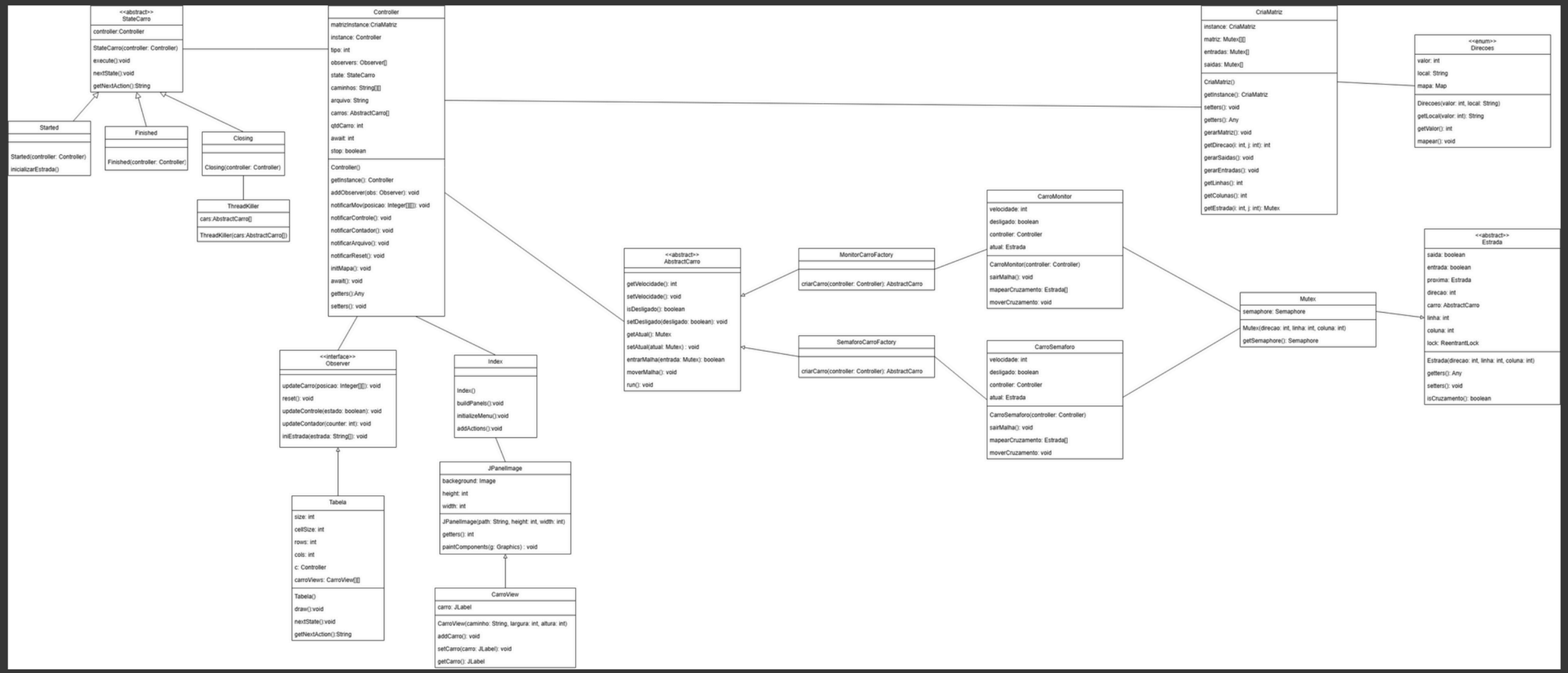
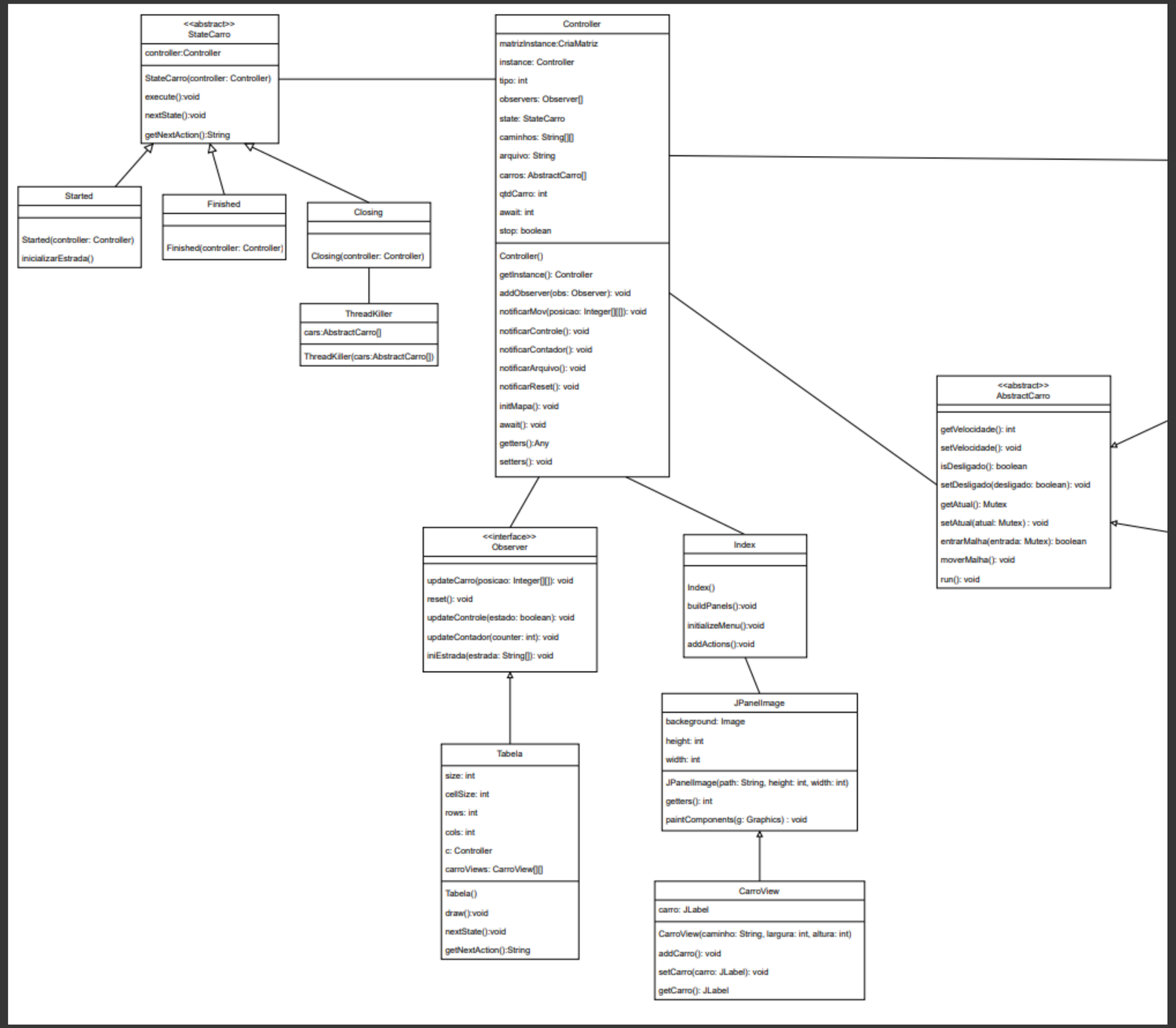
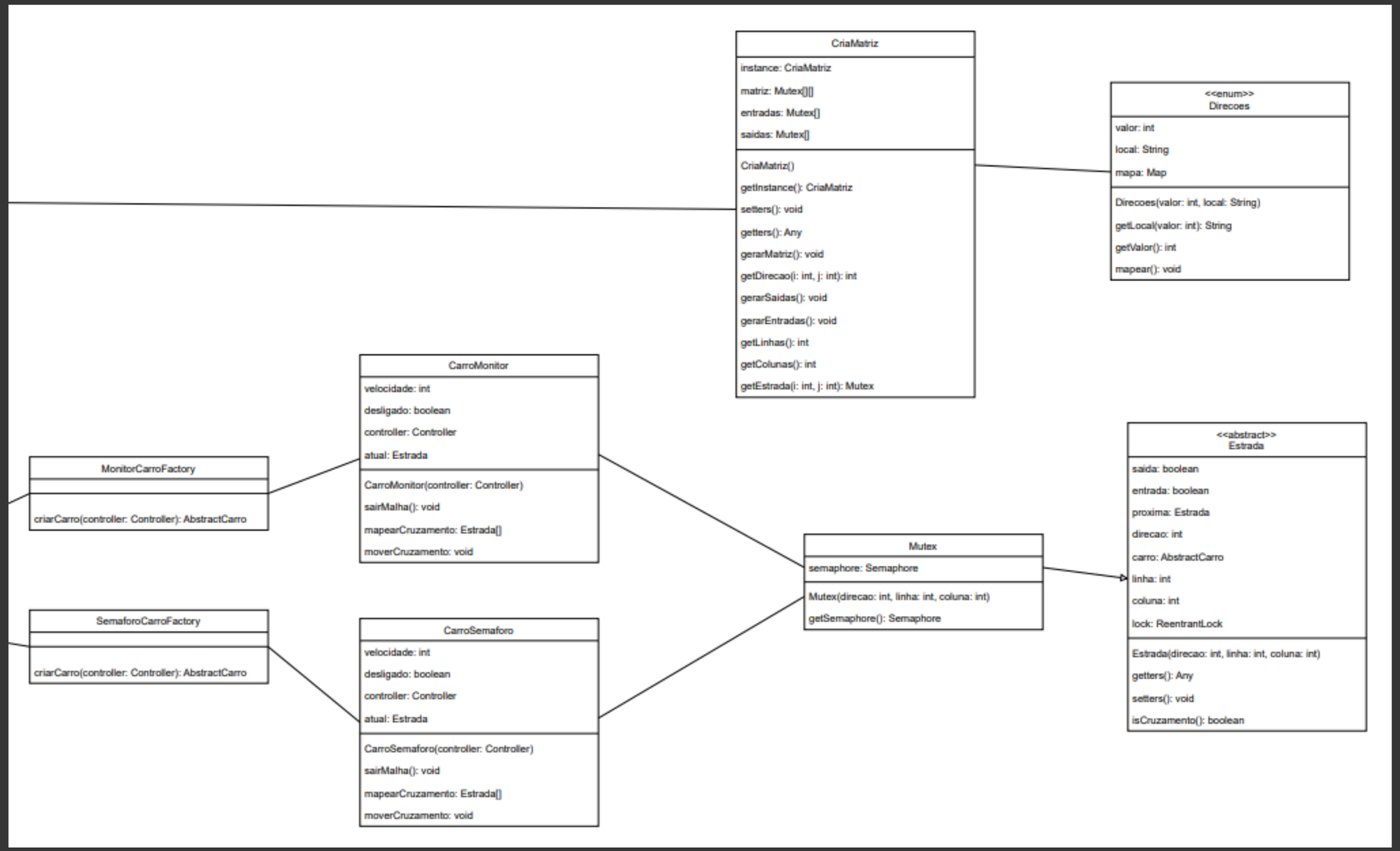
SIMULADOR DE TRÁFEGO

João Henrique de Carvalho
Lucas da Cunha Rodrigues

DIAGRAMA


<<abstract>>
StateCarro
controller:Controller
StateCarro(controller: Controller)
execute():void
nextState():void
getNextAction():String
Started
Started(controller: Controller)
inicializarEstrada()
Finished
Finished(controller: Controller)
Closing
Closing(controller: Controller)
ThreadKiller
cars:AbstractCarro[]
ThreadKiller(cars:AbstractCarro[])
Controller
matrizInstance:CriaMatriz
instance: Controller
tipo: int
observers: Observer[]
state: StateCarro
caminhos: String[][]
arquivo: String
carros: AbstractCarro[]
qtdCarro: int
await: int
stop: boolean
Controller()
getInstance(): Controller
addObserver(obs: Observer): void
notificarMov(posicao: Integer[][]): void
notificarControle(): void
notificarContador(): void
notificarArquivo(): void
notificarReset(): void
initMapa(): void
await(): void
getters():Any
setters(): void
<<interface>>
Observer
updateCarro(posicao: Integer[][]): void
reset(): void
updateControle(estado: boolean): void
updateContador(counter: int): void
iniEstrada(estrada: String[]): void
Index
Index()
buildPanels():void
initializeMenu():void
addActions():void
Tabela
size: int
cellSize: int
rows: int
cols: int
c: Controller
carroViews: CarroView[][]
Tabela()
draw():void
nextState():void
getNextAction():String
JPanelImage
background: Image
height: int
width: int
JPanelImage(path: String, height: int, width: int)
getters(): int
paintComponents(g: Graphics) : void
CarroView
carro: JLabel
