FUNCIONAMENTO E DESAFIOS NA IMPLEMENTAÇÃO DO SIMULADOR DE ELEVADOR

João Lucas Martins Silva Carvalho João Tiago Lima Carvalho

Maio 2025

Resumo

O Simulador de Elevadores Inteligentes, desenvolvido como trabalho final da disciplina de Estrutura de Dados, modela o comportamento de elevadores em um prédio de múltiplos andares, com foco em mobilidade eficiente, priorização de usuários com necessidades especiais (cadeirantes e idosos) e análise de métricas como tempo de espera, consumo de energia e fluxo de pessoas. Implementado em Java, o sistema utiliza estruturas de dados personalizadas, como listas encadeadas e filas FIFO, para gerenciar andares, elevadores e chamadas de passageiros. A simulação opera em tempo discreto, orquestrada pela classe Simulador, que coordena a interação entre o prédio, a central de controle e uma interface gráfica desenvolvida em Java Swing. Três heurísticas foram implementadas: Ordem de Chegada (FCFS), Otimização de Tempo (priorizando tempo de espera e pessoas com necessidades especiais) e Otimização de Energia (minimizando a distância percorrida pelos elevadores). A interface gráfica exibe o estado dos elevadores, filas de espera por andar, horários de pico (7h-9h e 17h-19h) e estatísticas em tempo real, com configurações ajustáveis via interface. O projeto destaca sua modularidade e potencial de expansão, servindo como uma ferramenta educacional para o estudo de estruturas de dados, algoritmos e simulação discreta.

1. Introdução

Este relatório apresenta o funcionamento e os principais desafios enfrentados durante o desenvolvimento do Simulador de Elevadores Inteligentes para edifícios de múltiplos andares. O projeto foi desenvolvido como trabalho final da disciplina de Estrutura de Dados, com o objetivo de modelar o comportamento de elevadores, integrando conceitos de mobilidade eficiente, priorização de usuários com necessidades especiais, e análise de parâmetros como tempo de espera, consumo de energia e fluxo de pessoas, utilizando estruturas de dados personalizadas.

2. Funcionamento do Sistema de Simulação

2.1. Visão Geral do Funcionamento

O simulador modela um prédio com número configurável de andares (mínimo 5) e uma frota de elevadores inteligentes controlados por uma central única. Os usuários fazem chamadas através de um painel externo (que pode ser configurado para um único botão, dois botões ou um painel numérico) e selecionam seu destino por meio do painel interno dos elevadores. A simulação leva em conta

1. O tempo de viagem de cada elevador, que varia conforme o andar e o horário (horários de pico e fora de pico).
2. O tamanho da fila de espera em cada andar.
3. O tempo de espera de cada usuário.

4. Prioridades especiais para cadeirantes e pessoas idosas.
5. A capacidade máxima de cada elevador.

2.2. Implementação das Estruturas Básicas

Para implementar o simulador, desenvolvemos estruturas de dados personalizadas em vez de utilizar as coleções prontas do Java. As principais estruturas implementadas foram

FILA DEMONSTRAÇÃO:

PARTE 1:

```
public class Fila<T> {  no usages
    private Ponteiro<T> inicio = null;
    private Ponteiro<T> fim = null;
    private int tamanho = 0;

    public void inserirFim(T elemento) {
        Ponteiro<T> novo = new Ponteiro(elemento, (Ponteiro)null);
        if (this.inicio == null) {
            this.inicio = novo;
            this.fim = novo;
        } else {
            this.fim.setProximo(novo);
            this.fim = novo;
        }

        ++this.tamanho;
    }
```

FILA DEMONSTRAÇÃO:

PARTE 2:

```java
public T removerInicio() {
    if (this.inicio == null) {
        return null;
    } else {
        T elemento = (T)this.inicio.getElemento();
        this.inicio = this.inicio.getProximo();
        if (this.inicio == null) {
            this.fim = null;
        }

        --this.tamanho;
        return elemento;
    }
}

public Ponteiro<T> getInicio() { return this.inicio; }

public int tamanho() { return this.tamanho; }

public boolean estaVazia() { return this.tamanho == 0; }
```

A classe Fila<T> é uma estrutura de dados do tipo fila, que funciona como uma fila de espera na vida real: o primeiro que entra é o primeiro a sair (FIFO - First In, First Out). Ela organiza elementos de forma ordenada usando ponteiros (Ponteiro<T>), com um ponteiro inicio indicando o primeiro elemento e outro fim indicando o último. A fila também mantém um contador tamanho para saber quantos elementos ela tem. Suas funções principais são:

