<ZonaPerigo> historico;
  public HistoricoZona() {
    this.historico = new Stack<>();
  }
  public void visitarZona(ZonaPerigo zona) {
    historico.push(zona);
    System.out.println("Zona visitada: " + zona.getNome());
  }
  public void voltarZonaAnterior() {
    if (!historico.isEmpty()) {
      ZonaPerigo zonaAnterior = historico.pop();
      System.out.println("Voltando para a zona: " + zonaAnterior.getNome());
    } else {
```

```
            System.out.println("Nenhuma zona anterior.");
        }
    }
}
```

4. Considerações Finais

O projeto aplica de forma eficaz os princípios da Programação Orientada a Objetos, com classes bem definidas, herança e encapsulamento, além de demonstrar o uso prático de estruturas de dados fundamentais.
Essas escolhas promovem uma organização clara, facilitam a manutenção e expansão futura do sistema e contribuem para uma arquitetura sólida e escalável, pronta para futuras integrações.