Universidade do Estado de Santa Catarina
Centro de Ensino Superior Alto Vale do Itajaí – CEAVI

Desenvolvimento de Sistemas Paralelos e Distribuídos - 65DSD

Threads

Desenvolvimento de um Sistema Paralelo utilizando Threads

Erick Augusto Warmling
Marco Antonio Garlini Possamai

Professor: Fernando dos Santos

ESO Engenharia de Software

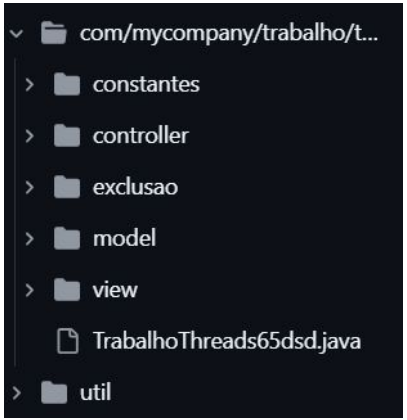
UDESC UNIVERSIDADE DO ESTADO DE SANTA CATARINA

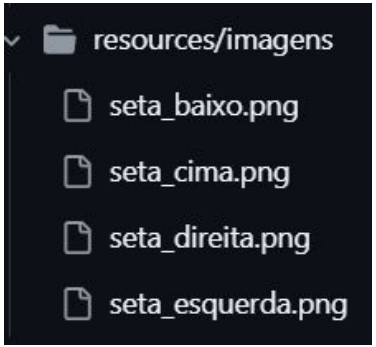
Estrutura do Projeto

- **Constantes:** Define valores fixos utilizados em todo o projeto, como tipos de células da malha viária;
- **Controller:** Contém a lógica de controle da simulação, gerenciando ciclo de vida dos carros e coordenação geral;
- **Exclusao:** Implementa mecanismos de exclusão mútua (Semáforo e Monitor) para sincronização entre threads;
- **Modelo:** Classes que representam as entidades do domínio: carros, células e malha viária;
- **Util:** Utilitários auxiliares: leitura de arquivos e gerenciamento de ícones;
- **View:** Interface gráfica (Swing): telas de configuração, simulação e painel de visualização;

Estrutura do Projeto

Além dessa estrutura apresentada anteriormente, foi colocado na raiz do projeto as malhas disponibilizadas para o desenvolvimento desse trabalho, e os ícones para melhor visualização da malha.





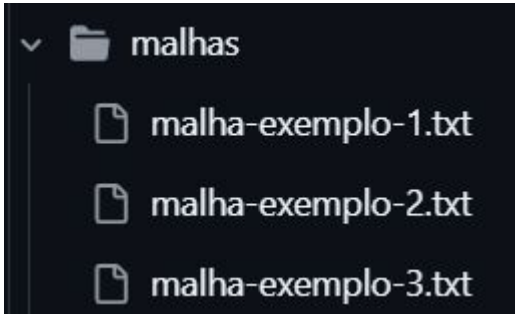
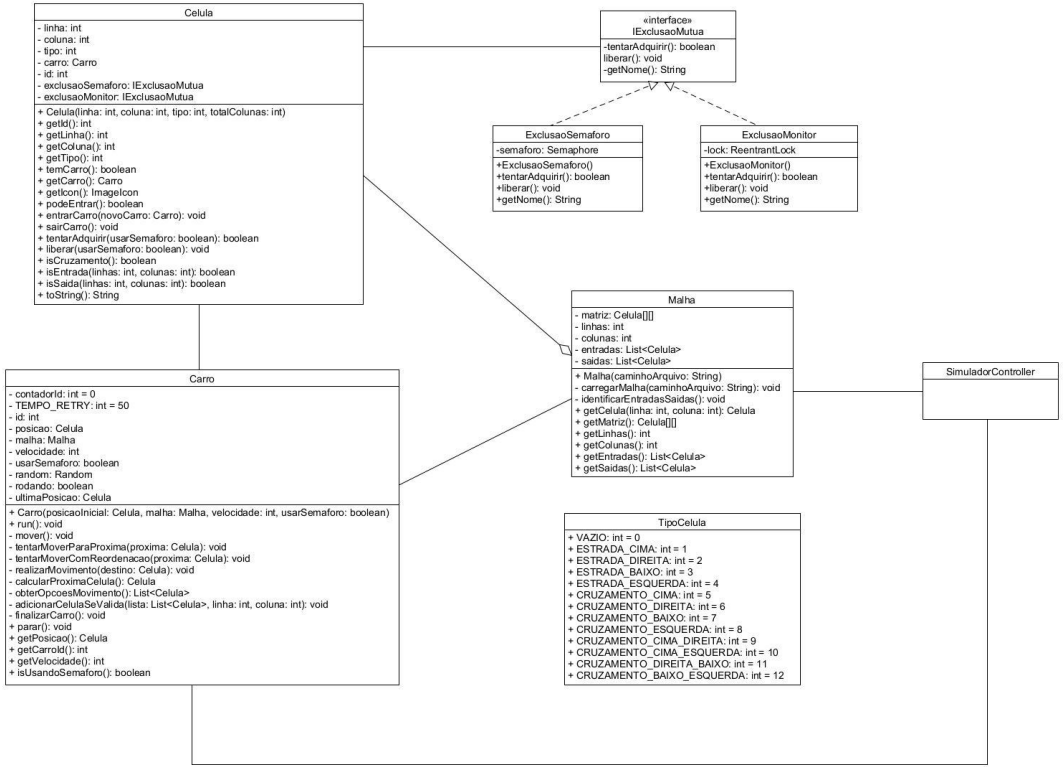


