



DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Licenciatura em Engenharia Informática e Multimédia

Trabalho Prático 1

Fundamentos de Sistemas Operativos

Alunos:

A51417 - David Santos
A52483 - Bernardo Aguiar

Docentes:

Prof. Carlos Gonçalves

Outubro de 2025

Conteúdo

1	Introdução	1
2	Estrutura	1
2.1	UML	1
3	Classes e Métodos	1
3.1	Classe: GUI	2
3.2	Classe: RobotController	2
3.2.1	Método: updateData	2
3.2.2	Métodos: Movimentos e Ações	2
3.2.3	Métodos: Movimentos Aleatórios	3
3.2.4	Controlador do Buffer	4
3.3	Classe: Data	6
3.4	Classe: RandomMovements	6
3.4.1	Método: setWorking	8
3.4.2	Funcionamento Básico	8
3.5	Classe: Buffer	8
3.5.1	Método: cleanBuffer	10
3.6	Classe: Movement	10
3.6.1	Construtores	10
3.6.2	Método: doMovement	11
3.6.3	Métodos: Delays	11
3.7	Classe: BufferManager	11
3.8	Classe: AvoidObstacles	12
3.9	Classe: RobotAccessManager	14
4	Conclusões	17

Lista de Figuras

1	UML	1
2	Diagrama de Estados: Buffer Controller	4
3	Diagrama de Estados: Movimentos Aleatórios	9
4	Adição de Gestor de Acesso ao Robô	15

Listings

1 Introdução

O principal objetivo do projeto consiste no desenvolvimento de uma aplicação em Java para o controle de um robô, com particular foco na programação de multi-tarefas. Para tal, a aplicação é composta por um modelo MVC.

A GUI, utilizando o editor *WindowBuilder*, permite ao utilizador interagir diretamente com o robô, enviando comandos como *Forward*, *Backwards*, *Stop*, *Left* e *Right*, além de ativar ou desativar a tarefa de movimentos aleatórios. Adicionalmente, a interface inclui uma consola para exibir as ações realizadas pelo robô.

2 Estrutura

2.1 UML

Esta figura representa o diagrama UML do programa e é crucial para o melhor entendimento sobre o funcionamento do programa e das relações entre as classes.

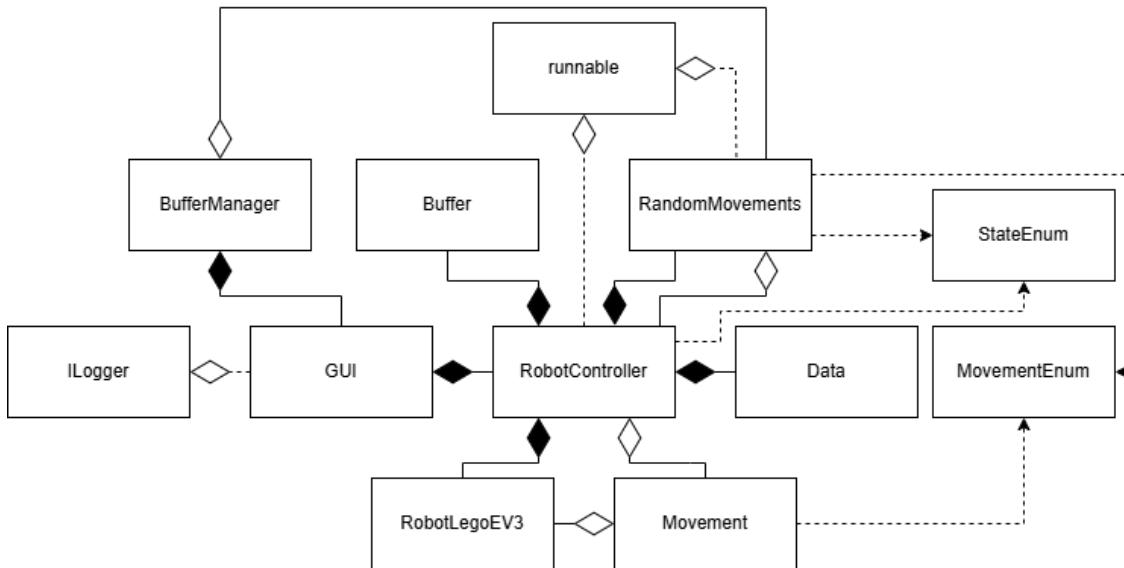


Figura 1: UML

3 Classes e Métodos

Para o desenvolvimento foi usado como referência principal os conceitos apreendidos nas aulas e mencionados nos slides da [1]disciplina.