- inserirFim(T elemento): Coloca um novo elemento no final da fila. Se a fila estiver vazia, o elemento vira o primeiro e o último; se não, ele é adicionado depois do último atual, e o ponteiro fim é atualizado. O contador tamanho aumenta em 1.
- removerInicio(): Tira o elemento que está no começo da fila e o retorna. Se a fila estiver vazia, retorna null. Após a remoção, o ponteiro inicio passa a apontar para o próximo elemento, e o tamanho diminui em 1. Se a fila ficar vazia, o ponteiro fim também é ajustado para null.
- getInicio(): Mostra o ponteiro do primeiro elemento da fila, para acessar quem está na frente.
- tamanho(): Informa quantos elementos estão na fila no momento.
- estaVazia(): Verifica se a fila não tem nenhum elemento. Retorna true se o tamanho for 0, e false se houver elementos.

No simulador de elevadores, a Fila é usada para organizar, de forma simples e eficiente, a sequência de passageiros ou chamadas, garantindo que tudo seja processado na ordem certa.

LISTA DEMONSTRAÇÃO:

PARTE 1:

```java
public class Lista<T> {  no usages
    private Ponteiro<T> inicio = null;
    private int tamanho = 0;

    public void inserirFim(T elemento) {
        Ponteiro<T> novo = new Ponteiro(elemento, (Ponteiro)null);
        if (this.inicio == null) {
            this.inicio = novo;
        } else {
            Ponteiro<T> atual;
            for(atual = this.inicio; atual.getProximo() != null; atual = atual.getProximo()) {
            }

            atual.setProximo(novo);
        }

        ++this.tamanho;
    }

    public T obterElemento(int indice) {
        if (indice >= 0 && indice < this.tamanho) {
            Ponteiro<T> atual = this.inicio;

            for(int i = 0; i < indice; ++i) {
                atual = atual.getProximo();
            }
```

LISTA DEMONSTRAÇÃO:

PARTE 2:

```java
                return (T)atual.getElemento();
            } else {
                return null;
            }
        }

        public boolean estaVazia() { return this.tamanho == 0; }

        public int tamanho() { return this.tamanho; }

        public Ponteiro<T> getInicio() { return this.inicio; }
}
```

A classe `Lista<T>` implementa uma estrutura de dados do tipo lista linear, que permite armazenar e gerenciar elementos de forma ordenada e flexível. Ela utiliza ponteiros (`Ponteiro<T>`), com o ponteiro `inicio` indicando o primeiro elemento e um contador `tamanho` para rastrear a quantidade de elementos. Suas funções principais são:

- `inserirFim(T elemento)`: Adiciona um novo elemento ao final da lista. Se a lista estiver vazia, o elemento se torna o primeiro; caso contrário, percorre a lista até o último elemento e conecta o novo elemento, atualizando o ponteiro `proximo` e incrementando o `tamanho`.

- `obterElemento(int indice)`: Retorna o elemento em uma posição específica da lista. Verifica se o índice é válido (entre 0 e `tamanho` - 1); se inválido, retorna `null`. Caso contrário, percorre a lista até o índice desejado e retorna o elemento correspondente.

- `estaVazia()`: Verifica se a lista não contém elementos, retornando `true` se o `tamanho` for 0, e

`false` se houver elementos.

- `tamanho()`: Informa o número total de elementos na lista.

- `getInicio()`: Retorna o ponteiro do primeiro elemento da lista, permitindo acesso ao início da sequência.

No contexto do simulador de elevadores, a `Lista` é útil para armazenar e acessar dinamicamente conjuntos de dados, como a lista de andares ou elevadores, de maneira organizada e eficiente.

2.3. Simulador Elevador

Para realizar a simulação do elevador foram feitas as seguintes funções 1

ELEVADOR DEMONSTRAÇÃO:

PARTE 1:

```java
public class Elevador implements EntidadeSimulavel {  no usages
    private int id;
    private int andarAtual;
    private Fila<Pessoa> pessoasDentro;
    private boolean emMovimento;
    private int andarDestino;

    public Elevador(int id) {
        this.id = id;
        this.andarAtual = 0;
        this.pessoasDentro = new Fila();
        this.emMovimento = false;
        this.andarDestino = 0;
    }

    public void atualizar(int minutoSimulado) {
        if (this.emMovimento) {
            this.andarAtual = this.andarDestino;
            this.emMovimento = false;
            Fila<Pessoa> temp = new Fila();

            for(Ponteiro<Pessoa> p = this.pessoasDentro.getInicio(); p != null; p = p.getProximo()) {
                Pessoa pessoa = (Pessoa)p.getElemento();
                if (pessoa.getAndarDestino() != this.andarAtual) {
                    temp.inserirFim(pessoa);
                } else {
                    System.out.println("[" + minutoSimulado + "] Pessoa " + pessoa.getId() + " desembarcou no andar " + this.andarAtual);
                }
            }
```

