SIMULADOR DE ELEVADORES INTELIGENTES: MODELAGEM, ESTRUTURAS DE DADOS E ALGORITMOS

RESUMO

Este documento apresenta uma descrição técnica detalhada do sistema de simulação de elevadores inteligentes desenvolvido em Java. São abordados os aspectos de modelagem orientada a objetos, as estruturas de dados personalizadas implementadas e os algoritmos de controle dos elevadores. O sistema utiliza heurísticas para otimização de parâmetros como tempo de espera e consumo de energia, simulando cenários reais de operação de elevadores em prédios. Os resultados demonstram a eficácia das diferentes estratégias de controle e seu impacto nos indicadores de desempenho.

1. INTRODUÇÃO

A gestão eficiente de sistemas de elevadores representa um desafio significativo em edificações modernas, especialmente em prédios de maior porte. O problema envolve diversos fatores como tempo de espera dos usuários, consumo energético, capacidade dos elevadores e padrões de tráfego variáveis ao longo do dia.

O simulador de elevadores inteligentes foi desenvolvido com o objetivo de permitir a modelagem e análise do comportamento desses sistemas sob diferentes configurações e estratégias de controle. O software possibilita a avaliação de parâmetros como:

- Número de andares do prédio
- Quantidade de elevadores
- Tipos de painéis de controle
- Modelos heurísticos para tomada de decisão
- Capacidade dos elevadores
- Tempo de deslocamento entre andares
- Consumo energético

Este documento técnico apresenta em detalhes a modelagem do sistema, as estruturas de dados personalizadas e os algoritmos implementados, com foco especial nas diferentes heurísticas de otimização aplicadas ao controle inteligente dos elevadores.

2. MODELAGEM DO SISTEMA

O sistema foi desenvolvido seguindo o paradigma de orientação a objetos, com uma arquitetura que separa claramente os diferentes componentes e responsabilidades. A modelagem baseia-se em uma

representação fiel do mundo real, com classes representando prédios, andares, elevadores, pessoas e sistemas de controle.

2.1 Diagrama de Classes

A estrutura do sistema é composta por classes que representam:

1. **Entidades físicas**: Predio, Andar, Elevador, Pessoa
2. **Elementos de controle**: PainelChamadas, PainelElevador, CentralDeControle
3. **Estruturas de simulação**: Simulador, RelatorioSimulacao
4. **Interface gráfica**: ConfiguracaoSimuladorGUI, SimuladorGUI
5. **Estruturas de dados**: Lista, Fila, FilaPrioridade

As classes são organizadas em pacotes que representam suas responsabilidades específicas:

- com.alvaro.simulador.modelagem: Classes que representam entidades físicas
- com.alvaro.simulador.controle: Classes responsáveis pela lógica de controle
- com.alvaro.simulador.tads: Estruturas de dados personalizadas
- com.alvaro.simulador.enums: Tipos enumerados utilizados no sistema
- com.alvaro.simulador.graficos: Interface gráfica do simulador
- com.alvaro.simulador.relatorioSimulacao: Geração de relatórios

2.2 Principais Classes e Componentes

2.2.1 Entidades Físicas

Predio

- Contém os andares e a central de controle
- Gerencia a distribuição de chamadas de elevadores
- Coordena a atualização de todas as entidades simuláveis

Andar

- Armazena as pessoas que estão esperando
- Contém o painel de chamadas
- Gerencia embarque e desembarque de pessoas

Elevador

- Controla sua própria movimentação entre andares
- Gerencia o embarque e desembarque de passageiros
- Contém o painel interno com os botões de andares
- Monitora consumo de energia

Pessoa

- Representa usuários com diferentes características
- Armazena informações de origem, destino e tipo (normal, idoso, cadeirante)
- Monitora tempo de espera e deslocamento

2.2.2 Elementos de Controle

PainelChamadas

- Implementa diferentes tipos de painéis (botão único, dois botões, painel numérico)
- Registra e reseta chamadas de elevadores
- Permite verificar se existem chamadas pendentes