CarroView(caminho: String, largura: int, altura: int)
addCarro(): void
setCarro(carro: JLabel): void
getCarro(): JLabel
<<abstract>>
AbstractCarro
getVelocidade(): int
setVelocidade(): void
isDesligado(): boolean
setDesligado(desligado: boolean): void
getAtual(): Mutex
setAtual(atual: Mutex) : void
entrarMalha(entrada: Mutex): boolean
moverMalha(): void
run(): void
MonitorCarroFactory
criarCarro(controller: Controller): AbstractCarro
SemaforoCarroFactory
criarCarro(controller: Controller): AbstractCarro
CarroMonitor
velocidade: int
desligado: boolean
controller: Controller
atual: Estrada
CarroMonitor(controller: Controller)
sairMalha(): void
mapearCruzamento: Estrada[]
moverCruzamento: void
CarroSemaforo
velocidade: int
desligado: boolean
controller: Controller
atual: Estrada
CarroSemaforo(controller: Controller)
sairMalha(): void
mapearCruzamento: Estrada[]
moverCruzamento: void
CriaMatriz
instance: CriaMatriz
matriz: Mutex[][]
entradas: Mutex[]
saidas: Mutex[]
CriaMatriz()
getInstance(): CriaMatriz
setters(): void
getters(): Any
gerarMatriz(): void
getDirecao(i: int, j: int): int
gerarSaidas(): void
gerarEntradas(): void
getLinhas(): int
getColunas(): int
getEstrada(i: int, j: int): Mutex
<<enum>>
Direcoes
valor: int
local: String
mapa: Map
Direcoes(valor: int, local: String)
getLocal(valor: int): String
getValor(): int
mapear(): void
Mutex
semaphore: Semaphore
Mutex(direcao: int, linha: int, coluna: int)
getSemaphore(): Semaphore
<<abstract>>
Estrada
saida: boolean
entrada: boolean
proxima: Estrada
direcao: int
carro: AbstractCarro
linha: int
coluna: int
lock: ReentrantLock