Diagrama de Classes

ESO
Engenharia de Software
UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Diagrama de Classes



Estrutura do Projeto

ESO
Engenharia de Software

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

TipoCelula

Define os tipos numéricos para os diferentes tipos de células.

```java
public class TipoCelula {

    // Célula vazia (sem estrada)
    public static final int VAZIO = 0;

    // Estradas direcionais (movimento único)
    public static final int ESTRADA_CIMA = 1;
    public static final int ESTRADA_DIREITA = 2;
    public static final int ESTRADA_BAIXO = 3;
    public static final int ESTRADA_ESQUERDA = 4;

    // Cruzamentos com uma saída possível
    public static final int CRUZAMENTO_CIMA = 5;
    public static final int CRUZAMENTO_DIREITA = 6;
    public static final int CRUZAMENTO_BAIXO = 7;
    public static final int CRUZAMENTO_ESQUERDA = 8;

    // Cruzamentos com duas saídas possíveis
    public static final int CRUZAMENTO_CIMA_DIREITA = 9;
    public static final int CRUZAMENTO_CIMA_ESQUERDA = 10;
    public static final int CRUZAMENTO_DIREITA_BAIXO = 11;
    public static final int CRUZAMENTO_BAIXO_ESQUERDA = 12;
}
```

SimuladorController

A classe SimuladorController é a responsável por coordenar toda a execução do sistema. Ela faz o controle do tempo da simulação, atualiza o movimento dos carros na malha e garante que o comportamento de cada componente (carros, células e a própria malha) ocorra de forma sincronizada. Em resumo, ela é o “cérebro” que faz a simulação funcionar.

```
public void inicializar(Malha malha, PainelMalha painel, int qtdCarros,
        int intervaloInsercao, boolean usarSemaforo) {
    // Encerra simulação anterior se estiver rodando
    if (this.rodando) {
        encerrarSimulacao();
        aguardarEncerramento();
    }

    this.malha = malha;
    this.painelMalha = painel;
    this.quantidadeMaxima = qtdCarros;
    this.intervaloInsercao = intervaloInsercao;
    this.usarSemaforo = usarSemaforo;
    this.inserindo = true;
    this.rodando = true;
    this.ultimaInsercao = 0;

    System.out.println("=== Simulação Inicializada ===");
    System.out.println("Quantidade máxima: " + qtdCarros + " carros");
    System.out.println("Velocidade base: " + VELOCIDADE_BASE + "ms");
    System.out.println("Intervalo de inserção: " + intervaloInsercao + "ms");
    System.out.println("Exclusão mútua: " + (usarSemaforo ? "Semáforo" : "Monitor"));
    System.out.println("==============================");
}
```

IExclusaoMutua

A interface IExclusaoMutua define o padrão que todas as classes de controle de acesso devem seguir, permitindo que apenas uma thread por vez acesse uma célula da malha. Seus métodos principais são tentarAdquirir(), que tenta acessar o recurso, liberar(), que o libera, e getNome(), que identifica o tipo de mecanismo usado.

```
public interface IExclusaoMutua {

    boolean tentarAdquirir();

    void liberar();

    String getNome();
}
```

ExclusaoMonitor

A classe ExclusaoMonitor implementa essa interface utilizando um ReentrantLock, funcionando como um monitor que controla o acesso exclusivo às células. O método tentarAdquirir() tenta obter o lock sem bloquear, enquanto liberar() o libera apenas se a thread atual for a dona.

```java
public class ExclusaoMonitor implements IExclusaoMutua {

    private final ReentrantLock lock;

    public ExclusaoMonitor() {
        this.lock = new ReentrantLock(true);
    }

    @Override
    public boolean tentarAdquirir() {
        return lock.tryLock();
    }

    @Override
    public void liberar() {
        if (lock.isHeldByCurrentThread()) {
            lock.unlock();
        }
    }

    @Override
    public String getNome() {
        return "Monitor";
    }
}
```

ExclusaoSemaforo

A classe ExclusaoSemaforo também implementa a interface, mas usa um Semáforo binário com uma única permissão, garantindo que apenas uma thread acesse a célula por vez.