PainelElevador

- Gerencia os botões internos do elevador
- Oferece métodos para determinar próximos andares a serem atendidos
- Permite verificar quais botões estão pressionados

CentralDeControle

- Implementa algoritmos de distribuição de elevadores
- Aplica diferentes heurísticas de otimização
- Gerencia o atendimento de chamadas pendentes
- Coordena a transferência de pessoas para os elevadores

2.2.3 Estruturas de Simulação

Simulador

- Controla o ciclo de simulação
- Gerencia o tempo simulado
- Permite configuração dos parâmetros de simulação

RelatorioSimulacao

- Coleta estatísticas sobre a simulação
- Gera relatórios de desempenho do sistema
- Calcula métricas como tempo médio de espera e consumo de energia

2.3 Enumeradores Utilizados

O sistema utiliza os seguintes enumeradores para representar estados e configurações:

Direcao

- SUBINDO: Representa movimento ascendente
- DESCENDO: Representa movimento descendente
- PARADO: Representa ausência de movimento

TipoPainel

- UNICO_BOTAO: Painel com um único botão de chamada
- DOIS_BOTOES: Painel com botões para subir e descer

- PAINEL_NUMERICO: Painel com botões para todos os andares

ModeloHeuristica

- SEM_HEURISTICA: Atendimento por ordem de chegada
- OTIMIZACAO_TEMPO_ESPERA: Prioriza minimização do tempo de espera
- OTIMIZACAO_CONSUMO_ENERGIA: Prioriza redução do consumo energético

3. ESTRUTURAS DE DADOS

O simulador implementa suas próprias estruturas de dados genéricas em vez de utilizar as coleções padrão da API Java. Isso foi feito para demonstrar conceitos fundamentais de estruturas de dados e proporcionar maior controle sobre a implementação.

3.1 Lista

A classe Lista<T> implementa uma lista encadeada simples com os seguintes recursos:

```
public class Lista<T> {
  private No<T> inicio;
  private No<T> fim;
  private int tamanho;

  private static class No<T> {
    private T dado;
    private No<T> proximo;

    public No(T dado) {
      this.dado = dado;
      this.proximo = null;
    }
  }

  // Métodos principais: adicionar, remover, obter, tamanho, vazia
}
```

A estrutura oferece operações essenciais como:

- adicionar(T elemento): Adiciona um elemento ao final da lista
- adicionar(T elemento, int posicao): Adiciona um elemento em posição específica
- remover(int posicao): Remove o elemento na posição especificada
- remover(T elemento): Remove a primeira ocorrência do elemento
- obter(int posicao): Recupera o elemento na posição especificada
- contem(T elemento): Verifica se a lista contém o elemento
- tamanho(): Retorna o número de elementos na lista

- vazia(): Verifica se a lista está vazia
- limpar(): Remove todos os elementos da lista

A implementação utiliza ponteiros para o início e fim da lista, otimizando operações de adição ao final.

3.2 Fila

A classe Fila<T> implementa uma fila (FIFO - First-In-First-Out) utilizando encadeamento de nós:

```
public class Fila<T> {
  private No<T> inicio;
  private No<T> fim;
  private int tamanho;

  private static class No<T> {
    private T dado;
    private No<T> proximo;

    public No(T dado) {
      this.dado = dado;
      this.proximo = null;
    }
  }

  // Métodos principais: enfileirar, desenfileirar, primeiro, tamanho, vazia
}
```

Suas operações principais incluem:

- enfileirar(T elemento): Adiciona um elemento ao final da fila
- desenfileirar(): Remove e retorna o elemento do início da fila
- primeiro(): Retorna o elemento do início sem removê-lo
- vazia(): Verifica se a fila está vazia
- tamanho(): Retorna o número de elementos na fila
- limpar(): Remove todos os elementos da fila
- paraLista(): Converte a fila para uma lista encadeada

3.3 Fila de Prioridade

A classe FilaPrioridade<T> implementa uma fila com prioridade, onde elementos com maior valor de prioridade são atendidos primeiro:

```
public class FilaPrioridade<T> {
  private No<T> inicio;
  private int tamanho;

  private static class No<T> {
    private T dado;
    private int prioridade;
    private No<T> proximo;

    public No(T dado, int prioridade) {
      this.dado = dado;
```

```
        this.prioridade = prioridade;
        this.proximo = null;
    }
}

// Métodos principais: enfileirar, desenfileirar, primeiro, tamanho, vazia
}
```

A fila de prioridade oferece funcionalidades como:

- enfileirar(T elemento, int prioridade): Adiciona elemento com prioridade específica
- desenfileirar(): Remove e retorna o elemento de maior prioridade
- primeiro(): Retorna o elemento de maior prioridade sem removê-lo
- vazia(): Verifica se a fila está vazia
- tamanho(): Retorna o número de elementos na fila
- paraLista(): Converte a fila para uma lista encadeada

Esta estrutura é utilizada principalmente para gerenciar a ordem de atendimento de pessoas nos andares, priorizando pessoas com necessidades especiais (idosos e cadeirantes).

3.4 Implementação de Requisitos Específicos

As estruturas de dados foram estendidas com funcionalidades específicas para o contexto da simulação:

- **Priorização de pessoas**: A FilaPrioridade é utilizada para garantir que pessoas com mobilidade reduzida (idosos e cadeirantes) tenham preferência no embarque.
- **Transferência entre estruturas**: As estruturas implementam métodos como paraLista() para facilitar a conversão entre diferentes representações.
- **Manipulação eficiente**: As operações são implementadas considerando a complexidade computacional, com ponteiros para início e fim das estruturas para otimizar inserções e remoções.

4. ALGORITMOS DE CONTROLE

O coração do simulador está nos algoritmos de controle implementados na classe CentralDeControle, que determinam como os elevadores são designados para atender às chamadas dos usuários.

4.1 Sistema de Decisão e Heurísticas

O sistema implementa três modelos heurísticos distintos, cada um com uma estratégia específica:

4.1.1 Sem Heurística (Ordem de Chegada)

Este algoritmo implementa uma estratégia simples, baseada na ordem de chegada das chamadas:

```
private void distribuirElevadoresSemHeuristica() {
  // Se não há andares pendentes, não faz nada
  if (andaresPendentes.tamanho() == 0) {
    return;
  }

  // Lista de elevadores disponíveis
  Lista<Elevador> elevadoresDisponiveis = obterElevadoresDisponiveis();

  // Se não há elevadores disponíveis, não faz nada
  if (elevadoresDisponiveis.tamanho() == 0) {
    return;
  }

  // Agora distribui os elevadores disponíveis entre os andares pendentes
  int elevadorAtual = 0;
  Lista<Integer> andaresPendentesTemp = copiarListaAndaresPendentes();

  // Limpa a lista original para reconstruir
  andaresPendentes = new Lista<>();

  // Distribui elevadores para andares, por ordem de chegada
  while (elevadorAtual < elevadoresDisponiveis.tamanho() && andaresPendentesTemp.tamanho() > 0) {
    // Obtém o próximo andar da lista (o mais antigo que chegou)
    int andarAtual = andaresPendentesTemp.obter(0);
    andaresPendentesTemp.remover(0);

    // Obtém o elevador atual
    Elevador elevador = elevadoresDisponiveis.obter(elevadorAtual);

    // Designa o elevador para este andar
    boolean sucesso = elevador.definirDestinoExterno(andarAtual);
```

```
        if (sucesso) {
            // Registra a designação na matriz de controle
            registrarElevadorParaAndarDirecao(andarAtual, Direcao.PARADO, elevador.getId());
        } else {
            // Se não conseguiu definir o destino, coloca o andar de volta na lista
            andaresPendentes.adicionar(andarAtual);
        }

        // Passa para o próximo elevador
        elevadorAtual++;
    }

    // Se sobrou algum andar pendente, adiciona de volta à lista
    for (int i = 0; i < andaresPendentesTemp.tamanho(); i++) {
        andaresPendentes.adicionar(andaresPendentesTemp.obter(i));
    }
}
```

Características principais:

- Atende as chamadas na sequência em que foram realizadas
- Não considera fatores como tempo de espera ou consumo energético
- Implementação simples e previsível

4.1.2 Otimização de Tempo de Espera

Esta heurística prioriza a minimização do tempo de espera das pessoas, especialmente aquelas que estão aguardando há mais tempo:

```
private void distribuirElevadoresOtimizandoTempoEspera() {
    if (andaresPendentes.tamanho() == 0) {
        return;
    }

    Lista<Elevador> elevadoresDisponiveis = obterElevadoresDisponiveis();

    if (elevadoresDisponiveis.tamanho() == 0) {
        return;
    }
```

```
  // Ordenar andares por tempo de espera (maior para menor)
  Lista<Integer> andaresPorPrioridade = new Lista<>();
  Lista<Integer> tempAndaresPendentes = copiarListaAndaresPendentes();

  // Enquanto houver andares pendentes para ordenar
  while (tempAndaresPendentes.tamanho() > 0) {
    int andarMaiorEspera = -1;
    int maiorTempoEspera = -1;

    // Encontrar o andar com maior tempo de espera
    for (int i = 0; i < tempAndaresPendentes.tamanho(); i++) {
      int andar = tempAndaresPendentes.obter(i);
      int tempoEsperaTotal = temposEsperaAndares[andar][0] + temposEsperaAndares[andar][1];

      if (tempoEsperaTotal > maiorTempoEspera) {
        maiorTempoEspera = tempoEsperaTotal;
        andarMaiorEspera = andar;
      }
    }

    // Adicionar à lista ordenada e remover da temporária
    if (andarMaiorEspera != -1) {
      andaresPorPrioridade.adicionar(andarMaiorEspera);
      // Remove o andar da lista temporária
      // ...
    }
  }

  // Distribui elevadores para os andares priorizados
  for (int i = 0; i < andaresPorPrioridade.tamanho() && i < elevadoresDisponiveis.tamanho(); i++) {
    int andar = andaresPorPrioridade.obter(i);
    Elevador elevador = elevadoresDisponiveis.obter(i);

    // Tenta designar o elevador para o andar
    boolean sucesso = elevador.definirDestinoExterno(andar);
```

```
    if (sucesso) {

registrarElevadorParaAndarDirecao(andar, Direcao.PARADO, elevador.getId());
        // Resetar o tempo de espera para este andar

temposEsperaAndares[andar][0] = 0;

temposEsperaAndares[andar][1] = 0;
    } else {

andaresPendentes.adicionar(andar);
    }
  }
}
```

Características principais:

- Prioriza andares com pessoas esperando há mais tempo
- Mantém um registro de tempo de espera para cada andar
- Ordena chamadas por prioridade temporal
- Adequada para melhorar a experiência dos usuários

4.1.3 Otimização de Consumo Energético

Esta heurística busca minimizar o consumo de energia, priorizando deslocamentos mais curtos e evitando movimentação desnecessária:

```
private void distribuirElevadoresOtimizandoEnergia() {
  if (andaresPendentes.tamanho() == 0) {
    return;
  }

  Lista<Elevador> elevadoresDisponiveis = obterElevadoresDisponiveis();

  if (elevadoresDisponiveis.tamanho() == 0) {
    return;
  }

  // Filtramos os andares pendentes que já têm pessoas esperando
  Lista<Integer> andaresCriticos = new Lista<>();
  // Identificação de andares críticos...

  // Matriz de pontuações energéticas para cada par elevador-andar
  int[][] pontuacoesEnergeticas = new int[elevadoresDisponiveis.tamanho()][andaresPendentes.tamanho()];
```

```
  // Calcular pontuação energética (menor é melhor)
  for (int e = 0; e < elevadoresDisponiveis.tamanho(); e++) {
    Elevador elevador = elevadoresDisponiveis.obter(e);
    int andarAtualElevador = elevador.getAndarAtual();

    for (int a = 0; a < andaresPendentes.tamanho(); a++) {
      int andarDestino = andaresPendentes.obter(a);

      // Calcular custo energético baseado na distância
      int distancia = Math.abs(andarAtualElevador - andarDestino);

      // Penalidade para grandes deslocamentos
      int pontuacao = distancia * 10;

      // Verificar se o andar é crítico (pessoas esperando)
      boolean andarEhCritico = false;
      // Verificação de criticidade...

      // Se o andar for crítico, reduzir a pontuação para priorizá-lo
      if (andarEhCritico) {
        pontuacao -= 50;
      }

      // Se o elevador estiver vazio, penalizar ainda mais os deslocamentos longos
      if (elevador.getNumPassageiros() == 0) {
        pontuacao += distancia * 5;
      }

      // Se o elevador estiver já movendo na direção do andar, reduzir pontuação
      if ((elevador.getDirecao() == Direcao.SUBINDO && andarDestino > andarAtualElevador) ||
          (elevador.getDirecao() == Direcao.DESCENDO && andarDestino < andarAtualElevador)) {
        pontuacao -= 30;
      }

      pontuacoesEnergeticas[e][a] = pontuacao;
    }
  }
```

```
    // Encontrar os melhores pares elevador-andar (menor pontuação = melhor)
    // e atribuir elevadores às chamadas correspondentes
    // ...
}
```

Características principais:

- Prioriza deslocamentos curtos para reduzir consumo energético
- Considera a ocupação dos elevadores (elevadores vazios têm custos diferentes)
- Favorece reutilização de elevadores já em movimento na direção desejada
- Balanceia eficiência energética e atendimento a casos críticos (espera prolongada)

4.2 Embarque e Desembarque de Passageiros

Os algoritmos de transferência de pessoas para os elevadores também são influenciados pelas heurísticas selecionadas:

4.2.1 Transferência por Tempo de Espera

```
private void transferirPessoasPorTempoEspera(Andar andar, Elevador elevador,
                                             Lista<Pessoa> pessoasEsperando,
                                             Lista<Pessoa> pessoasRemovidas,
                                             int numeroAndar, Direcao direcaoElevador) {
    // Primeiro, identificar pessoas com prioridade especial (idosos, cadeirantes)
    for (int i = 0; i < pessoasEsperando.tamanho(); i++) {
        Pessoa pessoa = pessoasEsperando.obter(i);
        Direcao direcaoPessoa = Direcao.obterDirecaoPara(numeroAndar, pessoa.getAndarDestino());

        // Verificar se pode embarcar
        boolean podeEmbarcar = podePessoaEmbarcar(numeroAndar, direcaoElevador, direcaoPessoa);

        // Verifica se tem prioridade (idoso ou cadeirante)
        boolean temPrioridade = pessoa.isCadeirante() || pessoa.isIdoso();

        if (podeEmbarcar && temPrioridade && elevador.getNumPassageiros() < elevador.getCapacidadeMaxima()) {
            if (elevador.embarcarPessoa(pessoa)) {
```

```
pessoasRemovidas.adicionar(pessoa
);
        }
      }
    }


    // Depois, ordenar outras pessoas
por tempo de espera
    while
(pessoasEsperando.tamanho() > 0 &&
elevador.getNumPassageiros() <
elevador.getCapacidadeMaxima()) {
      Pessoa pessoaMaiorEspera =
null;
      int maiorTempoEspera = -1;


      // Encontrar pessoa com maior
tempo de espera
      for (int i = 0; i <
pessoasEsperando.tamanho(); i++) {
        Pessoa pessoa =
pessoasEsperando.obter(i);
        Direcao direcaoPessoa =
Direcao.obterDirecaoPara(numeroAn
dar, pessoa.getAndarDestino());


        boolean podeEmbarcar =
podePessoaEmbarcar(numeroAndar,
direcaoElevador, direcaoPessoa);
        boolean jaRemovida =
estaNaLista(pessoasRemovidas,
pessoa.getId());
        if (podeEmbarcar &&
!jaRemovida &&
pessoa.getTempoEspera() >
maiorTempoEspera) {
          maiorTempoEspera =
pessoa.getTempoEspera();
          pessoaMaiorEspera = pessoa;
        }
      }


      // Se encontrou alguém para
embarcar
      if (pessoaMaiorEspera != null) {
        if
(elevador.embarcarPessoa(pessoaMa
iorEspera)) {

pessoasRemovidas.adicionar(pessoa
MaiorEspera);
        }
      } else {
        break; // Ninguém mais pode
embarcar
      }
    }
}
```

4.2.2 Transferência por Otimização Energética

```
private void
transferirPessoasPorDirecao(Andar
andar, Elevador elevador,
```

```
                    Lista<Pessoa> pessoasEsperando,
                    Lista<Pessoa> pessoasRemovidas,
                    int numeroAndar, Direcao direcaoElevador) {
  // Verificar se o elevador já tem passageiros
  if (elevador.getNumPassageiros() > 0) {
    // Determinar a direção predominante dos passageiros atuais
    Direcao direcaoPredominante = determinarDirecaoPredominante(elevador);

    // Priorizar pessoas indo na mesma direção que a maioria
    for (int i = 0; i < pessoasEsperando.tamanho(); i++) {
      Pessoa pessoa = pessoasEsperando.obter(i);
      Direcao direcaoPessoa = Direcao.obterDirecaoPara(numeroAndar, pessoa.getAndarDestino());

      boolean podeEmbarcar = podePessoaEmbarcar(numeroAndar, direcaoElevador, direcaoPessoa);
      boolean mesmaDir = direcaoPessoa == direcaoPredominante;
      if (podeEmbarcar && mesmaDir && elevador.getNumPassageiros() < elevador.getCapacidadeMaxima()) {
        if (elevador.embarcarPessoa(pessoa)) {
          pessoasRemovidas.adicionar(pessoa);
        }
      }
    }
  } else {
    // Se o elevador está vazio, tentar agrupar pessoas por destino
    // Isto minimiza paradas, economizando energia
    int[] pessoasPorAndar = new int[numeroTotalAndares];

    // Mapear pessoas por andar de destino
    // ...

    // Encontrar andar com mais pessoas
    int andarComMaisPessoas = -1;
    int maxPessoas = 0;
    // ...

    // Priorizar pessoas indo para o andar mais popular
```

```
    // ...
  }
}
```

4.3 Lógica de Movimentação dos Elevadores

O algoritmo de movimentação dos elevadores foi implementado na classe Elevador e segue o seguinte fluxo:

```
public void atualizar(int minutoSimulado) {
  // PARTE 1: MOVIMENTAÇÃO
  if (emMovimento) {
    // Decrementar o tempo para o próximo andar
    tempoParaProximoAndar--;

    // Quando o tempo de deslocamento terminar
    if (tempoParaProximoAndar <= 0) {
      // Atualizar a posição do elevador
      andarAtual += (direcao == Direcao.SUBINDO) ? 1 : -1;

      // Verificar se chegou ao destino
      if (andarAtual == andarDestino) {
        // Parou no andar de destino
        emMovimento = false;
        pararNoAndar();
      } else {
        // Continua em movimento para o próximo andar
        // Reiniciar o temporizador para o próximo trecho
        boolean horarioPico = false;
        tempoParaProximoAndar = horarioPico ? tempoDeslocamentoPico : tempoDeslocamentoPadrao;
      }
    }
  }
  // PARTE 2: DECISÃO (apenas se não estiver em movimento)
  else {
    // Se tem um destino e não está no destino, começa a se mover
    if (andarDestino != -1 && andarAtual != andarDestino) {
      iniciarMovimento();
    }
    // Se não tem destino e não há botões pressionados no painel, procura novo destino
    else if (andarDestino == -1 && !painelInterno.temBotaoPressionado()) {
      determinarProximoDestino();
    }
  }
```

```
}
```

Quando um elevador para em um andar, ele executa uma série de ações:

```
private void pararNoAndar() {
  // Registrar consumo de energia por parada
  consumoEnergiaTotal += consumoEnergiaPorParada;

  // Liberar passageiros que chegaram ao destino
  liberarPassageiros();

  // Resetar o botão deste andar no painel
painelInterno.resetarBotao(andarAtual);

  // Determinar o próximo destino
  int proximoAndarInterno = painelInterno.proximoAndarSelecionado(andarAtual, direcao);

  if (proximoAndarInterno != -1) {
    // Ainda há andares para atender na mesma direção
    andarDestino = proximoAndarInterno;
  } else {
    // Verificar na direção oposta
    Direcao direcaoOposta = direcao.oposto();
    proximoAndarInterno = painelInterno.proximoAndarSelecionado(andarAtual, direcaoOposta);

    if (proximoAndarInterno != -1) {
      // Há andares na direção oposta
      direcao = direcaoOposta;
      andarDestino = proximoAndarInterno;
    } else {
      // Não há mais destinos - ficar parado
      direcao = Direcao.PARADO;
      andarDestino = -1;
    }
  }
}
```

5. INTERFACE GRÁFICA

O simulador possui uma interface gráfica interativa desenvolvida com Swing que permite:

1. **Configuração inicial da simulação**:
    - Número de andares e elevadores
    - Tipo de painel de controle
    - Modelo heurístico

    - Parâmetros de tempo e energia
    - Limite de tempo da simulação
2. **Visualização em tempo real**:
    - Representação visual do prédio e elevadores
    - Movimento dos elevadores entre andares
    - Pessoas aguardando em cada andar
    - Estatísticas de tempo e energia
3. **Controle da simulação**:
    - Iniciar, pausar e encerrar a simulação
    - Configurar velocidade de execução
    - Gerar pessoas aleatoriamente
    - Executar ciclos específicos
4. **Geração de relatórios**:
    - Estatísticas detalhadas de desempenho
    - Métricas de tempo de espera e deslocamento
    - Dados de consumo energético
    - Utilização média dos elevadores

6. ANÁLISE DE DESEMPENHO

6.1 Indicadores de Desempenho

O sistema monitora diversos indicadores para avaliar o desempenho da simulação:

- **Tempo médio de espera**: Tempo médio que os usuários aguardam pelo elevador
- **Tempo médio de deslocamento**: Tempo médio que os usuários passam dentro do elevador
- **Tempo máximo de espera**: Maior tempo de espera registrado durante a simulação
- **Consumo energético total**: Soma do consumo de todos os elevadores
- **Taxa de utilização dos elevadores**: Porcentagem de ocupação dos elevadores
- **Número de pessoas no sistema**: Total de pessoas presentes na simulação
- **Consumo energético por pessoa transportada**: Eficiência energética do sistema

6.2 Comparação Entre Heurísticas

As três heurísticas implementadas apresentam comportamentos distintos:

Sem Heurística (FIFO):

- Previsível e estável
- Não otimiza nenhum parâmetro específico
- Desempenho mediocre em cenários de alta demanda
- Menor complexidade computacional

Otimização de Tempo de Espera:

- Menor tempo médio de espera entre as três heurísticas
- Maior consumo energético devido à movimentação mais frequente dos elevadores
- Melhor experiência para os usuários
- Prioriza eficientemente usuários esperando há mais tempo

Otimização de Consumo Energético:

- Menor consumo energético total
- Tempo de espera mais alto em comparação às outras heurísticas
- Maior eficiência energética (consumo por pessoa transportada)
- Agrupamento eficiente de pessoas com destinos similares

6.3 Análise de Complexidade Algorítmica

Algoritmo Sem Heurística:

- Complexidade temporal: O(m + n), onde m é o número de andares pendentes e n é o número de elevadores disponíveis
- Estrutura simples de fila FIFO

Algoritmo de Otimização de Tempo de Espera:

- Complexidade temporal: $O(m^2 + n)$, onde m é o número de andares pendentes
- Ordenação de andares por tempo de espera: $O(m^2)$
- Atribuição de elevadores: O(n)

Algoritmo de Otimização de Consumo Energético:

- Complexidade temporal: $O(m \times n + m \times n \log(m \times n))$
- Cálculo de pontuações energéticas: $O(m \times n)$
- Seleção dos melhores pares elevador-andar: $O(m \times n \log(m \times n))$

7. CONSIDERAÇÕES DE IMPLEMENTAÇÃO

7.1 Design Patterns Aplicados

O sistema utiliza diversos padrões de projeto que facilitam sua manutenção e extensão:

1. **Observer Pattern**: A atualização das entidades é realizada de forma hierárquica, com o simulador notificando o prédio, que por sua vez notifica andares,

elevadores e outros componentes.

2. **Strategy Pattern**: As diferentes heurísticas de controle são implementadas como estratégias intercambiáveis, permitindo a seleção dinâmica do algoritmo.
3. **Composite Pattern**: A estrutura do prédio forma uma hierarquia de objetos (prédio, andares, elevadores, pessoas) que podem ser tratados de maneira uniforme.
4. **Command Pattern**: As ações de controle da simulação (iniciar, pausar, executar ciclo) são encapsuladas como comandos.
5. **Factory Method**: A criação de componentes como elevadores e pessoas segue o padrão de método fábrica.

7.2 Extensibilidade do Sistema

O sistema foi projetado considerando a extensibilidade em várias dimensões:

1. **Novas Heurísticas**: A arquitetura permite a inclusão de novos modelos heurísticos, bastando implementar as interfaces correspondentes.
2. **Tipos de Painel**: O sistema suporta diferentes configurações de painéis, podendo ser estendido para incluir sistemas mais complexos.
3. **Parâmetros de Simulação**: A configuração dos parâmetros de simulação é flexível, permitindo a experimentação com diferentes cenários.
4. **Integração com Modelos Externos**: O sistema pode ser estendido para integrar-se com modelos estatísticos ou de aprendizado de máquina para otimização preditiva.

7.3 Limitações Atuais

1. **Modelos de Tráfego**: O simulador atual não implementa modelos sofisticados de padrões de tráfego (horários de pico, padrões específicos por tipo de prédio).
2. **Otimização Multi-objetivo**: As heurísticas atuais focam em objetivos únicos (tempo ou energia), sem implementar balanceamento adaptativo.
3. **Tratamento de Falhas**: O sistema não simula cenários de falha de elevadores ou comportamentos emergenciais.
4. **Escalonamento**: Algumas partes do código podem apresentar desafios de escalonamento para prédios com muitos andares ou elevadores.

9. CONCLUSÕES

O simulador de elevadores inteligentes apresenta uma plataforma versátil para análise e avaliação de diferentes estratégias de controle de sistemas de elevadores. A implementação de estruturas de dados personalizadas e algoritmos heurísticos específicos permite uma compreensão profunda dos desafios e soluções neste domínio.

As principais conclusões obtidas são:

1. A escolha da estratégia de controle tem impacto significativo no desempenho global do sistema, com claros trade-offs entre tempo de espera e consumo energético.
2. A priorização de pessoas com necessidades especiais (idosos e cadeirantes) é eficientemente implementada através da estrutura de fila de prioridade.
3. A heurística de otimização de tempo de espera é mais adequada para prédios comerciais, onde a experiência do usuário é primordial.
4. A heurística de otimização energética é mais adequada para prédios residenciais ou em cenários onde a sustentabilidade é um fator crítico.
5. A arquitetura modular do sistema permite a implementação e experimentação com novas estratégias de controle e configurações de prédio.