Estrada(direcao: int, linha: int, coluna: int)
getters(): Any
setters(): void
isCruzamento(): boolean


<<abstract>>
StateCarro
controller:Controller
StateCarro(controller: Controller)
execute():void
nextState():void
getNextAction():String
Started
Started(controller: Controller)
inicializarEstrada()
Finished
Finished(controller: Controller)
Closing
Closing(controller: Controller)
ThreadKiller
cars:AbstractCarro[]
ThreadKiller(cars:AbstractCarro[])
Controller
matrizInstance:CriaMatriz
instance: Controller
tipo: int
observers: Observer[]
state: StateCarro
caminhos: String[][]
arquivo: String
carros: AbstractCarro[]
qtdCarro: int
await: int
stop: boolean
Controller()
getInstance(): Controller
addObserver(obs: Observer): void
notificarMov(posicao: Integer[][]): void
notificarControle(): void
notificarContador(): void
notificarArquivo(): void
notificarReset(): void
initMapa(): void
await(): void
getters():Any
setters(): void
<<abstract>>
AbstractCarro
getVelocidade(): int
setVelocidade(): void
isDesligado(): boolean
setDesligado(desligado: boolean): void
getAtual(): Mutex
setAtual(atual: Mutex) : void
entrarMalha(entrada: Mutex): boolean
moverMalha(): void
run(): void
<<interface>>
Observer
updateCarro(posicao: Integer[][]): void
reset(): void
updateControle(estado: boolean): void
updateContador(counter: int): void
iniEstrada(estrada: String[]): void
Index
Index()
buildPanels():void
initializeMenu():void
addActions():void
Tabela
size: int
cellSize: int
rows: int
cols: int
c: Controller
carroViews: CarroView[][]
Tabela()
draw():void
nextState():void
getNextAction():String
JPanelImage
background: Image
height: int
width: int
JPanelImage(path: String, height: int, width: int)
getters(): int
paintComponents(g: Graphics) : void
CarroView
carro: JLabel
CarroView(caminho: String, largura: int, altura: int)
addCarro(): void
setCarro(carro: JLabel): void
getCarro(): JLabel


CriaMatriz
instance: CriaMatriz
matriz: Mutex[][]
entradas: Mutex[]
saidas: Mutex[]
CriaMatriz()
getInstance(): CriaMatriz
setters(): void
getters(): Any
gerarMatriz(): void
getDirecao(i: int, j: int): int
gerarSaidas(): void
gerarEntradas(): void
getLinhas(): int
getColunas(): int
getEstrada(i: int, j: int): Mutex
<<enum>>
Direcoes
valor: int
local: String
mapa: Map
Direcoes(valor: int, local: String)
getLocal(valor: int): String
getValor(): int
mapear(): void
CarroMonitor
velocidade: int
desligado: boolean
controller: Controller
atual: Estrada
CarroMonitor(controller: Controller)
sairMalha(): void
mapearCruzamento: Estrada[]
moverCruzamento: void
MonitorCarroFactory
criarCarro(controller: Controller): AbstractCarro
SemaforoCarroFactory
criarCarro(controller: Controller): AbstractCarro
CarroSemaforo
velocidade: int
desligado: boolean
controller: Controller
atual: Estrada
CarroSemaforo(controller: Controller)
sairMalha(): void
mapearCruzamento: Estrada[]
moverCruzamento: void
Mutex
semaphore: Semaphore
Mutex(direcao: int, linha: int, coluna: int)
getSemaphore(): Semaphore
<<abstract>>
Estrada
saida: boolean
entrada: boolean
proxima: Estrada
direcao: int
carro: AbstractCarro
linha: int
coluna: int
lock: ReentrantLock
Estrada(direcao: int, linha: int, coluna: int)
getters(): Any
setters(): void
isCruzamento(): boolean

REGIÕES CRÍTICAS

MONITOR

```java
private void moverCruzamento() {  1 usage
    List<Estrada> caminhoA = null;
    try {
        caminhoA = mapearCruzamento();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    if (caminhoA != null && !caminhoA.isEmpty()) {
        boolean atravessou = false;
        Random rand = new Random();
        while (!atravessou) {
            List<Estrada> acquired = new ArrayList<>();
            boolean conseguiuTodos = true;
            try {
                // Tenta adquirir todos os locks do caminho
                for (Estrada pos : caminhoA) {
                    if (!pos.getLock().tryLock( timeout: 100, java.util.concurrent.TimeUnit.MILLISECONDS)) {
                        conseguiuTodos = false;
                        break;
                    } else {
                        acquired.add(pos);
                    }
                }
                if (conseguiuTodos) {
                    // Marca todas as posições como ocupadas
                    for (Estrada pos : caminhoA) {
                        pos.setCarro(this);
                    }
                    // Faz a travessia, SEM liberar lock/carro da posição anterior a cada passo
                    Estrada anterior = atual;
                    for (int i = 0; i < caminhoA.size(); i++) {
```

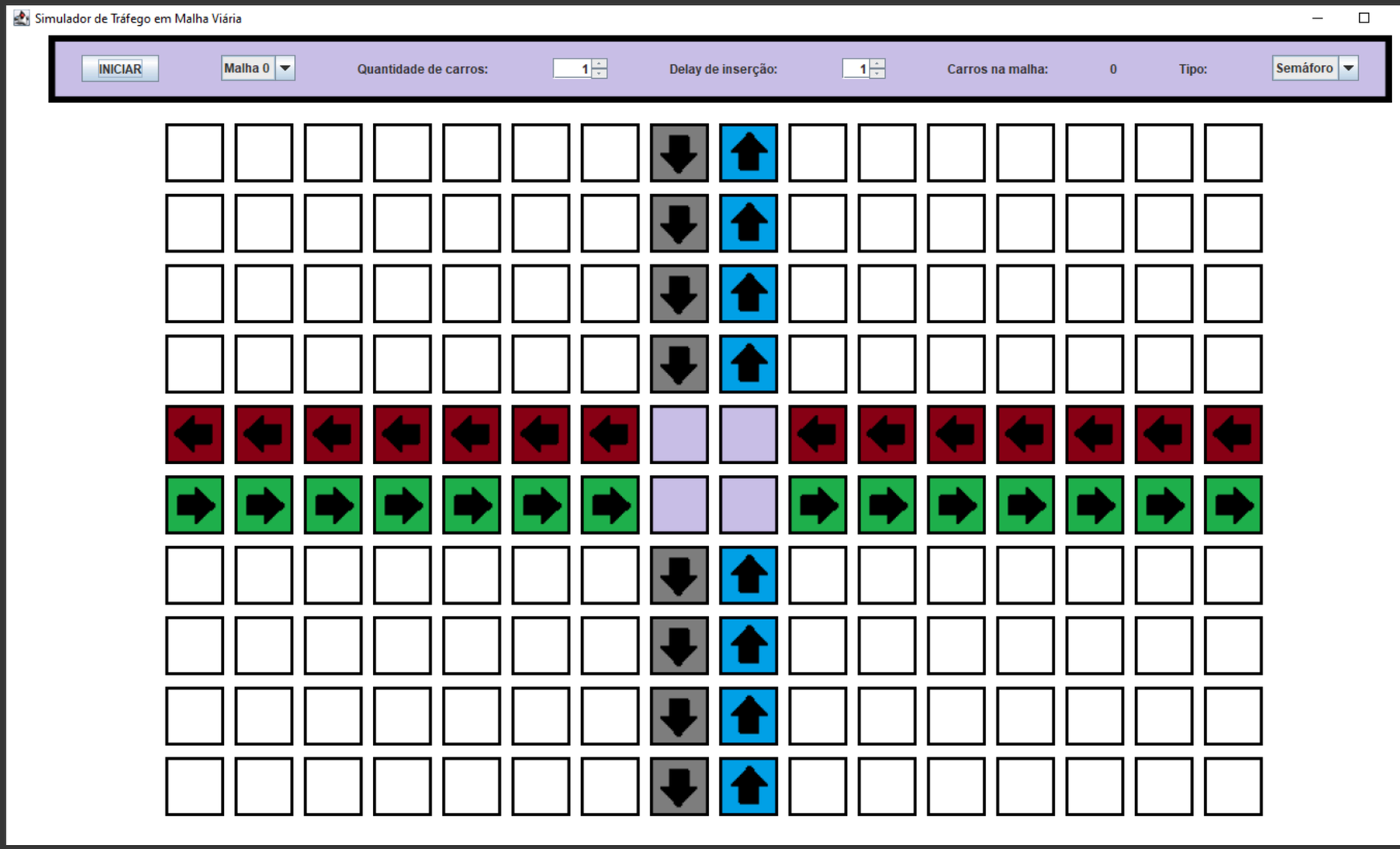
```
    // Faz a travessia, SEM liberar lock/carro da posição anterior a cada passo
    Estrada anterior = atual;
    for (int i = 0; i < caminhoA.size(); i++) {
        Estrada caminho = caminhoA.get(i);
        Integer[][] posicoes = {
                { anterior.getLinha(), anterior.getColuna() },