3.1 Classe: GUI

A classe GUI age como a parte do *Viewer* do modelo MVC, centralizando a lógica da interface. Quando o utilizador interage com um componente, o método *updateData()* é chamado para sincronizar os valores da interface com a classe *RobotController*.

```
1 public void updateData() {  
2     robotController.updateData(textRadius.getText(),  
3         textAngle.getText(), textDistance.getText(),  
4         textRobotName.getText(),  
5         spinnerNumber.getValue().toString());  
6 }
```

3.2 Classe: RobotController

A classe *RobotController* age como a parte do *Controller* do modelo MVC funcionando como a ponte essencial entre as classes e o robô físico. A sua principal função é traduzir as ações do utilizador em comandos específicos que controlam o robô.

3.2.1 Método: updateData

Este método serve como uma ponte dos valores introduzidos na GUI para a classe Data para uma melhor organização e separação do código.

```
1 public void updateData(String radius, String angle, String  
2     distance, String name, String actionNumber) {  
3     data.setRadius(Integer.parseInt(radius));  
4     data.setAngle(Integer.parseInt(angle));  
5     data.setDistance(Integer.parseInt(distance));  
6     data.setName(name);  
7     data.setActionNumber(Integer.parseInt(actionNumber));  
8 }
```

3.2.2 Métodos: Movimentos e Ações

Métodos simples que usam os valores da classe Data para introduzir os movimentos no buffer e para ligar o robô. Exemplos:

```
1 public void turnOnRobot() {  
2     robot.OpenEV3(data.getName());  
3 }  
4  
5 public synchronized void moveBackwards() {  
6     buffer.put(new Movement(robot, logger,  
7         MovementEnum.BACKWARDS, -data.getDistance()));  
8     notify();  
9 }
```

```

8 }
9
10 public synchronized void moveRightCurve() {
11     buffer.put(new Movement(robot, logger,
12         MovementEnum.RIGHT, data.getRadius(),
13         data.getAngle()));
14     notify();
15 }
16
17 public synchronized void putBuffer(Movement movement) {
18     buffer.put(movement);
19     notify();
20 }
```

3.2.3 Métodos: Movimentos Aleatórios

Estes métodos são responsáveis por iniciar e parar a máquina de estados responsável pela criação dos Movimentos Aleatórios do robô.

```

1 public void startRandomMovements() {
2     randomMovements.setActionNumber(data.getActionNumber());
3     randomMovements.setWorking(true);
4 }
5
6 public void stopRandomMovements() {
7     randomMovements.setWorking(false);
8 }
```

3.2.4 Controlador do Buffer

Esta máquina de estados é responsável pela execução dos movimentos dentro do buffer. Espera que o buffer tenha movimentos para executar e, quando tiver, executa-os e espera que o mesmo seja executado antes de voltar a executar outro, ou voltar para o seu estado *IDLE*.