Métodos:

- tentarAdquirir()
- liberar()
- getNome()

```java
public class ExclusaoSemaforo implements IExclusaoMutua {

    private final Semaphore semaforo;

    public ExclusaoSemaforo() {
        this.semaforo = new Semaphore(1, true);
    }

    @Override
    public boolean tentarAdquirir() {
        return semaforo.tryAcquire();
    }

    @Override
    public void liberar() {
        semaforo.release();
    }

    @Override
    public String getNome() {
        return "Semáforo";
    }
}
```

Carro

A classe Carro representa um veículo que navega pela malha como uma Thread independente. O método run() executa um loop que move o carro célula por célula até encontrar uma saída.

```java
@Override
public void run() {
    try {
        System.out.println("Carro " + id + " iniciou em " + posicao +
                           " com velocidade " + velocidade + "ms");

        while (rodando && posicao != null) {
            // Verifica se chegou na saída
            if (posicao.isSaida(malha.getLinhas(), malha.getColunas())) {
                System.out.println("Carro " + id + " chegou na saída");
                finalizarCarro();
                break;
            }

            // Tenta mover
            mover();

            // Aguarda baseado na velocidade
            Thread.sleep(velocidade);
        }
    } catch (InterruptedException e) {
        System.out.println("Carro " + id + " foi interrompido");
    } finally {
        finalizarCarro();
    }
}
```

Celula

A classe Célula representa uma unidade da malha viária, como se fosse um quadrado do mapa. Cada célula pode estar vazia, ocupada por um carro ou representar uma parte específica da via. Ela é fundamental para definir as regras de movimentação, já que indica se um carro pode ou não avançar para aquele ponto da malha.

```
public ImageIcon getIcon() {
    int idCarro = temCarro() ? carro.getCarroId() : -1;
    return IconeCelula.getInstance().getIconeCelula(tipo, idCarro);
}

public synchronized boolean podeEntrar() {
    return carro == null && tipo != TipoCelula.VAZIO;
}

public synchronized void entrarCarro(Carro novoCarro) {
    if (carro != null) {
        throw new IllegalStateException(
            "Célula [" + linha + "," + coluna + "] já possui um carro!"
        );
    }
    this.carro = novoCarro;
}
```

Malha

A classe Malha funciona como o ambiente da simulação, sendo responsável por organizar as células que formam o mapa por onde os carros circulam. Ela carrega a estrutura da malha viária a partir de um arquivo e mantém o controle de onde cada célula está posicionada, permitindo que o simulador saiba onde os carros podem se mover.

```
private void carregarMalha(String caminhoArquivo) {
    int[][] dados = ArquivoMalhaReader.lerMalha(caminhoArquivo);

    if (dados == null || dados.length == 0) {
        throw new IllegalArgumentException("Arquivo de malha inválido");
    }

    this.linhas = dados.length;
    this.colunas = dados[0].length;
    this.matriz = new Celula[linhas][colunas];

    // Cria células passando totalColunas para cálculo de ID único
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            int tipo = dados[i][j];
            matriz[i][j] = new Celula(i, j, tipo, colunas);
        }
    }
}
```

ArquivoMalhaReader

Responsável por ler arquivos de configuração de malha.

Método lerMalha():

- Abre arquivo de texto;
- Lê primeira linha: quantidade de linhas;
- Lê segunda linha: quantidade de colunas;
- Lê linhas seguintes: valores da matriz (separados por espaços);
- Retorna matriz de inteiros;
- Trata exceções de IO;

```
try (BufferedReader br = new BufferedReader(new FileReader(caminhoArquivo))) {
    String linha;
    int contador = 0;
    int qtdLinhas = 0;
    int qtdColunas = 0;

    while ((linha = br.readLine()) != null) {
        linha = linha.trim();
        if (linha.isEmpty()) {
            continue;
        }

        if (contador == 0) {
            // Primeira linha: quantidade de linhas
            qtdLinhas = Integer.parseInt(linha);
        } else if (contador == 1) {
            // Segunda linha: quantidade de colunas
            qtdColunas = Integer.parseInt(linha);
        } else {
            // Linhas seguintes: dados da matriz
            String[] partes = linha.split("\\s+");
            int[] valores = new int[qtdColunas];
            for (int i = 0; i < qtdColunas && i < partes.length; i++) {
                valores[i] = Integer.parseInt(partes[i]);
            }
            linhas.add(valores);
        }
        contador++;
    }
```