                { caminho.getLinha(), caminho.getColuna() }
        };
        controller.notificarMov(posicoes);
        atual = caminho;
        if (i < caminhoA.size() - 1) {
            try {
                Thread.sleep(velocidade);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        anterior = caminho;
    }
    // Após sair do cruzamento, libera todos os locks e campos carro
    for (Estrada pos : caminhoA) {
        pos.setCarro(null);
        if (pos.getLock().isHeldByCurrentThread()) {
            pos.getLock().unlock();
        }
    }
    atravessou = true;
} else {
    // Não conseguiu todos, libera os já adquiridos
    for (Estrada pos : acquired) {
```

```
            atravessou = true;
        } else {
            // Não conseguiu todos, libera os já adquiridos
            for (Estrada pos : acquired) {
                if (pos.getLock().isHeldByCurrentThread()) {
                    pos.getLock().unlock();
                }
            }
            try {
                Thread.sleep( millis: 100 + rand.nextInt( bound: 400));
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {
        // Garante que todos os locks sejam liberados em caso de erro
        for (Estrada pos : acquired) {
            if (pos.getLock().isHeldByCurrentThread()) {
                pos.getLock().unlock();
            }
```

SEMÁFORO

```
private void moverCruzamento(){  1 usage
    List<Mutex> caminhoA = null;
    try {
        caminhoA=mapearCruzamento();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    if(caminhoA!=null)
    {
        for(int i=0;i<caminhoA.size();i++)
        {
            Mutex caminho = caminhoA.get(i);
            Integer[][] posicoes =
                    {
                            {atual.getLinha(), atual.getColuna()},
                            {caminho.getLinha(),caminho.getColuna()}
                    };
            controller.notificarMov(posicoes);
            atual.getSemaphore().release();
            atual.setCarro(null);
            atual=caminho;
            if(i<caminhoA.size()-1)
            {
                try {
                    Thread.currentThread().sleep(velocidade);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
}
```

```
    }
    for(Mutex caminho1 : caminho)
    {
        List<Semaphore> acquired = new ArrayList<>();
        acquired.add(caminho1.getSemaphore());
        boolean b = caminho1.getSemaphore().tryAcquire( timeout: 100, TimeUnit.MILLISECONDS);
        if(!b)
        {
            escolhido.getSemaphore().release();
            for(Semaphore semaphore : acquired)
                semaphore.release();
            Thread.currentThread().sleep(velocidade);
            return null;
        }
    }
    caminho.add(escolhido);
    return caminho;
}
Thread.currentThread().sleep(velocidade);
return null;
```


Simulador de Tráfego em Malha Viária
INICIAR
Malha 0
Quantidade de carros:
1
Delay de inserção:
1
Carros na malha:
0
Tipo:
Semáforo

DESAFIOS

- Implementar um sistema que acomodasse o uso de monitores e de semáforos;
- Garantir que não ocorram deadlocks;
- Garantir que os carros percorram as malhas corretamente;
- Garantir que carros só entrem em cruzamentos se todos os espaços estiverem livres, evitando deadlocks