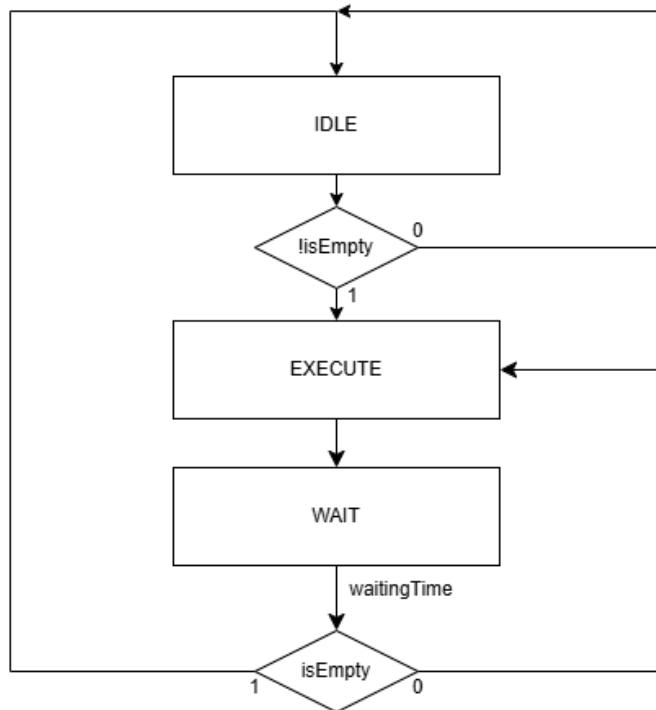


Figura 2: Diagrama de Estados: Buffer Controller

```
1 while (true) {
2     switch (bufferState) {
3         case IDLE:
4             synchronized (this) {
5                 try {
6                     this.wait();
7                 } catch (InterruptedException e) {
8                     e.printStackTrace();
9                     Thread.currentThread().interrupt();
10                }
11                if (!buffer.isEmpty())
12                    bufferState = StateEnum.EXECUTE;
13            break;
14        case EXECUTE:
15            movement = buffer.get();
16            movement.doMovement();
17            waitingTime = movement.getTime();
18            bufferState = StateEnum.WAIT;
19            break;
20        case WAIT:
```

```
21     try {
22         Thread.sleep(waitingTime);
23     } catch (InterruptedException e) {
24         e.printStackTrace();
25     }
26     if (buffer.isEmpty()) {
27         bufferState = StateEnum.IDLE;
28         stopMovementSync();
29     } else {
30         bufferState = StateEnum.EXECUTE;
31     }
32     break;
33 default:
34     break;
35 }
36 }
```

3.3 Classe: Data

A classe Data serve exclusivamente para armazenar e gerir dados. Esta agrupa todas as informações provenientes da GUI, como a distância, o ângulo, o raio e o nome, e o número de ações aleatórias, num único objeto.

```
1 public Data(int radius, int angle, int distance, int
2   actionNumber, String name) {
3   this.distance = distance;
4   this.angle = angle;
5   this.radius = radius;
6   this.actionNumber = actionNumber;
7   this.name = name;
8 }
```

3.4 Classe: RandomMovements

Esta classe implementa uma máquina de estados. Esta é iniciada em uma nova *Thread* que tem como objetivo a criação aleatória de movimentos do robô, que depois são enviados para o *buffer* através do *RobotController*. Ao colocar em uma máquina de estados, permite-se controlar a criação de novos movimentos aleatórios com maior facilidade.

```
1 public void run() {
2     while (true) {
3         switch (STATE) {
4             case IDLE:
5                 synchronized (this) {
6                     try {
7                         this.wait();
8                     } catch (InterruptedException e) {
9                         e.printStackTrace();
10                        Thread.currentThread().interrupt();
11                     }
12                     if (working)
13                         STATE = StateEnum.GENERATE;
14                     break;
15
16             case GENERATE:
17                 movementList = new Movement[actionNumber * 2];
18                 waitingTime = 0;
19
20                 for (int i = 0; i < this.actionNumber; i++) {
21                     MovementEnum[] movement =
22                         MovementEnum.values();
23                     int direction =
24                         random.nextInt(movement.length);
```

```

24         if (lastDirection == movement[direction]) {
25             i--;
26             continue;
27         }
28
29         if (movement[direction] ==
30             MovementEnum.FORWARD)
31             movementList[i * 2] = (new
32                 Movement(robot, this.logger,
33                     movement[direction],
34                     random.nextInt(40) + 10));
35         else if (movement[direction] ==
36             MovementEnum.RIGHT || movement[direction]
37             == MovementEnum.LEFT)
38             movementList[i * 2] = (new
39                 Movement(robot, this.logger,
40                     movement[direction],
41                     random.nextInt(20) + 10,
42                     random.nextInt(70) + 20));
43         else {
44             i--;
45             continue;
46         }
47         waitingTime += movementList[i * 2].getTime();
48
49         movementList[i * 2 + 1] = (new
50             Movement(robot, this.logger,
51                 MovementEnum.STOP));
52         waitingTime += movementList[i * 2 +
53             1].getTime();
54
55         lastDirection = movement[direction];
56     }
57
58     STATE = StateEnum.SEND;
59     break;
60
61 case SEND:
62     bufferManager.acquire();
63     for (int i = 0; i < this.actionNumber * 2; i++) {
64         robotController.putBuffer(movementList[i]);
65     }
66     bufferManager.release();
67     STATE = StateEnum.WAIT;
68     break;
69
70 case WAIT:
71     try {
72         Thread.sleep(waitingTime + 4000);

```

```

62         } catch (InterruptedException e) {
63             e.printStackTrace();
64         }
65         if (this.working)
66             STATE = StateEnum.GENERATE;
67         else
68             STATE = StateEnum.IDLE;
69         break;
70     default:
71         break;
72     }
73 }
74 }
```

3.4.1 Método: setWorking

Este método sempre que é chamado muda o estado da máquina e acorda a Thread para que esta confirme o seu estado.

```

1 public synchronized void setWorking(boolean working) {
2     this.working = working;
3     this.notify();
4 }
```

3.4.2 Funcionamento Básico

Os movimentos aleatórios são gerados em grupos definidos por um valor inserido na *GUI*, respeitando os parâmetros do enunciado, e o programa impede a criação de dois movimentos consecutivos na mesma direção. A geração de novos movimentos aleatórios é mantida até que o botão da interface seja desativado. Após a adição de um movimento gerado ao *buffer*, o comando de parar é enviado, e entre a geração de cada grupo é aplicado um intervalo de 4 segundos, em ambos os casos, o objetivo é otimizar a visualização e separação das ações no funcionamento do programa.

Esta operação é gerida por uma máquina de estados composta por quatro estados: *IDLE*, *GENERATE*, *SEND* e *WAIT*. No estado *IDLE*, a máquina apenas verifica se foi ligada para iniciar a geração e o envio dos valores aleatórios. Assim que é ativada, entra no estado *GENERATE*, onde gera o grupo de movimentos aleatórios igual ao número introduzido pelo utilizador, de seguida, entra no estado *SEND* e envia para o *buffer*, seguindo imediatamente para o estado *WAIT*. Neste último estado, a máquina espera o tempo necessário que foi alocado na variável *waitingTime* e depois certifica-se se a máquina ainda está ligada. Se estiver ligada, retorna ao estado *GENERATE*, repetindo o ciclo até que seja verificado que não está mais ligada, o que causa o retorno ao estado *IDLE*.

3.5 Classe: Buffer

O *buffer* feito com monitores, utiliza um *ArrayList* (em oposição a um *Array* de capacidade limitada) para prevenir erros como a perda de dados ou a paragem de *Threads*,

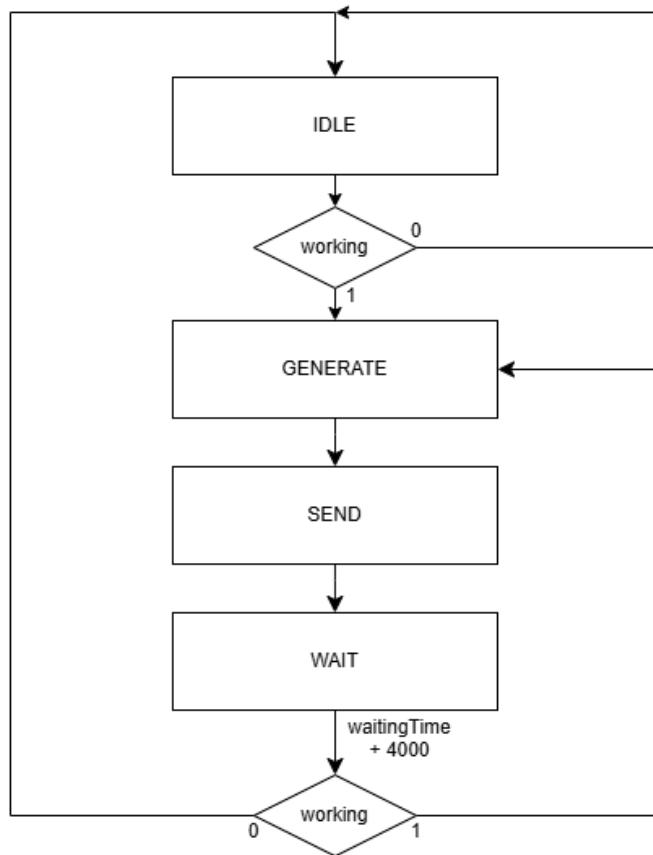


Figura 3: Diagrama de Estados: Movimentos Aleatórios

garantindo assim um uso mais eficiente dos recursos da máquina sendo a solução mais simples, eficiente e intuitiva.

```

1 public class Buffer {
2
3     private ArrayList<Movement> buffer;
4
5     public Buffer() {
6         this.buffer = new ArrayList<>();
7     }
8
9     public synchronized void put(Movement movement) {
10        buffer.add(movement);
11        notifyAll();
12    }
13
14    public synchronized void higherPriorityPut(Movement
15        movement) {
16        buffer.add(0, movement);
17        notifyAll();
18    }
19
20    public synchronized Movement get() {

```

```

20     while (isEmpty()) {
21         try {
22             this.wait();
23         } catch (InterruptedException e) {
24             e.printStackTrace();
25         }
26     }
27
28     return buffer.remove(0);
29 }
30
31     public boolean isEmpty() {
32         return buffer.isEmpty();
33     }
34 }
```

3.5.1 Método: cleanBuffer

Este método é usado para reiniciar o buffer quando o comando de parar o robô é chamado.

```

1     public void clearBuffer() {
2         buffer.clear();
3     }
```

3.6 Classe: Movement

3.6.1 Construtores

Esta classe possui dois construtores diferentes para os dois tipos de movimentos do robô.

```

1 public Movement(RobotLegoEV3 robot, ILogger logger,
2     MovementEnum movement, int distance) {
3     this.logger = logger;
4     this.movement = movement;
5     this.distance = distance;
6     this.robot = robot;
7 }
8
9 public Movement(RobotLegoEV3 robot, ILogger logger,
10    MovementEnum movement, int radius, int angle) {
11    this.logger = logger;
12    this.movement = movement;
13    this.radius = radius;
14    this.angle = angle;
15    this.robot = robot;
16 }
```

3.6.2 Método: doMovement

Este método é usado para executar o movimento do robô.

```
1 public void doMovement() {
2     switch (this.movement) {
3         case FORWARD:
4             robot.Reta(this.distance);
5             //log
6             break;
7         case BACKWARDS:
8             robot.Reta(-this.distance);
9             //log
10            break;
11        case RIGHT:
12            robot.CurvarDireita(this.radius, this.angle);
13            //log
14            break;
15        case LEFT:
16            robot.CurvarEsquerda(this.radius, this.angle);
17            //log
18            break;
19    }
20 }
```

3.6.3 Métodos: Delays

Esta classe ainda possuí dois métodos que se destinam a calcular o tempo de atraso (em milissegundos) que o robô deve esperar para completar um movimento, seja este em linha reta ou em curva.

```
1 public int getDelayStraightLine() {
2     return (int) ((distance / 0.02) + 100);
3 }
4
5 public int getDelayCurve() {
6     return (int) (((Math.toRadians(this.angle) * this.radius)
7 / 0.02) + 100);
8 }
```

3.7 Classe: BufferManager

A classe *BufferManager* implementa um mecanismo simples de controlo de acesso concorrente a um recurso compartilhado. Por meio de monitores com os métodos *synchronized*, *wait()* e *notifyAll()*, ela garante que apenas uma thread utiliza o recurso por vez. A variável booleana *inUse* indica se o recurso está ocupado, enquanto os métodos *entrar()* e *sair()* controlam, respetivamente, a entrada e a libertação do uso, evitando conflitos e garantindo a exclusão mútua entre threads.

```

1 public class BufferManager {
2     private boolean inUse = false;
3
4     public BufferManager() {
5
6
7         public synchronized void entrar() {
8             while (inUse) {
9                 try {
10                     wait();
11                 } catch (InterruptedException e) {
12                     e.printStackTrace();
13                     Thread.currentThread().interrupt();
14                 }
15             }
16             inUse = true;
17         }
18
19         public synchronized void sair() {
20             inUse = false;
21             notifyAll();
22         }
23     }

```

3.8 Classe: AvoidObstacles

A classe *AvoidObstacle* é uma componente crucial do sistema de controlo do robô. Possui uma máquina de estados que de 50 em 50 milissegundos verifica o estado do sensor do robô e, se for necessário agir, envia 3 movimentos prioritários para o buffer, evitando os obstáculos com sucesso.

```

1 public void run() {
2     while(true) {
3         switch(STATE) {
4             case IDLE:
5                 synchronized(this) {
6                     try {
7                         Thread.sleep(50);
8                     } catch (InterruptedException e) {
9                         e.printStackTrace();
10                    }
11
12                 /* MODO ROBO */
13                 if(robotController.robotOn &&
14                     robotController.obstacleFound())
15                     STATE = StateEnum.SEND;

```

```

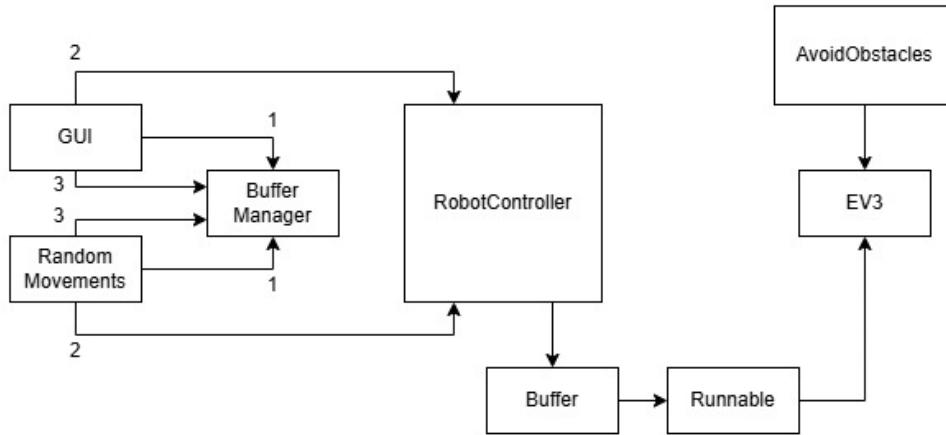
16         /* MODO TESTE
17         if(robotController.obstacleFound())
18             STATE = StateEnum.GENERATE;
19         */
20
21     }
22     break;
23 case GENERATE:
24     waitingTime = 0;
25
26     movementList[0] = new Movement(this.robot,
27         this.logger, MovementEnum.STOP);
27     movementList[1] = new Movement(this.robot,
28         this.logger, MovementEnum.BACKWARDS, 20);
29
30     while(true) {
31         MovementEnum[] movement =
32             MovementEnum.values();
33         int direction =
34             random.nextInt(movement.length);
35
36         if(movement[direction] ==
37             MovementEnum.LEFT ||
38             movement[direction] ==
39             MovementEnum.RIGHT) {
40             movementList[2] = new
41                 Movement(this.robot, this.logger,
42                     movement[direction], 0, 90);
43             break;
44         }
45     }
46
47     for(Movement movement : movementList)
48         waitingTime += movement.getTime();
49
50 case SEND:
51     bufferManager.acquire();
52     for(int i = MOVEMENT_NUMBER - 1; i >= 0; i--)
53     {
54         robotController.
55         putBufferHigherPriority(movementList[i]);
56     }
57     bufferManager.release();
58     STATE = StateEnum.WAIT;
59     break;
60 case WAIT:
61     try {
62         Thread.sleep(waitingTime);
63     } catch (InterruptedException e) {