ELEVADOR DEMONSTRAÇÃO:

PARTE 2:

```java
            this.pessoasDentro = temp;
        }

    }

    public int getAndarAtual() { return this.andarAtual; }

    public void moverPara(int andarDestino, int minutoSimulado) {
        this.andarDestino = andarDestino;
        this.emMovimento = true;
        System.out.println("[" + minutoSimulado + "] Elevador " + this.id + " movendo-se para o andar " + andarDestino);
    }

    public boolean isEmMovimento() { return this.emMovimento; }

    public boolean podeReceberMaisPessoas() { return this.pessoasDentro.tamanho() < 8; }

    public void embarcarPessoa(Pessoa pessoa, int minutoSimulado) {
        this.pessoasDentro.inserirFim(pessoa);
        System.out.println("[" + minutoSimulado + "] Pessoa " + pessoa.getId() + " embarcou no elevador " + this.id);
    }

    public Fila<Pessoa> getPessoasDentro() { return this.pessoasDentro; }

    public int getId() { return this.id; }
```

Nessa parte do código é demonstrado a Implementação a lógica de movimentação e controle de passageiros.

SIMULADOR DEMONSTRAÇÃO:

PARTE 1:

```
public class Simulador {  no usages
    private SimuladorGUI gui = new SimuladorGUI( simulador: this,  numAndares: 6,  numElevadores: 4);
    private Predio predio = new Predio( simulador: 6,  numAndares: 4);
    private int minutoSimulado = 0;
    private boolean emExecucao = false;

    public void iniciar() {
        if (!this.emExecucao) {
            this.emExecucao = true;
            (new Thread(() -> {
                while(this.emExecucao) {
                    this.atualizar();

                    try {
                        Thread.sleep( millis: 1000L);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

            })).start();
        }

    }

    public void pausar() { this.emExecucao = false; }

    public void continuar() {
        this.minutoSimulado = 0;
        this.predio = new Predio( simulador: 6,  numAndares: 4);
```

SIMULADOR DEMONSTRAÇÃO:

PARTE 2:

```
    public void continuar() {
        this.minutoSimulado = 0;
        this.predio = new Predio( simulador: 6,  numAndares: 4);
        this.emExecucao = false;
        this.gui.atualizarInterface(this.minutoSimulado);
        this.gui.atualizarHorarioPico(false);
        this.iniciar();
    }

    public void atualizar() {
        ++this.minutoSimulado;
        boolean horarioPico = this.isHorarioPico();
        this.gui.atualizarHorarioPico(horarioPico);

        for(int i = 0; i < this.predio.getAndares().tamanho(); ++i) {
            Andar andar = (Andar)this.predio.getAndares().obterElemento(i);
            andar.atualizar(this.minutoSimulado);
        }
```

SIMULADOR DEMONSTRAÇÃO:

PARTE 3:

```
    this.predio.getCentral().atualizar(this.minutoSimulado);
    this.gui.atualizarInterface(this.minutoSimulado);
}

private boolean isHorarioPico() {
    int hora = this.minutoSimulado / 60 % 24;
    return hora >= 7 && hora <= 9 || hora >= 17 && hora <= 19;
}

public Predio getPredio() { return this.predio; }

public int getMinutoSimulado() { return this.minutoSimulado; }

public boolean isEmExecucao() { return this.emExecucao; }
```

A classe `Simulador` é o núcleo do sistema, responsável por gerenciar e coordenar a simulação de elevadores em um edifício. Ela controla o tempo simulado, configurações do prédio e a interação com a interface gráfica. Suas funções principais são:

- `Simulador()`: Inicializa a simulação com valores padrão (6 andares, 3 elevadores, capacidade de 8 pessoas, tempo por andar em pico de 1,5s e fora de pico de 1s, painel com dois botões), criando um objeto `Predio` e a interface `SimuladorGUI`.

- `iniciar()`: Inicia a simulação em uma nova thread, atualizando o estado a cada segundo enquanto `emExecucao` for `true`.

- `pausar()`: Pausa a simulação, definindo `emExecucao` como `false`.