IconeCelula

A classe IconManager é responsável por cuidar da parte visual da simulação, sendo ela quem define os ícones e as cores que representam cada elemento na malha. Ela associa imagens e cores diferentes para indicar, por exemplo, ruas, cruzamentos, carros e áreas vazias. Além disso, possui métodos que atualizam a aparência das células conforme a simulação acontece.

```
private void carregarImagensBase() {
    try {
        setas.put(DirecaoSeta.CIMA, carregarIcone("seta_cima.png"));
        setas.put(DirecaoSeta.BAIXO, carregarIcone("seta_baixo.png"));
        setas.put(DirecaoSeta.DIREITA, carregarIcone("seta_direita.png"));
        setas.put(DirecaoSeta.ESQUERDA, carregarIcone("seta_esquerda.png"));

        System.out.println("Setas carregadas com sucesso");
    } catch (Exception e) {
        System.err.println("Erro ao carregar setas: " + e.getMessage());
        e.printStackTrace();
    }
}
```

PainelMalha

A classe PainelMalha atua como o painel gráfico principal onde a malha é desenhada. Ela organiza as células em uma grade (grid) e exibe o estado atual de cada uma, atualizando a interface sempre que algo muda. Um método importante nela é o de atualização da malha, que redesenha o painel em tempo real durante a simulação.

```
private void configurarPainel() {
    int largura = malha.getColunas() * TAMANHO_CELULA;
    int altura = malha.getLinhas() * TAMANHO_CELULA;

    Dimension tamanho = new Dimension(largura, altura);
    setSize(tamanho);
    setPreferredSize(tamanho);
    setDoubleBuffered(true); // Ativa double buffering para suavidade
    setBackground(Color.BLACK);
}
```

TelaSimulacao

A classe TelaSimulacao representa a interface principal onde a simulação de fato acontece. Ela exibe a malha com os carros em movimento, além de mostrar informações em tempo real, como o número de carros ativos. Um dos métodos mais importantes dessa classe é o que atualiza os dados da simulação, garantindo que tudo o que acontece na lógica seja refletido visualmente.

```
try {
    // Obtém instância (pode ser nova ou existente)
    controller = SimuladorController.getInstance();

    // Se já estiver rodando, para primeiro
    if (controller.isAlive()) {
        System.out.println("Controller anterior ainda ativo, aguardando...");
        controller.encerrarSimulacao();
        controller.join(1000);
    }

    // Cria nova instância garantida
    controller = SimuladorController.getInstance();

    // Inicializa sem parâmetro de velocidade (velocidade base é fixa)
    controller.inicializar(malha, painelMalha, qtdCarros,
                           intervaloInsercao, usarSemaforo);
    controller.start();

    System.out.println("Simulação iniciada com sucesso");
```

TelaInicial

A classe TelaInicial funciona como a tela de entrada do sistema. É nela que o usuário faz as configurações antes de começar a simulação. Ela também é responsável por iniciar a simulação, chamando o controlador principal quando o usuário clica para começar.

```
try {
    int qtdCarros = (int) spnQtdCarros.getValue();
    int intervaloInsercao = (int) spnIntervaloInsercao.getValue();
    String nomeMalha = (String) cmbMalha.getSelectedItem();
    boolean usarSemaforo = "Semáforo".equals(cmbExclusao.getSelectedItem());

    if (nomeMalha == null || nomeMalha.equals("Nenhuma malha encontrada")) {
        JOptionPane.showMessageDialog(this,
                "Selecione uma malha válida!",
                "Erro", JOptionPane.ERROR_MESSAGE);
        return;
    }

    String caminhoMalha = "malhas/" + nomeMalha;
    Malha malha = new Malha(caminhoMalha);

    // Velocidade base fixa
    // Os carros terão velocidades diferentes automaticamente (±30%)
    TelaSimulacao telaSimulacao = new TelaSimulacao(
            malha, qtdCarros, intervaloInsercao, usarSemaforo
    );
    telaSimulacao.setVisible(true);

    dispose();
```

Execução do Sistema

ESO
Engenharia de Software
UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Dificuldades na implementação

Dificuldades

- Montagem da estrutura;
- Carros percorrerem corretamente a malha;
- Integrar os critérios de exclusão;
- Utilizar o recurso do Graphics para melhor visualização dos “carros”;
- Encontrar os pontos certos para debugar corretamente;

Obrigado!

Desenvolvimento de um Sistema Paralelo utilizando Threads

ESO
Engenharia de Software

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA