

The background is a light gray with various abstract shapes and dots. There are orange, blue, and red outlines of irregular shapes, some of which are filled with small dots of the same color. There are also small circles and a triangle in the background.

Threads

Gabriela e Ariadne

Estrutura do projeto

Config

Contém classes relacionadas à configuração visual da simulação

Constantes

Define constantes utilizadas para classificar e tipificar as células

Controller

Controla a lógica de operação e execução da simulação

Model

Contém as classes modelos

View

Contém as classes modelos

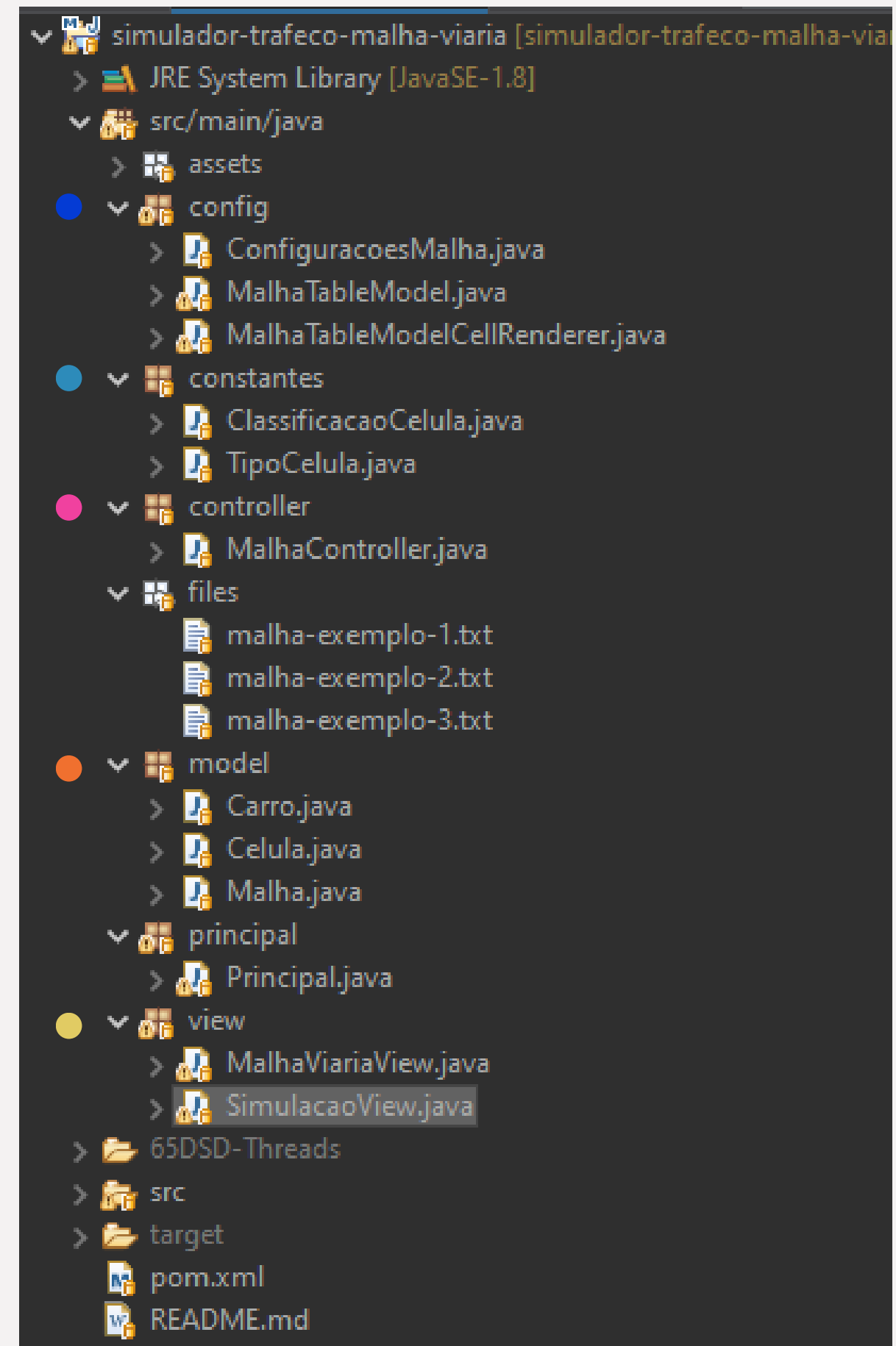


Diagrama de Classes

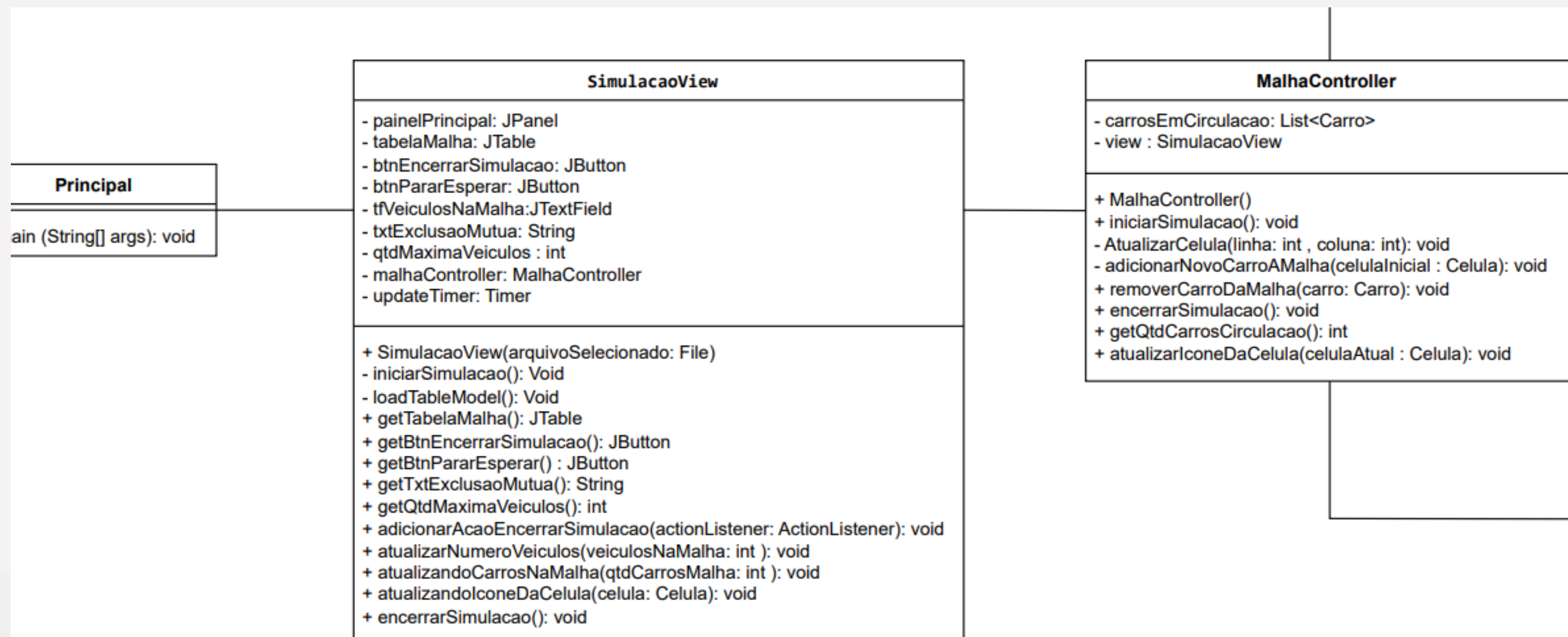


Diagrama de Classes

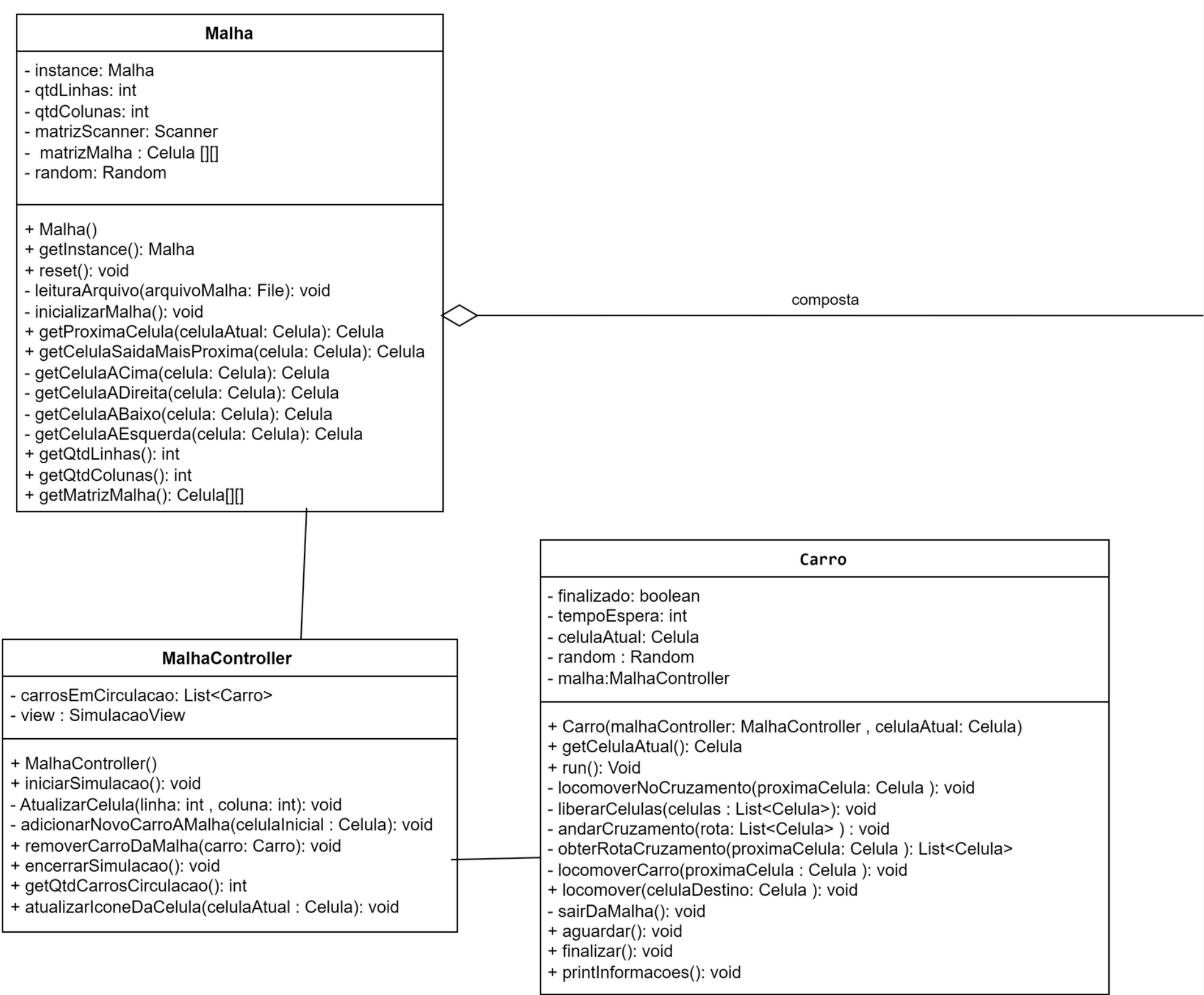


Diagrama de Classes

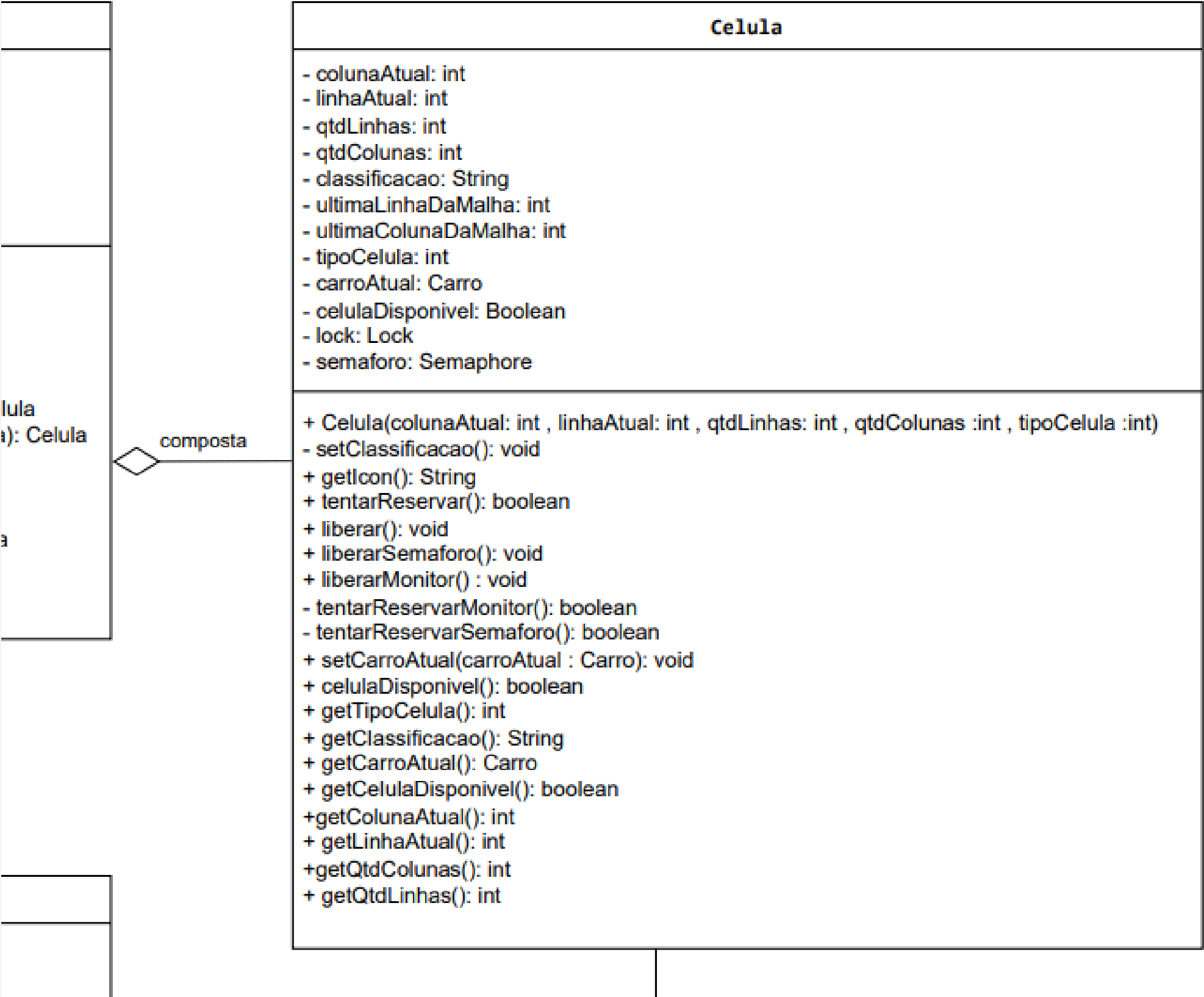
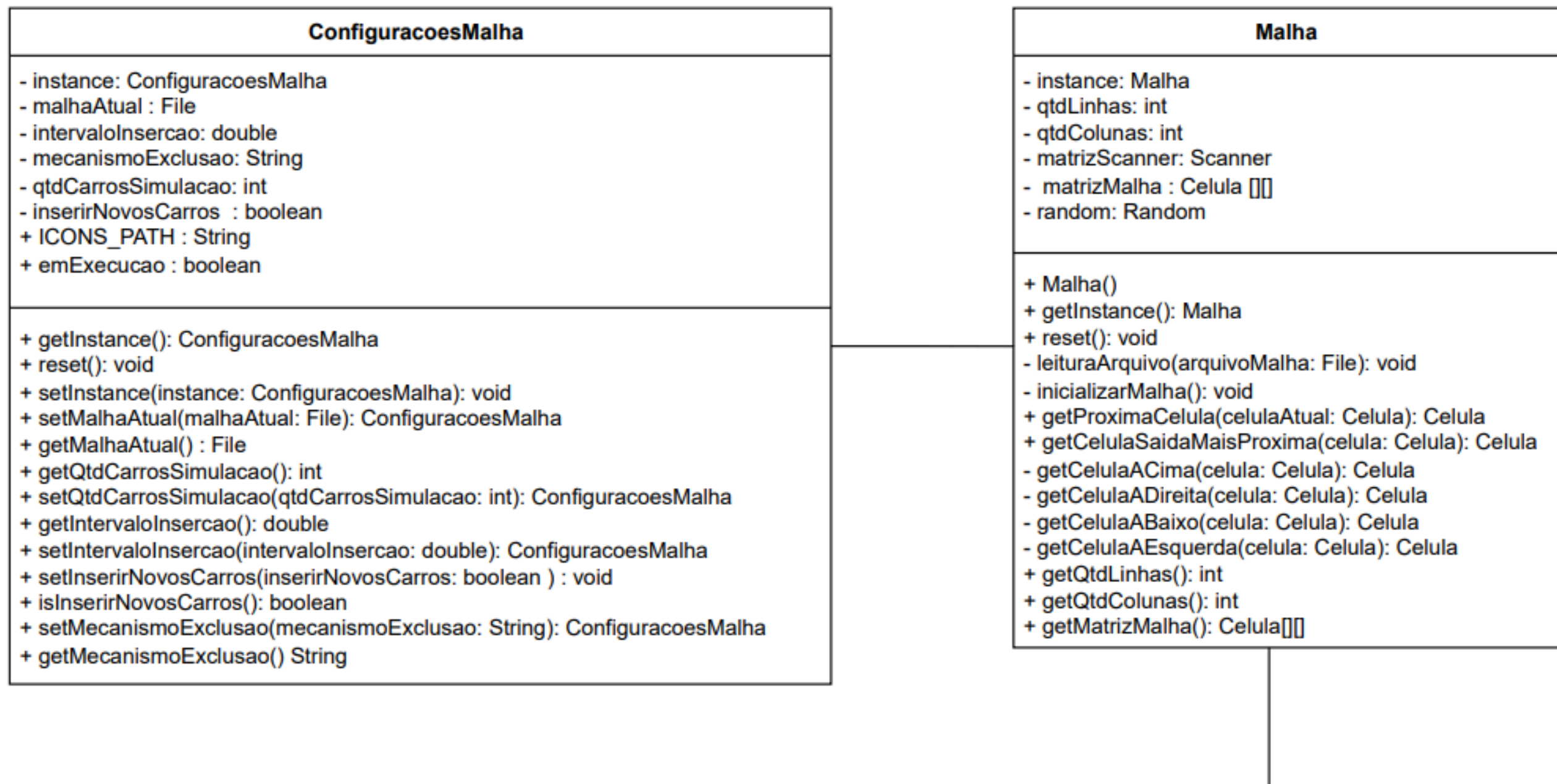


Diagrama de Classes



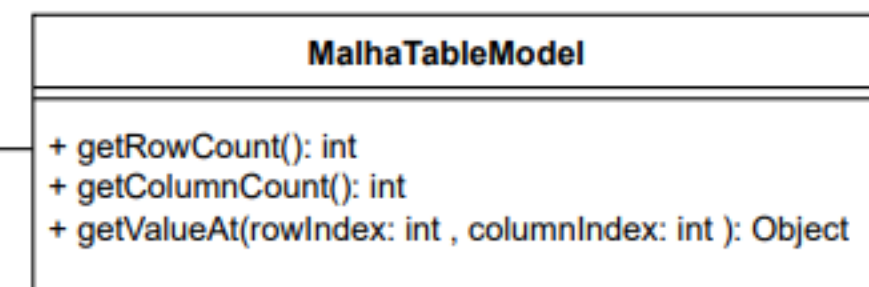
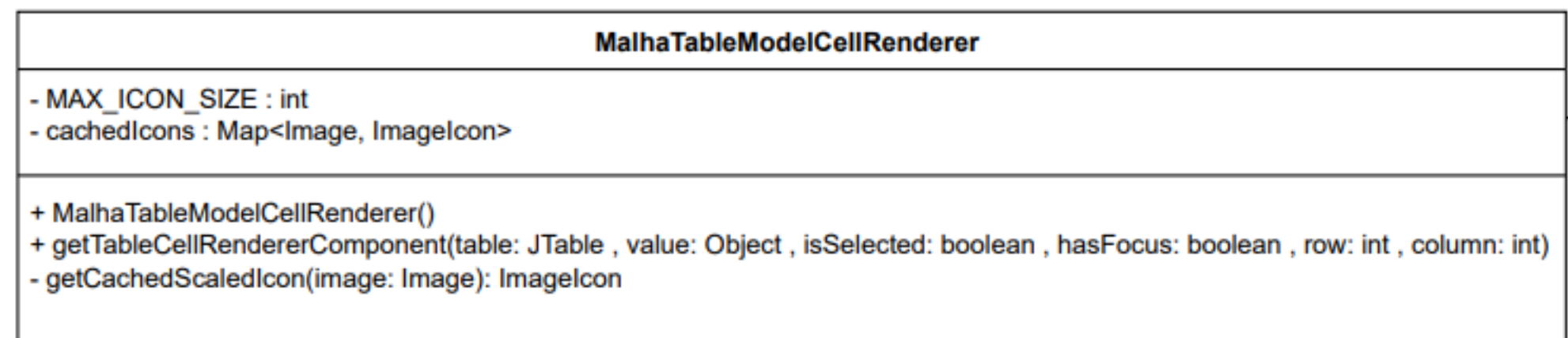
MalhaViariaView
<div>- contentPane: JPanel</div> <div>- textField_1 : JTextField</div> <div>- textField_2 : JTextField</div> <div>- arquivoSelecionado : File</div> <div>- qtdMaxVeiculos:int</div>
<div>+ MalhaViariaView()</div> <div>+ getBotaoSelecionado(): String</div> <div>+ verificaPadraoDoArquivo(arquivo: File): boolean</div> <div>+ getArquivoSelecionado(): File</div> <div>+ setArquivoSelecionado(arquivoSelecionado : File): Void</div> <div>+ getQtdMaxVeiculos() : int</div> <div>+ setQtdMaxVeiculos(qtdMaxVeiculos : Int): Void</div>

ClassificacaoCelula
<div>+ VAZIO: String</div> <div>+ ESTRADA : String</div> <div>+ CRUZAMENTO : String</div> <div>+ ENTRADA : String</div> <div>+ SAIDA : String</div>

TipoCelula
<div>+ VAZIO: int</div> <div>+ ESTRADA_CIMA : int</div> <div>+ ESTRADA_DIREITA : int</div> <div>+ ESTRADA_BAIXO : int</div> <div>+ ESTRADA_ESQUERDA : int</div> <div>+ CRUZAMENTO_CIMA : int</div> <div>+ CRUZAMENTO_DIREITA : int</div> <div>+ CRUZAMENTO_BAIXO : int</div> <div>+ CRUZAMENTO_ESQUERDA : int</div> <div>+ CRUZAMENTO_CIMA_E_DIREITA : int</div> <div>+ CRUZAMENTO_CIMA_E_ESQUERDA : int</div> <div>+ CRUZAMENTO_DIREITA_E_BAIXO : int</div> <div>+ CRUZAMENTO_BAIXO_E_ESQUERDA : int</div> <div>+ ESTRADAS : int[]</div> <div>+ CRUZAMENTOS : int[]</div>

Diagrama de Classes

Diagrama de Classes



Fluxo do Sistema

01



O sistema começa na classe Principal, que inicializa a interface MalhaViariaView.

```
public static void main(String[] args) {  
    MalhaViariaView malha = new MalhaViariaView();  
}
```

Usuário seleciona malha, intervalo, exclusão mútua e inicia simulação

02



Essas configurações são salvas em ConfiguracoesMalha, e após clicar pra iniciar simulação abre a tela de simulacaoView

03



Controller vai gerenciar a inserção dos carros verificando as entradas disponíveis

```
public void iniciarSimulacao() {  
    ConfiguracoesMalha.getInstance().emExecucao = true;  
    while (ConfiguracoesMalha.getInstance().emExecucao) {  
        // Loop para inserir novos carros  
        while (ConfiguracoesMalha.getInstance().isInserirNovosCarros()  
            && ConfiguracoesMalha.getInstance().emExecucao) {  
            for (int linha = 0; linha < Malha.getInstance().getQtdLinhas(); linha++) {  
                for (int coluna = 0; coluna < Malha.getInstance().getQtdColunas(); coluna++) {  
                    this.AtualizarCelula(linha, coluna);  
                    try {  
                        Thread.sleep(10); // Controla o tempo de atualização para ser gradual  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        }  
    }  
}
```

Fluxo do Sistema

04



A malha controller faz as devidas chamadas das classes modelos que são responsáveis pela parte de execução de movimentos

05

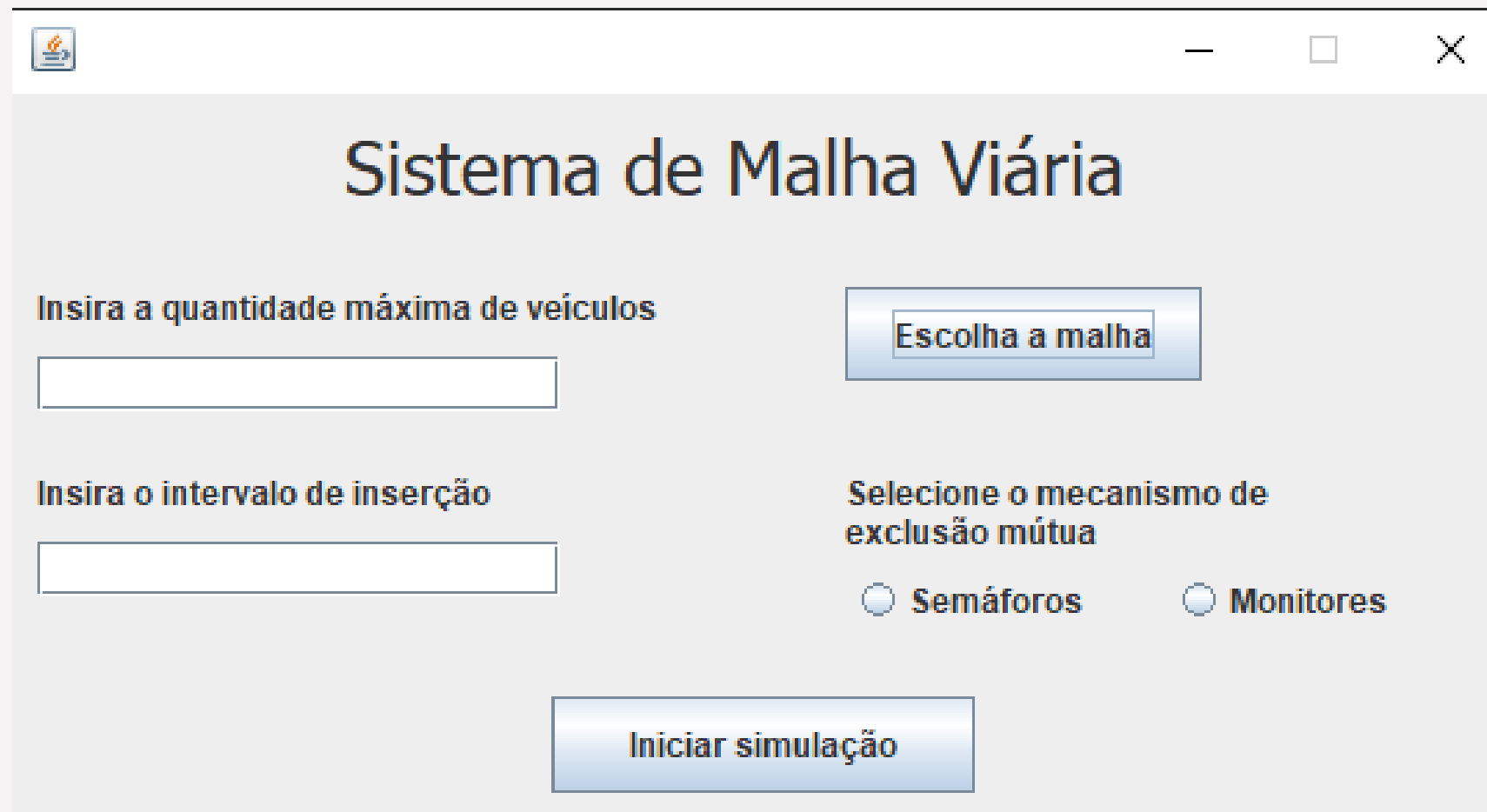


E por fim encerra o processo na SimulacaoView quando clicado em Encerrar.



**Como foi
implementado
sistema ?**

MalhaViariaView e ConfiguracoesMalha



Sistema de Malha Viária

Insira a quantidade máxima de veículos

Insira o intervalo de inserção

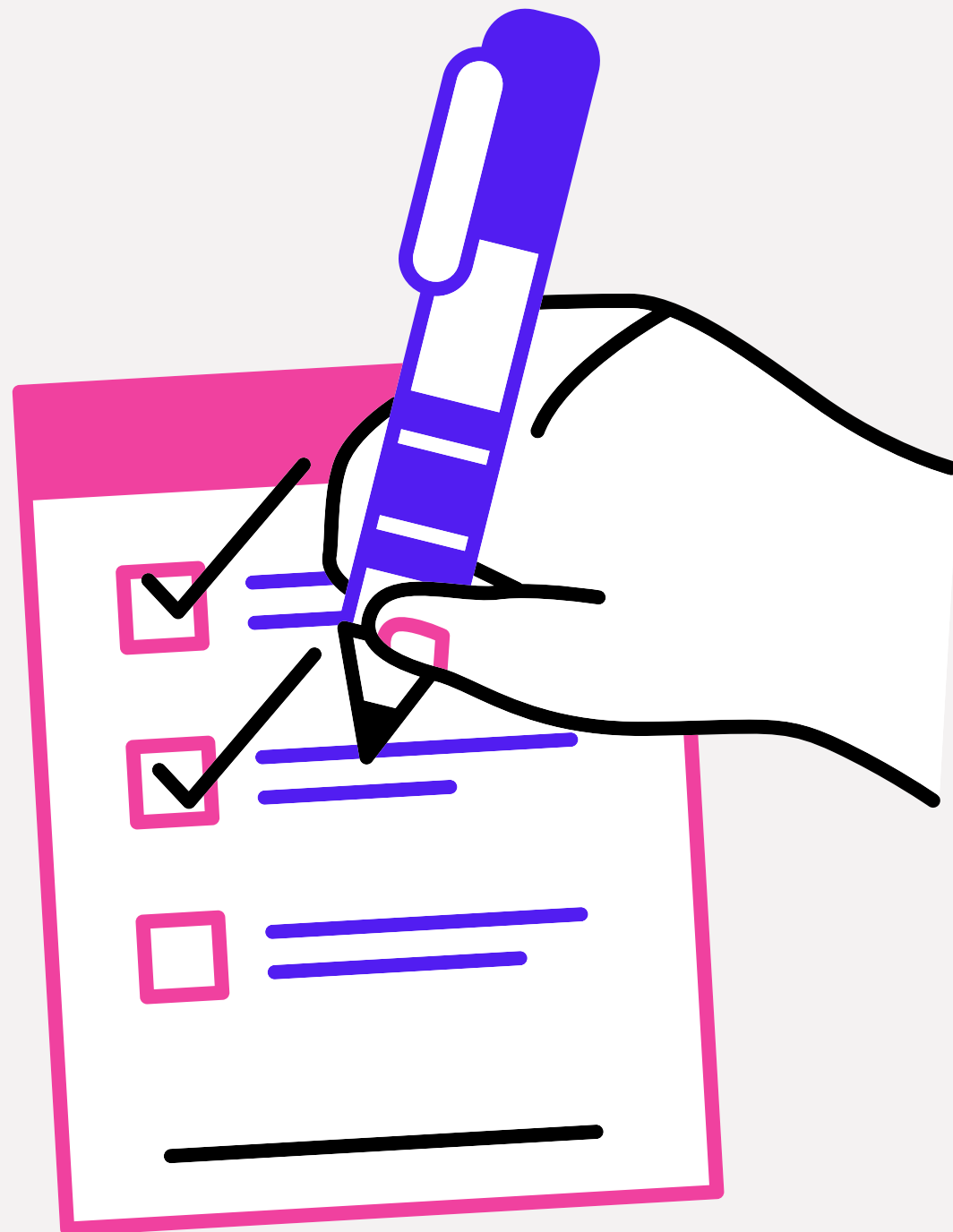
Escolha a malha

Selecione o mecanismo de exclusão mútua

☐ Semáforos ☐ Monitores

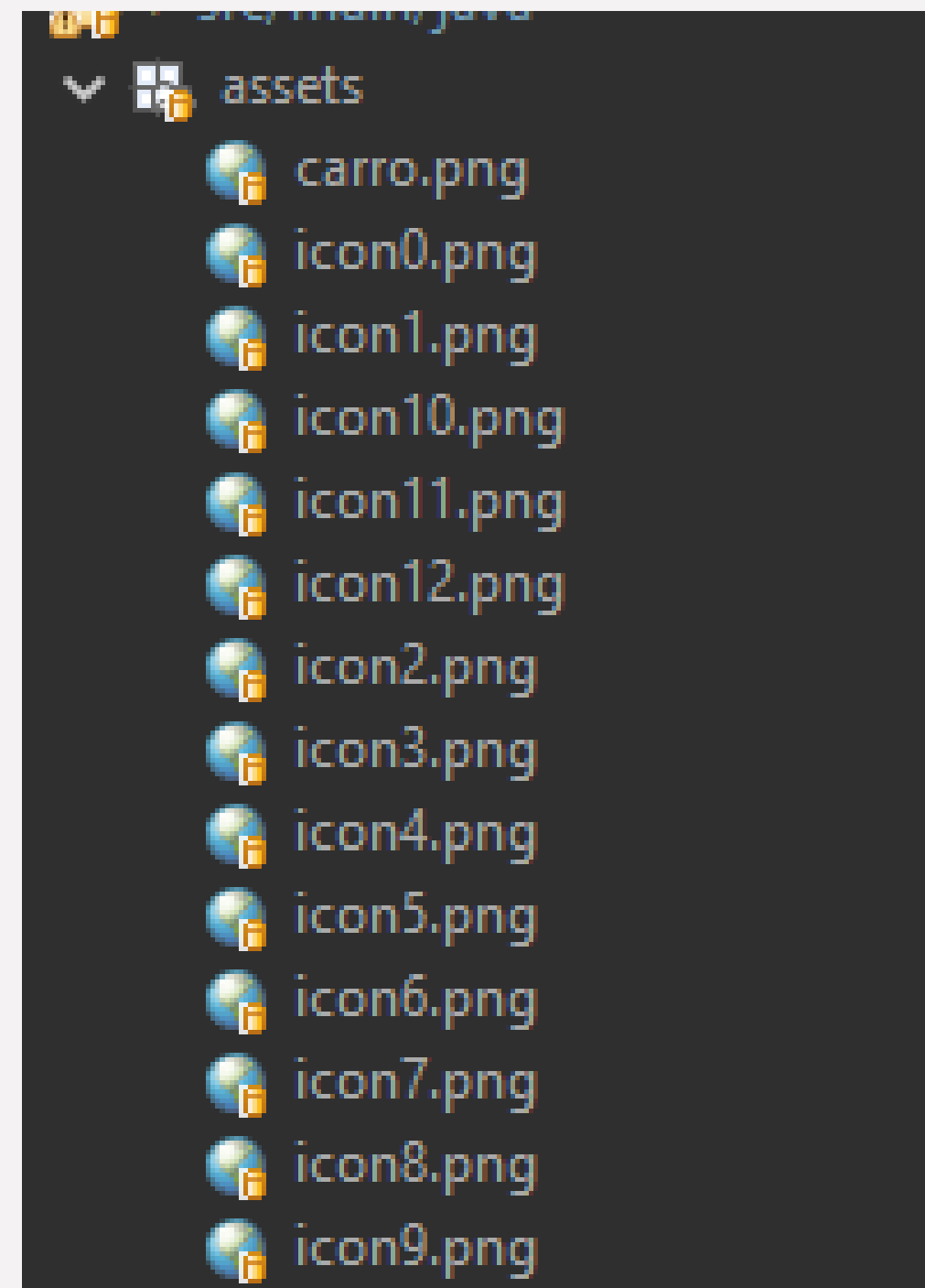
Iniciar simulação

```
public class ConfiguracoesMalha {  
  
    private static ConfiguracoesMalha instance;  
    private File malhaAtual;  
    private double intervaloInsercao;  
    private String mecanismoExclusao;  
    private int qtdCarrosSimulacao;  
    private boolean inserirNovosCarros = true;  
  
    public static final String ICONS_PATH = "src/main/java/assets/";  
    public boolean emExecucao = false;  
  
    public static synchronized ConfiguracoesMalha getInstance() {  
        if (instance == null)  
            reset();  
        return instance;  
    }  
  
    public static synchronized void reset() {  
        instance = new ConfiguracoesMalha();  
    }  
  
    public static void setInstance(ConfiguracoesMalha instance) {  
        ConfiguracoesMalha.instance = instance;  
    }  
  
    public ConfiguracoesMalha setMalhaAtual(File malhaAtual) {  
        this.malhaAtual = malhaAtual;  
        return instance;  
    }  
}
```



Assets

Somente os icones e de acordo com o tipo de segmento



Config

MalhaTableModel: Estende `AbstractTableModel` e define como a malha é exibida em uma tabela.

```
public class MalhaTableModel extends AbstractTableModel {  
  
    @Override  
    public int getRowCount() {  
        return Malha.getInstance().getQtdLinhas();  
    }  
  
    @Override  
    public int getColumnCount() {  
        return Malha.getInstance().getQtdColunas();  
    }  
  
    @Override  
    public Object getValueAt(int rowIndex, int columnIndex) {  
        return new ImageIcon(Malha.getInstance().getMatrizMalha()[rowIndex][columnIndex].getIcon());  
    }  
}
```

Ele busca a quantidade de linhas e colunas a partir da instância de `Malha` e utiliza `ImageIcon` para renderizar as células com base no estado atual da malha.



Config

MalhaTableModelCellRenderer:
Implementa TableCellRenderer
para customizar a exibição das
células da tabela.

Garante que os ícones sejam
escalados para um tamanho
adequado e gerencia a cache de
ícones escalados para melhorar
a performance.

```
public class MalhaTableModelCellRenderer extends JLabel implements TableCellRenderer {

    private static final int MAX_ICON_SIZE = 32;
    private Map<Image, ImageIcon> cachedIcons = new HashMap<>();

    public MalhaTableModelCellRenderer() {
        setHorizontalAlignment(SwingConstants.CENTER);
    }

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {
        if (value instanceof ImageIcon) {
            ImageIcon icon = (ImageIcon) value;
            Image image = icon.getImage();

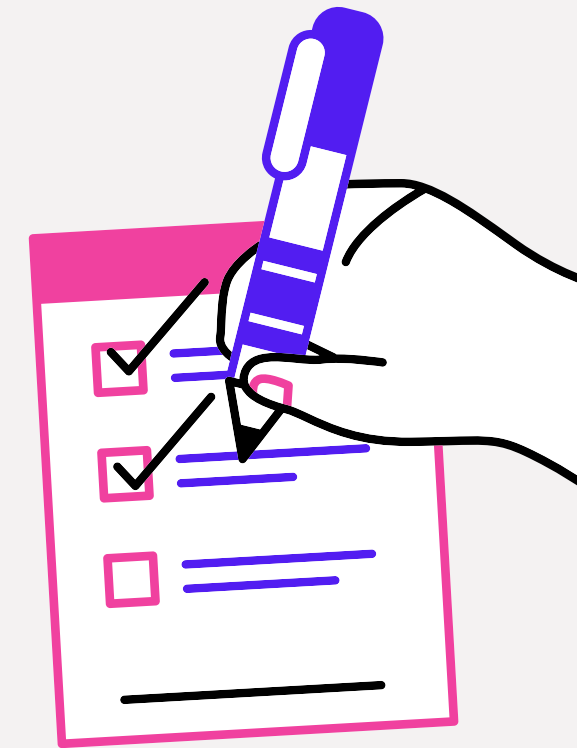
            if (image != null) {
                ImageIcon scaledIcon = getCachedScaledIcon(image);
                setIcon(scaledIcon);
            } else {
                setIcon(null);
            }
        } else {
            setText(value.toString());
        }

        if (isSelected) {
            setBackground(table.getSelectionBackground());
            setForeground(table.getSelectionForeground());
        } else {
            setBackground(table.getBackground());
            setForeground(table.getForeground());
        }

        return this;
    }

    private ImageIcon getCachedScaledIcon(Image image) {
        if (!cachedIcons.containsKey(image)) {
            int width = Math.min(MAX_ICON_SIZE, image.getWidth(null));
            int height = Math.min(MAX_ICON_SIZE, image.getHeight(null));
            Image scaledImage = image.getScaledInstance(width, height, Image.SCALE_SMOOTH);
            cachedIcons.put(image, new ImageIcon(scaledImage));
        }
        return cachedIcons.get(image);
    }
}
```

Constantes



ClassificacaoCelula: Definição do comportamento das células na malha.

```
public class TipoCelula {
    public static final int VAZIO = 0;
    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;
    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;
    public static final int CRUZAMENTO_CIMA_E_DIREITA = 9;
    public static final int CRUZAMENTO_CIMA_E_ESQUERDA = 10;
    public static final int CRUZAMENTO_DIREITA_E_BAIXO = 11;
    public static final int CRUZAMENTO_BAIXO_E_ESQUERDA = 12;

    public static final int[] ESTRADAS = {ESTRADA_CIMA, ESTRADA_DIREITA, ESTRADA_BAIXO, ESTRADA_ESQUERDA};
    public static final int[] CRUZAMENTOS = {
        CRUZAMENTO_CIMA, CRUZAMENTO_DIREITA, CRUZAMENTO_BAIXO,
        CRUZAMENTO_ESQUERDA, CRUZAMENTO_CIMA_E_DIREITA, CRUZAMENTO_CIMA_E_ESQUERDA,
        CRUZAMENTO_DIREITA_E_BAIXO, CRUZAMENTO_BAIXO_E_ESQUERDA,
    };
}
```

TipoCelula: Define tipos numéricos para diferentes direções e cruzamentos...

```
*/
public class ClassificacaoCelula {
    public static final String VAZIO = "VAZIO";
    public static final String ESTRADA = "ESTRADA";
    public static final String CRUZAMENTO = "CRUZAMENTO";
    public static final String ENTRADA = "ENTRADA";
    public static final String SAIDA = "SAIDA";
}
```

Controller

```
public class MalhaController {

    private List<Carro> carrosEmCirculacao;
    private SimulacaoView view;

    public MalhaController(SimulacaoView view) {
        this.carrosEmCirculacao = new ArrayList<>();
        this.view = view;
        view.atualizandoCarrosNaMalha(0);
    }

    public void iniciarSimulacao() {
        ConfiguracoesMalha.getInstance().emExecucao = true;
        while (ConfiguracoesMalha.getInstance().emExecucao) {
            // Loop para inserir novos carros
            while (ConfiguracoesMalha.getInstance().isInserirNovosCarros()
                    && ConfiguracoesMalha.getInstance().emExecucao) {
                for (int linha = 0; linha < Malha.getInstance().getQtdLinhas(); linha++) {
                    for (int coluna = 0; coluna < Malha.getInstance().getQtdColunas(); coluna++) {
                        this.AtualizarCelula(linha, coluna);
                        try {
                            Thread.sleep(10); // Controla o tempo de atualização para ser gradual
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }

}
```

```
public void encerrarSimulacao() {
    view.encerrarSimulacao();
}

public int getQtdCarrosCirculacao() {
    return this.carrosEmCirculacao.size();
}

public void atualizarIconeDaCelula(Celula celulaAtual) {
    view.atualizandoIconeDaCelula(celulaAtual);
}
```

```
private void AtualizarCelula(int linha, int coluna) {

    if (!ConfiguracoesMalha.getInstance().isInserirNovosCarros()
        || this.getQtdCarrosCirculacao() >= ConfiguracoesMalha.getInstance().getQtdCarrosSimulacao()) {
        return;
    }

    Celula celulaAtual = Malha.getInstance().getMatrizMalha()[linha][coluna];
    if (!celulaAtual.getClassificacao().equals(ClassificacaoCelula.ENTRADA)) // Tem de ser entrada
        return;
    if (!celulaAtual.celulaDisponivel()) // Tem de estar disponível
        return;
    try {
        adicionarNovoCarroAMalha(celulaAtual);
    } catch (Exception e) {
        System.out.println(e.getMessage() + " - " + Arrays.toString(e.getStackTrace()));
    }
}

private void adicionarNovoCarroAMalha(Celula celulaInicial) {
    Carro carro = new Carro(this, celulaInicial);

    carrosEmCirculacao.add(carro);
    view.atualizandoCarrosNaMalha(this.getQtdCarrosCirculacao());
    view.atualizandoIconeDaCelula(celulaInicial);
    carro.printInformacoes();
    carro.start();
}

public void removerCarroDaMalha(Carro carro) {
    this.carrosEmCirculacao.remove(carro);
    Celula celula = carro.getCelulaAtual();
    celula.setCarroAtual(null);
    view.atualizandoCarrosNaMalha(this.getQtdCarrosCirculacao());
    view.atualizandoIconeDaCelula(celula);
}
```

MalhaController: Gerencia a inserção de carros na malha, atualiza células e controla quando carros são adicionados ou removidos da malha. Mantém a simulacaoView atualizada.

Model

Celula: Representa uma célula na malha e armazena informações sobre sua posição, tipo, e se está disponível para carros. (Recurso compartilhado)

Usa mecanismos de controle como Semaphore e Lock para gerenciar o acesso simultâneo de carros, dependendo da configuração selecionada.

```
public class Celula {  
  
    private int colunaAtual;  
    private int linhaAtual;  
    private int qtdLinhas;  
    private int qtdColunas;  
    private String classificacao;  
    private int ultimaLinhaDaMalha;  
    private int ultimaColunaDaMalha;  
    private int tipoCelula;  
    private Carro carroAtual;  
    private Boolean celulaDisponivel;  
    private Lock lock;  
    private Semaphore semaforo;  
  
}
```

```
public boolean tentarReservar() {  
    if (this.carroAtual != null) {  
        return false;  
    }  
  
    if (ConfiguracoesMalha.getInstance().getMecanismoExclusao() == "Semáforos") {  
        return tentarReservarSemaforo();  
    } else {  
        return tentarReservarMonitor();  
    }  
}  
  
public void liberar() {  
    if (ConfiguracoesMalha.getInstance().getMecanismoExclusao() == "Semáforos")  
        this.liberarSemaforo();  
    else  
        this.liberarMonitor();  
}  
  
public void liberarSemaforo() {  
    try {  
        this.semaforo.release();  
    } catch (Exception e) {  
    }  
}  
  
public void liberarMonitor() {  
    try {  
        this.lock.unlock();  
    } catch (Exception e) {  
    }  
}
```

```
private boolean tentarReservarMonitor() {  
    try {  
        return this.lock.tryLock(100, TimeUnit.MILLISECONDS);  
    } catch (InterruptedException e) {  
        System.out.println(e.getStackTrace());  
        return false;  
    }  
}  
  
private boolean tentarReservarSemaforo() {  
    try {  
        return this.semaforo.tryAcquire(100, TimeUnit.MILLISECONDS);  
    } catch (InterruptedException e) {  
        System.out.println(e.getStackTrace());  
        return false;  
    }  
}
```

Exclusão mútua: Semáforo

O Semáforo é usado para controlar o acesso às células quando múltiplos carros tentam entrar ao mesmo tempo.

Cada célula possui um semáforo que permite apenas um carro de cada vez.

Quando um carro quer entrar na célula, ele tenta adquirir o semáforo. Se conseguir, ele se move para a célula; se não, espera até que o semáforo seja liberado por outro carro que está saindo.

```
private boolean tentarReservarSemaforo() {  
    try {  
        return this.semaforo.tryAcquire(100, TimeUnit.MILLISECONDS);  
    } catch (InterruptedException e) {  
        System.out.println(e.getStackTrace());  
        return false;  
    }  
}
```

Tenta pegar o semáforo por 100ms. Se conseguir, o carro pode entrar na célula; se não, ele espera e tenta de novo mais tarde.

```
public void liberarSemaforo() {  
    try {  
        this.semaforo.release();  
    } catch (Exception e) {  
    }  
}
```

Libera o semáforo para que outros carros possam acessar a célula depois que o carro atual sair.

Exclusão mútua: Monitor

É outra forma de garantir que apenas um carro ocupe uma célula por vez.

Quando o projeto está configurado para usar monitores, cada célula tem um lock que é tentado pelo carro antes de se mover para ela.

Se o lock está disponível, o carro pode se mover; se não, ele espera. Assim, apenas um carro consegue acessar a célula de cada vez, evitando colisões.

```
private boolean tentarReservarMonitor() {  
    try {  
        return this.lock.tryLock(100, TimeUnit.MILLISECONDS);  
    } catch (InterruptedException e) {  
        System.out.println(e.getStackTrace());  
        return false;  
    }  
}
```

Tenta bloquear o acesso à célula usando um lock (monitor) por até 100ms. Se conseguir, o carro pode entrar na célula; se não, o carro espera e tenta de novo depois.

```
public void liberarMonitor() {  
    try {  
        this.lock.unlock();  
    } catch (Exception e) {  
    }  
}
```

Libera o lock para permitir que outros carros possam acessar a célula depois que o carro atual sair. Isso desbloqueia a célula para uso.

Model

Malha: Singleton responsável por ler e armazenar a estrutura da malha a partir de um arquivo.

Inicializa a matriz de Celula, e gerencia a movimentação dos carros, determinando para onde cada carro deve se mover com base nas regras definidas em TipoCelula

```
public class Malha {
    private static Malha instance;
    private int qtdLinhas;
    private int qtdColunas;
    private Scanner matrizScanner;
    private Celula matrizMalha[][];

    private Random random = new Random();

    public Malha() {
        this.leituraArquivo(ConfiguracoesMalha.getInstance().getMalhaAtual());
        this.inicializarMalha();
    }

    public synchronized static Malha getInstance() {
        if (instance == null) {
            reset();
        }
        return instance;
    }

    public synchronized static void reset() {
        instance = new Malha();
    }
}
```

```
private void leituraArquivo(File arquivoMalha) {
    try {
        matrizScanner = new Scanner(arquivoMalha);
        this.qtdLinhas = matrizScanner.nextInt();
        this.qtdColunas = matrizScanner.nextInt();
        this.matrizMalha = new Celula[qtdLinhas][qtdColunas];
    } catch (Exception e) {
        System.out.println(e.getMessage()+" - "+ Arrays.toString(e.getStackTrace()));
    }
}

private void inicializarMalha() {
    while(matrizScanner.hasNextInt()) {
        for(int linha = 0; linha < this.qtdLinhas; linha++) {
            for(int coluna = 0 ; coluna < this.qtdColunas; coluna++) {
                int tipoCelula = matrizScanner.nextInt();
                Celula celulaAtual = new Celula(coluna, linha, qtdLinhas, qtdColunas, tipoCelula);
                this.matrizMalha [linha][coluna] = celulaAtual;
            }
        }
    }
}
```

Model

Run: Controla o ciclo de vida do carro, movimentando-o enquanto a simulação está ativa.

```
@Override
public void run() {
    while (ConfiguracoesMalha.getInstance().emExecucao && !this.finalizado) {
        Celula proximaCelula = Malha.getInstance().getProximaCelula(celulaAtual);

        if (proximaCelula == null) {
            sairDaMalha();
        } else if (proximaCelula.getClassificacao().equals(ClassificacaoCelula.CRUZAMENTO)) {
            locomoverNoCruzamento(proximaCelula);
        } else {
            locomoverCarro(proximaCelula);
        }
    }
    this.finalizar();
}
```

Define se o carro deve seguir para uma nova célula, atravessar um cruzamento ou sair da malha.

```
private List<Celula> obterRotaCruzamento(Celula proximaCelula) {
    LinkedList<Celula> rotaCruzamento = new LinkedList<Celula>();
    rotaCruzamento.add(proximaCelula);

    while (rotaCruzamento.getLast().getClassificacao().equals(ClassificacaoCelula.CRUZAMENTO)) {
        if (rotaCruzamento.size() >= 3) {
            proximaCelula = Malha.getInstance().getCelulaSaidaMaisProxima(proximaCelula);
        } else {
            proximaCelula = Malha.getInstance().getProximaCelula(proximaCelula);
        }
        rotaCruzamento.add(proximaCelula);
    }

    return rotaCruzamento;
}
```

Carro: Simula o comportamento dos carros na malha de tráfego, funcionando como uma Thread para permitir movimentos independentes

```
private void locomoverNoCruzamento(Celula proximaCelula) {
    List<Celula> rotaCruzamento = obterRotaCruzamento(proximaCelula);
    boolean locomoveu = false;
    while (!locomoveu) {
        List<Celula> celulasReservadas = new ArrayList<>();

        for (Celula celula : rotaCruzamento) {
            if (!celula.tentarReservar()) {
                liberarCelulas(celulasReservadas);
                try {
                    sleep(100 + random.nextInt(1000));
                } catch (Exception e) {
                    System.out.println(e);
                    System.out.println(e.getMessage());
                }
                break;
            }

            celulasReservadas.add(celula);
            locomoveu = celulasReservadas.size() == rotaCruzamento.size();
        }
        andarCruzamento(rotaCruzamento);
    }
}
```

LocomoverNoCruzamento: Gera uma rota através de um cruzamento, reservando todas as células necessárias para evitar bloqueios. Se não puder atravessar, espera e tenta novamente.

Padrões de projeto

Singleton

MVC



Parte Principal do sistema

Como os carros decidem para onde se mover na malha?

No código, o método `getProximaCelula()` na classe `Malha` decide para onde o carro vai se mover com base no tipo da célula atual

```
public Celula getProximaCelula(Celula celulaAtual) {  
    if (celulaAtual.getClassificacao().equals(ClassificacaoCelula.SAIDA))  
        return null;  
    switch (celulaAtual.getTipoCelula()) {  
  
        case TipoCelula.ESTRADA_CIMA:  
        case TipoCelula.CRUZAMENTO_CIMA:  
            return getCelulaACima(celulaAtual);  
  
        case TipoCelula.ESTRADA_DIREITA:  
        case TipoCelula.CRUZAMENTO_DIREITA:  
            return getCelulaADireita(celulaAtual);  
  
        case TipoCelula.ESTRADA_BAIXO:  
        case TipoCelula.CRUZAMENTO_BAIXO:  
            return getCelulaABaixo(celulaAtual);  
    }  
}
```

Se a célula atual for um cruzamento, as regras permitem múltiplos caminhos possíveis, e o método escolhe aleatoriamente entre as opções definidas pelo tipo de cruzamento

```
case TipoCelula.CRUZAMENTO_CIMA_E_DIREITA:  
    if (random.nextInt(2) == 0)  
        return getCelulaACima(celulaAtual);  
    else  
        return getCelulaADireita(celulaAtual);  
  
case TipoCelula.CRUZAMENTO_CIMA_E_ESQUERDA:  
    if (random.nextInt(2) == 0)  
        return getCelulaACima(celulaAtual);  
    else  
        return getCelulaAESquerda(celulaAtual);  
  
case TipoCelula.CRUZAMENTO_DIREITA_E_BAIXO:  
    if (random.nextInt(2) == 0)  
        return getCelulaADireita(celulaAtual);  
    else  
        return getCelulaABaixo(celulaAtual);  
  
case TipoCelula.CRUZAMENTO_BAIXO_E_ESQUERDA:  
    if (random.nextInt(2) == 0)  
        return getCelulaABaixo(celulaAtual);  
    else  
        return getCelulaAESquerda(celulaAtual);
```

O que acontece quando um carro chega a um cruzamento?

A classe Carro usa o método `locomoverNoCruzamento`, que primeiro ● calcula a rota completa para atravessar o cruzamento. O carro então tenta reservar todas as células dessa rota antes de começar a se mover. Se não conseguir reservar todas, ele libera qualquer célula já reservada e espera um pouco antes de tentar de novo.

```
private void locomoverNoCruzamento(Celula proximaCelula) {
    ● List<Celula> rotaCruzamento = obterRotaCruzamento(proximaCelula);
    boolean locomoveu = false;
    while (!locomoveu) {
        List<Celula> celulasReservadas = new ArrayList<>();

        for (Celula celula : rotaCruzamento) {
            if (!celula.tentarReservar()) {
                liberarCelulas(celulasReservadas);
                try {
                    sleep(100 + random.nextInt(1000));
                } catch (Exception e) {
                    System.out.println(e);
                    System.out.println(e.getMessage());
                }
                break;
            }

            celulasReservadas.add(celula);
            locomoveu = celulasReservadas.size() == rotaCruzamento.size();
        }

        andarCruzamento(rotaCruzamento);
    }
}
```


O que acontece quando um carro chega a um cruzamento?

Ele inicializa uma lista `rotaCruzamento` que armazenará as células pelas quais o carro deve passar para atravessar o cruzamento.

Aí vem o loop, enquanto a última célula adicionada na lista for cruzamento, continua buscando mais células.

Se o carro já percorreu 3 células (para não ficar em loop no cruzamento), ele procura a saída mais próxima, caso contrário, continua procurando a próxima célula.

Após calcular a próxima célula, independente de célula normal ou saída, vai adicionando na lista.

Ao final quando o laço termina, retorna a lista com a rota onde o carro deve atravessar.

```
private List<Celula> obterRotaCruzamento(Celula proximaCelula) {  
    LinkedList<Celula> rotaCruzamento = new LinkedList<Celula>();  
    rotaCruzamento.add(proximaCelula);  
  
    while (rotaCruzamento.getLast().getClassificacao().equals(ClassificacaoCelula.CRUZAMENTO)) {  
        if (rotaCruzamento.size() >= 3) {  
            proximaCelula = Malha.getInstance().getCelulaSaidaMaisProxima(proximaCelula);  
        } else {  
            proximaCelula = Malha.getInstance().getProximaCelula(proximaCelula);  
        }  
        rotaCruzamento.add(proximaCelula);  
    }  
  
    return rotaCruzamento;  
}
```



O que acontece quando um carro chega a um cruzamento?

- O carro então tenta reservar todas as células dessa rota antes de começar a se mover (Celula).
- Se não conseguir reservar todas, ele libera qualquer célula já reservada e espera um pouco antes de tentar de novo (Celula).

```
private void locomoverNoCruzamento(Celula proximaCelula) {
    List<Celula> rotaCruzamento = obterRotaCruzamento(proximaCelula);
    boolean locomoveu = false;
    while (!locomoveu) {
        List<Celula> celulasReservadas = new ArrayList<>();

        for (Celula celula : rotaCruzamento) {
            ● if (!celula.tentarReservar()) {
                ● liberarCelulas(celulasReservadas);
                try {
                    sleep(100 + random.nextInt(1000));
                } catch (Exception e) {
                    System.out.println(e);
                    System.out.println(e.getMessage());
                }
                break;
            }

            celulasReservadas.add(celula);
            locomoveu = celulasReservadas.size() == rotaCruzamento.size();
        }
    }
    andarCruzamento(rotaCruzamento);
}
```

Como o projeto garante que não haverá um "deadlock" no uso de recursos compartilhados?

Antes de um carro se mover para uma nova célula ou cruzamento, ele tenta reservar a célula usando tryLock (para monitores) ou tryAcquire (para semáforos), e define um tempo máximo de espera.

Se não conseguir acessar a célula dentro desse tempo, ele libera qualquer célula reservada e espera antes de tentar de novo.

```
for (Celula celula : rotaCruzamento) {
    if (!celula.tentarReservar()) {
        liberarCelulas(celulasReservadas);
        try {
            sleep(100 + random.nextInt(1000));
        } catch (Exception e) {
            System.out.println(e);
            System.out.println(e.getMessage());
        }
        break;
    }
}
```

```
public boolean tentarReservar() {
    if (this.carroAtual != null) {
        return false;
    }

    if (ConfiguracoesMalha.getInstance().getMecanismoExclusao() == "Semáforos") {
        return tentarReservarSemaforo();
    } else {
        return tentarReservarMonitor();
    }
}
```

```
private boolean tentarReservarMonitor() {
    try {
        return this.lock.tryLock(100, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
        System.out.println(e.getStackTrace());
        return false;
    }
}

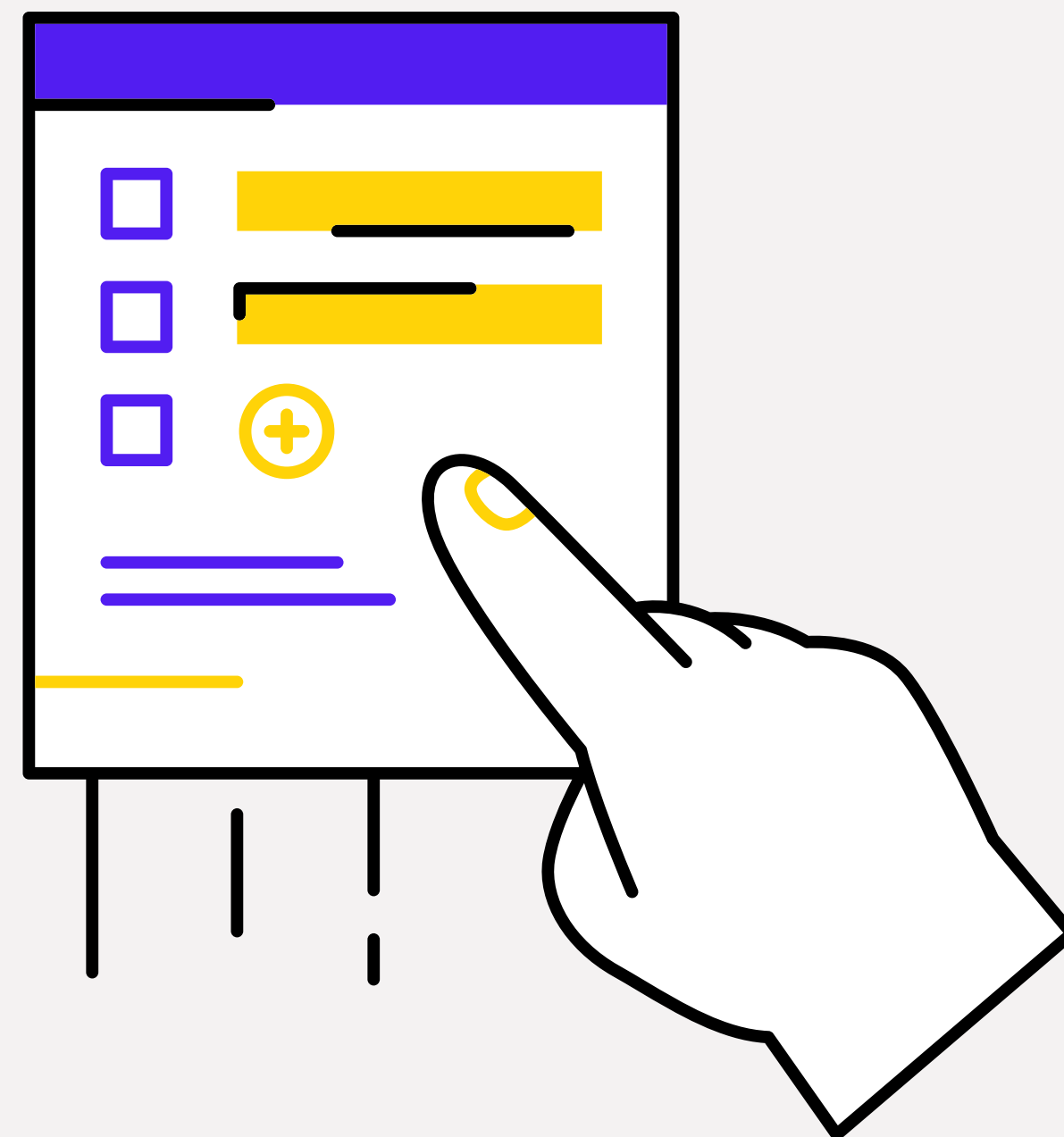
private boolean tentarReservarSemaforo() {
    try {
        return this.semaforo.tryAcquire(100, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
        System.out.println(e.getStackTrace());
        return false;
    }
}
```

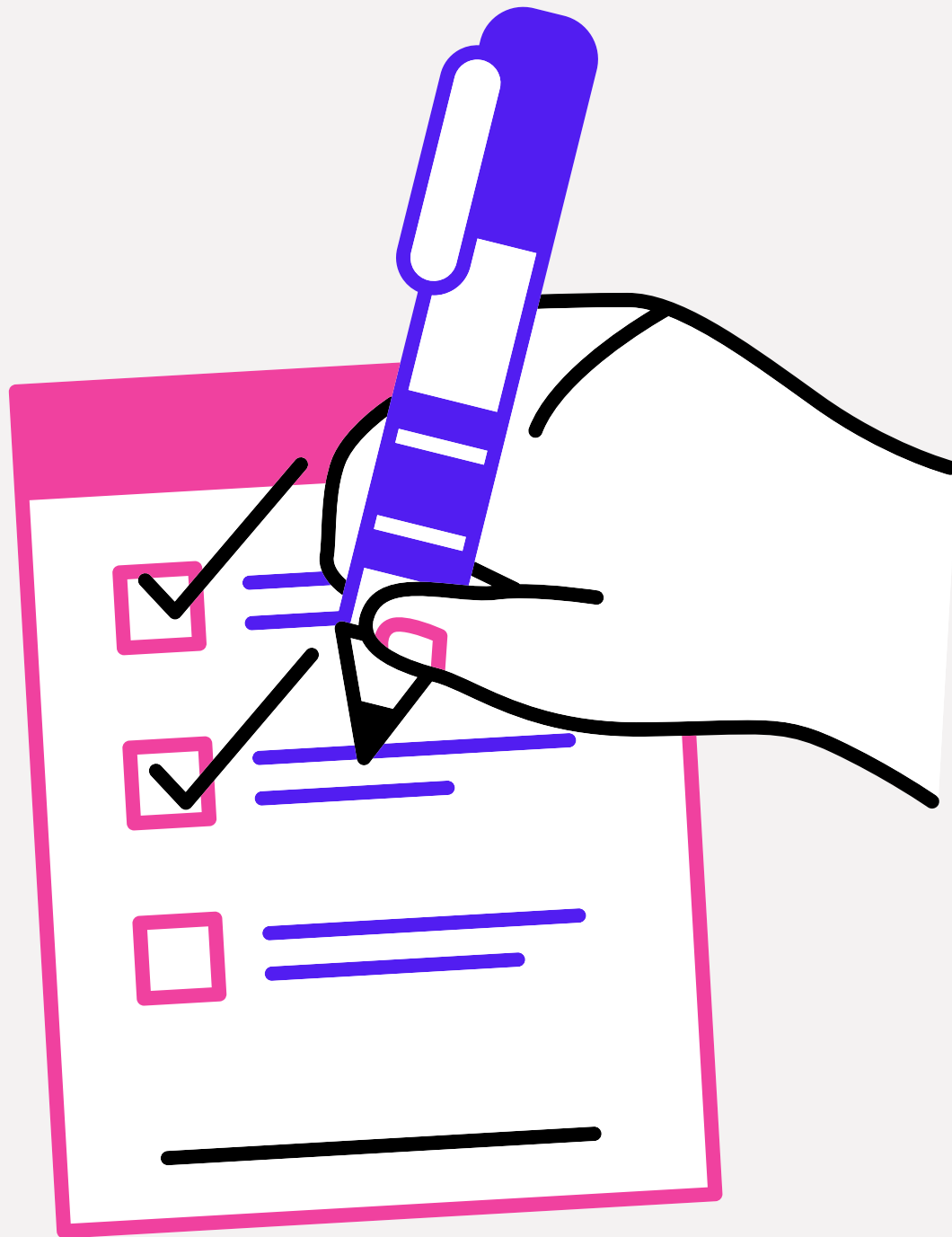
Dificuldades

01 Gestão de ícones.

02 Visualização para testes

03 Encontrar os pontos necessários para debuggar corretamente as threads





Soluções

- Em relação aos ícones foi criar uma classe que iria fazer a renderização deles, já deixando setado que o ícone seria centralizado e tudo mais, e ordenar os ícones com a numeração da malha.
- Visualização pra testes, foi colocar logs ao longo do código para conseguir acompanhar as atualizações
- Depois de longos testes, conseguimos criar cenários e conseguimos encontrar onde seriam necessários os breakpoints

The background is a solid red color. It is decorated with various white abstract shapes, including circles, ovals, and irregular blobs. Some of these shapes are filled with small white dots, while others are empty outlines. The shapes are scattered across the entire background.

Obrigada

Gabriela e Ariadne