- `continuar()`: Reinicia a simulação, zerando o tempo, o contador de IDs de pessoas e o prédio, e atualiza a interface gráfica antes de iniciar novamente.

- `atualizarConfiguracao(int numAndares, int numElevadores, int capacidade, float tempoPico, float tempoForaPico, int tipoPainel, int heuristica)`: Reconfigura a simulação com novos parâmetros (número de andares, elevadores, capacidade, tempos de movimentação, tipo de painel e heurística), recria o prédio e a interface, e reinicia a simulação.

- `atualizar()`: Avança o tempo simulado (`minutoSimulado`), verifica se é horário de pico, atualiza os andares, a central de controle e a interface gráfica.

- `isHorarioPico()`: Determina se o momento atual é horário de pico (7h-9h ou 17h-19h), com base no `minutoSimulado`.

- `getPredio()`, `getMinutoSimulado()`, `isEmExecucao()`, `getProximoIdPessoa()`, `getTempoPorAndar()`, `getCapacidadeElevador()`, `getTipoPainel()`, `getTempoPorAndarPico()`, `getTempoPorAndarForaPico()`, `getGui()`: Métodos de acesso que retornam o prédio, tempo simulado, estado de execução, próximo ID de pessoa, tempo por andar (ajustado por horário), capacidade do elevador, tipo de painel, tempos de pico e fora de pico, e a interface gráfica, respectivamente.

No simulador de elevadores, a classe `Simulador` é essencial para orquestrar a execução, gerenciar configurações e simular realisticamente o comportamento do sistema, considerando horários de pico e ajustes dinâmicos.

2.4. Central De Controle

Área onde Coordena os elevadores com base nas chamadas pendentes.

CENTRAL DE CONTROLE DEMONSTRAÇÃO:

PARTE 1:

```
public class CentralDeControle implements EntidadeSimulavel {  no usages
    private Lista<Elevador> elevadores = new Lista();
    private Predio predio;
    private int heuristica;
    private int totalPessoasAtendidas;
    private int totalPrioritariasAtendidas;

    public CentralDeControle(int quantidadeElevadores, Predio predio) {
        this.predio = predio;
        this.heuristica = 2;
        this.totalPessoasAtendidas = 0;
        this.totalPrioritariasAtendidas = 0;

        for(int i = 0; i < quantidadeElevadores; ++i) {
            this.elevadores.inserirFim(new Elevador( simulador: i + 1));
        }

    }

    public void atualizar(int minutoSimulado) {
        for(Ponteiro<Elevador> p = this.elevadores.getInicio(); p != null; p = p.getProximo()) {
            Elevador e = (Elevador)p.getElemento();
            e.atualizar(minutoSimulado);
        }

        if (this.heuristica == 2) {
            this.atribuirElevadoresOtimizandoTempo(minutoSimulado);
        }
```

CENTRAL DE CONTROLE DEMONSTRAÇÃO:

PARTE 2:

```
private void atribuirElevadoresOtimizandoTempo(int minutoSimulado) {
    Lista<Andar> andaresComChamadas = new Lista();

    for(Ponteiro<Andar> pa = this.predio.getAndares().getInicio(); pa != null; pa = pa.getProximo()) {
        Andar andar = (Andar)pa.getElemento();
        if (andar.getPainel().isBotaoSubirAtivado() || andar.getPainel().isBotaoDescerAtivado()) {
            andaresComChamadas.inserirFim(andar);
        }
    }

    for(Ponteiro<Elevador> pe = this.elevadores.getInicio(); pe != null; pe = pe.getProximo()) {
        Elevador elevador = (Elevador)pe.getElemento();
        if (!elevador.isEmMovimento() && !andaresComChamadas.estaVazia()) {
            Andar andarMaisPrioritario = null;
            int maxPessoasPrioritarias = -1;
            int menorDistancia = Integer.MAX_VALUE;

            for(Ponteiro<Andar> pc = andaresComChamadas.getInicio(); pc != null; pc = pc.getProximo()) {
                Andar a = (Andar)pc.getElemento();
                int prioritarias = this.contarPessoasPrioritarias(a.getPessoasAguardando());
                int distancia = Math.abs(a.getNumero() - elevador.getAndarAtual());
                if (prioritarias > maxPessoasPrioritarias) {
                    maxPessoasPrioritarias = prioritarias;
                    andarMaisPrioritario = a;
                    menorDistancia = distancia;
                } else if (prioritarias == maxPessoasPrioritarias) {
                    if (distancia < menorDistancia) {
                        andarMaisPrioritario = a;
                        menorDistancia = distancia;
```

CENTRAL DE CONTROLE DEMONSTRAÇÃO:

PARTE 3:

```
                } else if (distancia == menorDistancia && a.getPessoasAguardando().tamanho() > andarMaisPrioritario.getPessoasAguardando().tamanho()) {
                    andarMaisPrioritario = a;
                }
            }
        }

        if (andarMaisPrioritario != null) {
            elevador.moverPara(andarMaisPrioritario.getNumero(), minutoSimulado);

            while(elevador.podeReceberMaisPessoas() && !andarMaisPrioritario.getPessoasAguardando().estaVazia()) {
                Pessoa pessoa = (Pessoa)andarMaisPrioritario.getPessoasAguardando().removerInicio();
                elevador.embarcarPessoa(pessoa, minutoSimulado);
                elevador.moverPara(pessoa.getAndarDestino(), minutoSimulado);
                ++this.totalPessoasAtendidas;
                if (pessoa.temPrioridade()) {
                    ++this.totalPrioritariasAtendidas;
                }
            }

            andarMaisPrioritario.getPainel().resetar();
            Lista<Andar> novaLista = new Lista();

            for(Ponteiro<Andar> temp = andaresComChamadas.getInicio(); temp != null; temp = temp.getProximo()) {
                Andar a = (Andar)temp.getElemento();
                if (a != andarMaisPrioritario) {
                    novaLista.inserirFim(a);
                }
            }
```

CENTRAL DE CONTROLE DEMONSTRAÇÃO:

PARTE 4:

```
private int contarPessoasPrioritarias(Fila<Pessoa> fila) {
    int count = 0;

    for(Ponteiro<Pessoa> p = fila.getInicio(); p != null; p = p.getProximo()) {
        Pessoa pessoa = (Pessoa)p.getElemento();
        if (pessoa.temPrioridade()) {
            ++count;
        }
    }

    return count;
}

public Lista<Elevador> getElevadores() { return this.elevadores; }

public void setHeuristica(int heuristica) { this.heuristica = heuristica; }

public int getHeuristica() { return this.heuristica; }

public int getTotalPessoasAtendidas() { return this.totalPessoasAtendidas; }

public int getTotalPrioritariasAtendidas() { return this.totalPrioritariasAtendidas; }
```

A classe `CentralDeControle` coordena os elevadores de um prédio, implementando a interface `EntidadeSimulavel` para reagir ao tempo simulado. Ela gerencia a alocação de elevadores com base em chamadas, aplicando diferentes heurísticas, e coleta estatísticas como tempo de espera e consumo de energia. Suas funções principais são:

- `CentralDeControle(Simulador simulador, int quantidadeElevadores, Predio predio, int capacidadeElevador)`: Inicializa a central com uma lista de elevadores, associando-a a um prédio e definindo heurística padrão (valor 2).
- `atualizar(int minutoSimulado)`: Atualiza todos os elevadores e aplica a heurística escolhida (ordem de chegada, otimização de tempo ou energia) para atribuir elevadores a chamadas, além de atualizar estatísticas.

- `atribuirElevadoresOrdemChegada(int minutoSimulado)`: Aloca elevadores disponíveis para chamadas na ordem em que os andares são encontrados, registrando tempos de espera e chamadas atendidas.
- `atribuirElevadoresOtimizandoTempo(int minutoSimulado)`: Prioriza andares com mais pessoas prioritárias e menor distância, alocando o elevador mais próximo para minimizar o tempo de espera.
- `atribuirElevadoresOtimizandoEnergia(int minutoSimulado)`: Aloca elevadores com base na menor distância entre o elevador e o andar, reduzindo o consumo de energia.
- `calcularTempoEspera(Andar andar, int minutoSimulado)`: Calcula o tempo de espera de uma pessoa com base no tempo de entrada e o momento atual.
- `calcularConsumoEnergia(int andarAtual, int andarDestino)`: Estima o consumo de energia com base na distância entre andares (0,5 unidades por andar).
- `atualizarEstatisticas(int minutoSimulado)`: Atualiza dados estatísticos, percorrendo os elevadores (ainda não implementado no código fornecido).
- `removerAndar(Lista<Andar> lista, Andar andar)`: Remove um andar de uma lista de chamadas pendentes.
- `contarPessoasPrioritarias(Fila<Pessoa> fila)`: Conta quantas pessoas com prioridade estão em uma fila.
- `getElevadores()`, `getHeuristica()`, `getTotalPessoasAtendidas()`, `getTotalPrioritariasAtendidas()`, `getTempoMedioEspera()`, `getTempoMaximoEspera()`, `getChamadasAtendidas()`, `getConsumoTotalEnergia()`, `getMediaPessoasPorChamada()`: Retornam a lista de elevadores, heurística atual, total de pessoas atendidas (com e sem prioridade), tempos médio e máximo de espera, número de chamadas atendidas, consumo total de energia e média de pessoas por chamada, respectivamente.
- `setHeuristica(int heuristica)`: Define a heurística a ser usada.
- `incrementarPessoasAtendidas(boolean temPrioridade)`: Incrementa o contador de pessoas atendidas, considerando se possuem prioridade.
- `incrementarConsumoEnergia(float consumo)`: Adiciona um valor ao consumo total de energia.

No simulador de elevadores, a `CentralDeControle` é fundamental para gerenciar eficientemente as chamadas, otimizando o atendimento com base em tempo ou energia, enquanto registra métricas importantes.

3. Interface Gráfica

Para visualização e controle do simulador, implementamos uma interface gráfica básica utilizando Swing. A interface mostra o estado dos elevadores e suas ações.

INTERFACE GRÁFICA DEMONSTRAÇÃO:

PARTE 1:

```
public class SimuladorGUI extends JFrame {  no usages
    private Simulador simulador;
    private int numAndares;
    private int numElevadores;
    private JLabel statusLabel;
    private JTextArea logArea;
    private List<List<JLabel>> elevadorLabels;
    private List<JLabel> andarLabels;
    private JTextField quantidadeField;
    private JComboBox<String> andarEntradaCombo;
    private JComboBox<String> andarDestinoCombo;
    private JCheckBox prioridadeCheckBox;
    private JLabel horarioPicoLabel;
    private JLabel tempoLabel;
    private JButton configuracoesButton;
    private JButton estatisticasButton;

    public SimuladorGUI(Simulador simulador, int numAndares, int numElevadores) {
        this.simulador = simulador;
        this.numAndares = numAndares;
        this.numElevadores = numElevadores;
        this.elevadorLabels = new ArrayList();
        this.andarLabels = new ArrayList();
        this.setTitle("Simulador de Elevador Inteligente");
        this.setSize( width: 800,  height: 600);
        this.setDefaultCloseOperation(3);
        this.setLayout(new BorderLayout());
        JPanel topPanel = new JPanel(new FlowLayout());
        this.statusLabel = new JLabel( text: "Simulação parada");
        topPanel.add(this.statusLabel);
```

INTERFACE GRÁFICA DEMONSTRAÇÃO:

PARTE 2:

```
this.horarioPicoLabel = new JLabel( text: "Fora de Pico");
topPanel.add(this.horarioPicoLabel);
this.tempoLabel = new JLabel( text: "Tempo: 00:00:00");
topPanel.add(this.tempoLabel);
this.configuracoesButton = new JButton( text: "Configurações");
this.configuracoesButton.addActionListener(( ActionEvent e) -> this.abrirTelaConfiguracoes());
topPanel.add(this.configuracoesButton);
this.estatisticasButton = new JButton( text: "Estatísticas");
this.estatisticasButton.addActionListener(( ActionEvent e) -> this.abrirTelaEstatisticas());
topPanel.add(this.estatisticasButton);
this.add(topPanel, constraints: "North");
JPanel centerPanel = new JPanel(new BorderLayout());
JPanel adicionarPanel = new JPanel(new FlowLayout());
adicionarPanel.add(new JLabel( text: "Quantidade:"));
this.quantidadeField = new JTextField( text: "1", columns: 5);
adicionarPanel.add(this.quantidadeField);
adicionarPanel.add(new JLabel( text: "Andar de Entrada:"));
String[] andares = new String[numAndares];

for(int i = 0; i < numAndares; ++i) {
    andares[i] = i == 0 ? "Térreo" : String.valueOf(i);
}

this.andarEntradaCombo = new JComboBox(andares);
adicionarPanel.add(this.andarEntradaCombo);
adicionarPanel.add(new JLabel( text: "Andar de Destino:"));
this.andarDestinoCombo = new JComboBox(andares);
```

INTERFACE GRÁFICA DEMONSTRAÇÃO:

PARTE 3:

```
this.andarDestinoCombo = new JComboBox(andares);
adicionarPanel.add(this.andarDestinoCombo);
this.prioridadeCheckBox = new JCheckBox( text: "Prioridade (Idoso/Cadeirante)");
adicionarPanel.add(this.prioridadeCheckBox);
JButton adicionarButton = new JButton( text: "Adicionar");
adicionarButton.addActionListener(( ActionEvent e) -> this.adicionarPessoas());
adicionarPanel.add(adicionarButton);
centerPanel.add(adicionarPanel, constraints: "North");
JPanel elevadoresPanel = new JPanel(new GridLayout( rows: numAndares + 1, cols: numElevadores + 2));

for(int i = 0; i <= numAndares; ++i) {
    if (i == 0) {
        elevadoresPanel.add(new JLabel( text: ""));

        for(int j = 0; j < numElevadores; ++j) {
            elevadoresPanel.add(new JLabel( text: "Elev " + (char)(65 + j), horizontalAlignment: 0));
        }

        elevadoresPanel.add(new JLabel( text: "Fila"));
    } else {
        int andar = numAndares - i;
        JLabel andarLabel = new JLabel(andar == 0 ? "T" : String.valueOf(andar), horizontalAlignment: 0);
        elevadoresPanel.add(andarLabel);
        List<JLabel> labelsAndar = new ArrayList();

        for(int j = 0; j < numElevadores; ++j) {
            JLabel elevLabel = new JLabel( text: "", horizontalAlignment: 0);
            elevLabel.setOpaque(true);
```

INTERFACE GRÁFICA DEMONSTRAÇÃO PARTE 4:

```
this.andarDestinoCombo = new JComboBox(andares);
adicionarPanel.add(this.andarDestinoCombo);
this.prioridadeCheckBox = new JCheckBox( text: "Prioridade (Idoso/Cadeirante)");
adicionarPanel.add(this.prioridadeCheckBox);
JButton adicionarButton = new JButton( text: "Adicionar");
adicionarButton.addActionListener(( ActionEvent e) -> this.adicionarPessoas());
adicionarPanel.add(adicionarButton);
centerPanel.add(adicionarPanel, constraints: "North");
JPanel elevadoresPanel = new JPanel(new GridLayout( rows: numAndares + 1, cols: numElevadores + 2));

for(int i = 0; i <= numAndares; ++i) {
    if (i == 0) {
        elevadoresPanel.add(new JLabel( text: ""));

        for(int j = 0; j < numElevadores; ++j) {
            elevadoresPanel.add(new JLabel( text: "Elev " + (char)(65 + j), horizontalAlignment: 0));
        }

        elevadoresPanel.add(new JLabel( text: "Fila"));
    } else {
        int andar = numAndares - i;
        JLabel andarLabel = new JLabel(andar == 0 ? "T" : String.valueOf(andar), horizontalAlignment: 0);
        elevadoresPanel.add(andarLabel);
        List<JLabel> labelsAndar = new ArrayList();

        for(int j = 0; j < numElevadores; ++j) {
            JLabel elevLabel = new JLabel( text: "", horizontalAlignment: 0);
            elevLabel.setOpaque(true);
```

INTERFACE GRÁFICA DEMONSTRAÇÃO PARTE 5:

```
private void adicionarPessoas() {
    int quantidade = Integer.parseInt(this.quantidadeField.getText());
    int andarEntrada = this.andarEntradaCombo.getSelectedIndex();
    int andarDestino = this.andarDestinoCombo.getSelectedIndex();
    boolean prioridade = this.prioridadeCheckBox.isSelected();
    Andar andar = (Andar)this.simulador.getPredio().getAndares().obterElemento(andarEntrada);
    if (andar == null) {
        JOptionPane.showMessageDialog( parentComponent: this, message: "Erro: Andar inválido selecionado.", title: "Erro", messageTyp
    } else {
        for(int i = 0; i < quantidade; ++i) {
            int id = (int)(Math.random() * (double)10000.0F);
            Pessoa pessoa = new Pessoa(id, andarEntrada, andarDestino, prioridade);
            andar.adicionarPessoa(pessoa);
            if (andarDestino > andarEntrada) {
                andar.getPainel().pressionarSubir();
            } else if (andarDestino < andarEntrada) {
                andar.getPainel().pressionarDescer();
            }
```

INTERFACE GRÁFICA DEMONSTRAÇÃO PARTE 6:

```
private void atualizarCamposPainel() {  2 usages
    boolean isPainelNumerico = simulador.getTipoPainel() == 2;
    andarDestinoCombo.setVisible(!isPainelNumerico);
    andarDestinoNumericoField.setVisible(isPainelNumerico);
}

public void adicionarLog(String mensagem) {  4 usages
    logArea.append(mensagem + "\n");
    logArea.setCaretPosition(logArea.getDocument().getLength());
}

public void limparLog() { logArea.setText(""); }

public void atualizarHorarioPico(boolean horarioPico) {  2 usages
    horarioPicoLabel.setText(horarioPico ? "Horário de Pico" : "Fora de Pico");
}

private void abrirTelaConfiguracoes() { new TelaConfiguracoes(simulador); }

private void abrirTelaEstatisticas() { new TelaEstatisticas(simulador); }
```

Dificuldades encotradas no processo

1. **Implementação Manual de Estruturas de Dados**: A construção de estruturas como Lista<T> e Fila<T> (páginas 2-4 do documento) sem bibliotecas padrão do Java exigiu um design robusto para evitar erros como manipulação incorreta de ponteiros ou vazamentos de memória. A implementação da lógica FIFO na classe Fila e a flexibilidade da Lista para gerenciar andares e elevadores demandaram testes rigorosos para garantir eficiência e corretude.
2. **Coordenação Multi-Elevador**: A classe CentralDeControle (páginas 8-10) enfrentou desafios na implementação das heurísticas de alocação. A heurística de **Otimização de Tempo** exigiu priorizar andares com mais pessoas com necessidades especiais, enquanto a **Otimização de Energia** demandou cálculos precisos de distância (0,5 unidades por andar) para minimizar o consumo energético, tornando a lógica de decisão complexa em cenários com múltiplas chamadas simultâneas.
3. **Sincronização em Tempo Discreto**: A classe Simulador (páginas 6-7) gerencia ciclos temporais, atualizando andares, elevadores e a central de controle a cada segundo. Garantir a sincronização de todos os componentes, especialmente durante horários de pico (7h-9h e 17h-19h), foi desafiador devido à necessidade de ajustar tempos de movimentação (1,5s por andar em pico, 1s fora de pico) e manter a consistência do sistema.
4. **Interface Gráfica em Java Swing**: A implementação da interface gráfica (páginas 12-14) enfrentou dificuldades relacionadas ao desempenho e à usabilidade. Atualizar em tempo real a posição dos elevadores, filas de espera e estatísticas (como tempo médio de espera e consumo de energia) exigiu otimizações para evitar lentidão, especialmente em prédios com muitos andares ou elevadores.
5. **Validação e Testes das Heurísticas**: Validar as três heurísticas (Ordem de Chegada, Otimização de Tempo e Otimização de Energia) foi um desafio, pois cada uma impacta métricas como tempo de espera, consumo de energia e atendimento de prioridades de forma diferente. Testes extensivos foram necessários para garantir que as heurísticas funcionassem corretamente em cenários variados, como alta demanda ou chamadas prioritárias frequentes.

Conclusão

O Simulador de Elevadores Inteligentes foi implementado com êxito, consolidando-se como uma ferramenta robusta para o estudo de conceitos avançados de engenharia de software, incluindo programação orientada a objetos, estruturas de dados personalizadas e simulação em tempo discreto. As três heurísticas implementadas — Ordem de Chegada, Otimização de Tempo e Otimização de Energia — permitiram uma análise comparativa detalhada dos trade-offs entre eficiência operacional, acessibilidade e sustentabilidade energética. A heurística de Otimização de Tempo destacou-se por priorizar usuários com necessidades especiais, enquanto a Otimização de Energia demonstrou potencial para reduzir o consumo em cenários de baixa demanda. A interface gráfica, desenvolvida em Java Swing, proporcionou uma visualização intuitiva e interativa, facilitando a análise de métricas como tempo médio de espera, consumo energético e fluxo de passageiros. Apesar dos desafios enfrentados, como a complexidade da implementação manual de estruturas de dados e a sincronização em tempo real, o projeto alcançou seus objetivos educacionais, reforçando o aprendizado prático em algoritmos e estruturas de dados. A modularidade do sistema permite futuras expansões, como a inclusão de limites de peso, algoritmos de aprendizado de máquina para otimização dinâmica ou integração com sensores simulados. Limitações, como o desempenho da interface em cenários de alta complexidade, sugerem oportunidades para otimizações futuras, como o uso de frameworks gráficos mais eficientes ou paralelização de processos. Este trabalho não apenas contribui para o ensino de conceitos técnicos, mas também inspira reflexões sobre a aplicação de tecnologia em problemas do mundo real, como mobilidade urbana e acessibilidade