```

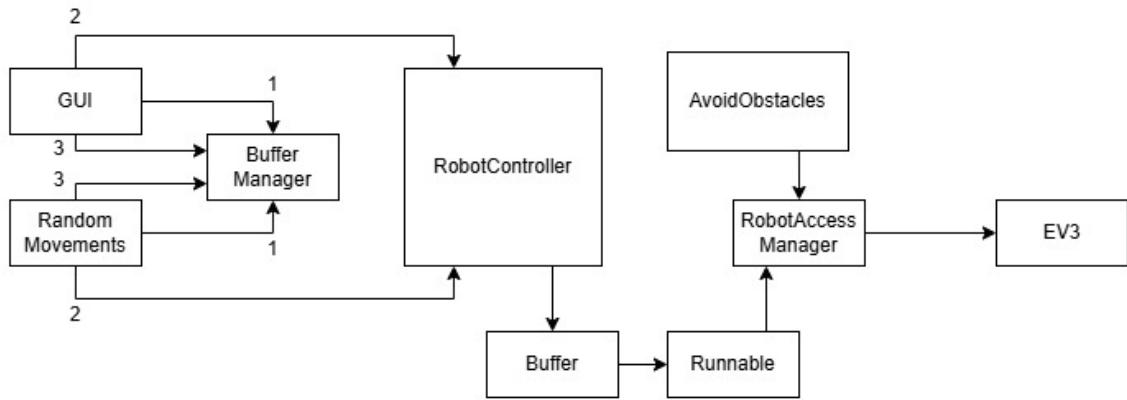
```
55             e.printStackTrace();
56         }
57
58         STATE = StateEnum.IDLE;
59
60         break;
61     default:
62         break;
63     }
64 }
65 }
```

3.9 Classe: RobotAccessManager

A classe *RobotAccessManager* garante que o acesso ao robô é feito em exclusão mútua. Isto, porque com a adição da leitura do sensor que é feita pelo mesmo canal de comunicação de envio de movimentos.



- 1 - Entrar
 2 - Enviar Comandos
 3 - Sair



- 1 - Entrar
 2 - Enviar Comandos
 3 - Sair

Figura 4: Adição de Gestor de Acesso ao Robô

A adição do gestor de acesso garante que as classes *AvoidObstacles* e *RobotController* (através da máquina de estados que retira movimentos do *Buffer*), acedem uma de cada vez.

```

1 public class RobotAccessManager {
2
3     private boolean inUse = false;
4
5     public RobotAccessManager() {
6
7     }
8
9     public synchronized void acquire() {
10        while (inUse) {
11            try {
  
```

```
12         wait();
13     } catch (InterruptedException e) {
14         e.printStackTrace();
15         Thread.currentThread().interrupt();
16     }
17 }
18     inUse = true;
19 }
20
21 public synchronized void release() {
22     inUse = false;
23     notify();
24 }
25
26 }
```

4 Conclusões

Com a realização deste projeto, foi possível consolidar e aprofundar os conceitos básicos da programação Java, com uma ênfase prática e crucial no desenvolvimento de um sistema para controlo de robótica.

O principal foco de aprendizagem residiu na implementação de programação concorrente por meio de *Threads*. Esta abordagem demonstrou-se essencial para a gestão independente e eficiente de diferentes processos do sistema, como a geração de comandos e a sua execução. Especificamente, o uso de *Threads* e *Buffers* foi fundamental para criar um mecanismo de comunicação robusto, garantindo que os comandos fossem transmitidos de forma segura e síncrona, prevenindo falhas de dados ou colisões.

Em suma, este trabalho não apenas permitiu a aprendizagem teórica de estruturas de dados e concorrência em Java, mas também ofereceu a experiência prática de aplicar estes conhecimentos para resolver um desafio de engenharia real: o controlo dinâmico e fiável de um robô. O resultado é uma solução que demonstra o poder e a necessidade da programação *multithread* para sistemas que requerem alta responsividade e coordenação de tarefas.

Referências

- [1] Jorge Pais. Folhas de fso. Slides da disciplina, 2025.