instituto
superior de
engenharia
de lisboa

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

Lista de Figuras

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 Classes e Métodos

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

2.1 Classe: GUI

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

```
public void updateData() {
    robotController.updateData(textRadius.getText(),
       textAngle.getText(), textDistance.getText(),
       textRobotName.getText(),
       spinnerNumber.getValue().toString());
}
```

2.2 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.

2.2.1 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.

```
public void updateData(String radius, String angle, String
   distance, String name, String actionNumber) {
    data.setRadius(Integer.parseInt(radius));
    data.setAngle(Integer.parseInt(angle));
```

```
    data.setDistance(Integer.parseInt(distance));
    data.setName(name);
    data.setActionNumber(Integer.parseInt(actionNumber));
}
```

2.2.2 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:

```
public void turnOnRobot() {
    robot.OpenEV3(data.getName());
}

public synchronized void moveBackwards() {
    buffer.put(new Movement(robot, logger,
       MovementEnum.BACKWARDS, -data.getDistance()));
    notify();
}

public synchronized void moveRightCurve() {
    buffer.put(new Movement(robot, logger,
       MovementEnum.RIGHT, data.getRadius(),
       data.getAngle()));
    notify();
}

public synchronized void putBuffer(Movement movement) {
    buffer.put(movement);
    notify();
}
```

2.2.3 Métodos: Movimentos Aleatórios

Estes métodos são responsáveis por iniciar e parar uma nova *Thread* responsável pela criação dos Movimentos Aleatórios do robô.

```
public void startRandomMovements() {
    randomMovements.setActionNumber(data.getActionNumber());
    randomMovements.setWorking(true);
}

public void stopRandomMovements() {
    randomMovements.setWorking(false);
}
```

2.2.4 Controlador do Buffer

```
while (true) {
    switch (bufferState) {
    case IDLE:
        synchronized (this) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
                Thread.currentThread().interrupt();
            }
        if (!buffer.isEmpty())
            bufferState = StateEnum.EXECUTE;
        break;
    case EXECUTE:
        movement = buffer.get();
        movement.doMovement();
        waitingTime = movement.getTime();
        bufferState = StateEnum.WAIT;
        break;
    case WAIT:
        try {
            Thread.sleep(waitingTime);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
            if (buffer.isEmpty()) {
                bufferState = StateEnum.IDLE;
                stopMovementSync();
            } else {
                bufferState = StateEnum.EXECUTE;
            }
        break;
    default:
        break;
    }
}
```

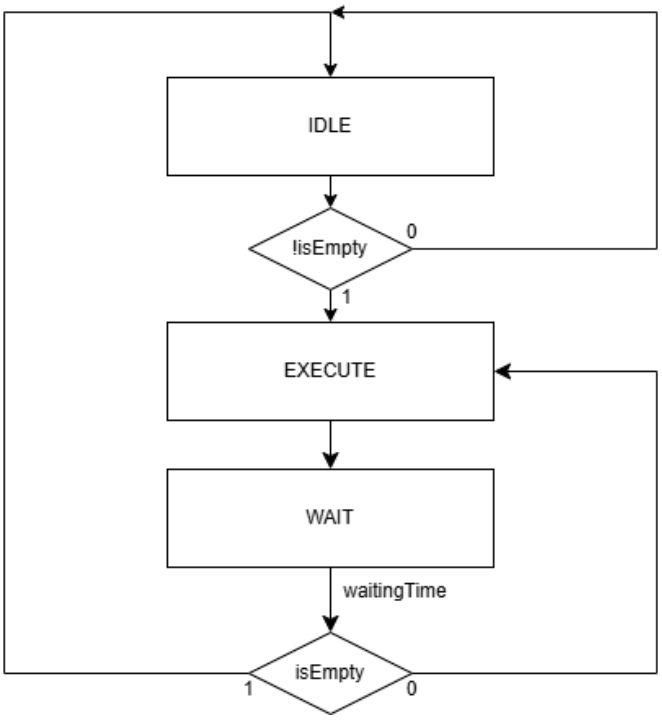

Figura 1: Diagrama de Estados: Buffer Controller

2.3 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.

```
public Data(int radius, int angle, int distance, int
   actionNumber, String name) {
    this.distance = distance;
    this.angle = angle;
    this.radius = radius;
    this.actionNumber = actionNumber;
    this.name = name;
}
```

2.4 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.

```
public void run() {
    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 =
                        MovementEnum.values();
22                  int direction =
                        random.nextInt(movement.length);
23
24                  if (lastDirection == movement[direction]) {
25                      i--;
26                      continue;
27                  }
28
29                  if (movement[direction] ==
                        MovementEnum.FORWARD)
30                      movementList[i * 2] = (new
                            Movement(robot, this.logger,
                            movement[direction],
31                                  random.nextInt(40) + 10));
32                  else if (movement[direction] ==
                        MovementEnum.RIGHT || movement[direction]
                        == MovementEnum.LEFT)
33                      movementList[i * 2] = (new
                            Movement(robot, this.logger,
                            movement[direction],
                            random.nextInt(20) + 10,
34                                  random.nextInt(70) + 20));
35                  else {
36                      i--;
37                      continue;
38                  }
39                  waitingTime += movementList[i * 2].getTime();
40
41                  movementList[i * 2 + 1] = (new
```

```
                    Movement(robot, this.logger,
                    MovementEnum.STOP));
                waitingTime += movementList[i * 2 +
                    1].getTime();

                lastDirection = movement[direction];
            }

            STATE = StateEnum.SEND;
            break;

        case SEND:
            bufferManager.acquire();
            for (int i = 0; i < this.actionNumber * 2; i++) {
                robotController.putBuffer(movementList[i]);
            }
            bufferManager.release();
            STATE = StateEnum.WAIT;
            break;

        case WAIT:
            try {
                Thread.sleep(waitingTime + 4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (this.working)
                STATE = StateEnum.GENERATE;
            else
                STATE = StateEnum.IDLE;
            break;
        default:
            break;
        }
    }
}
```

2.4.1 Método: setWorking

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

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

2.4.2 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 variàvel *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*.

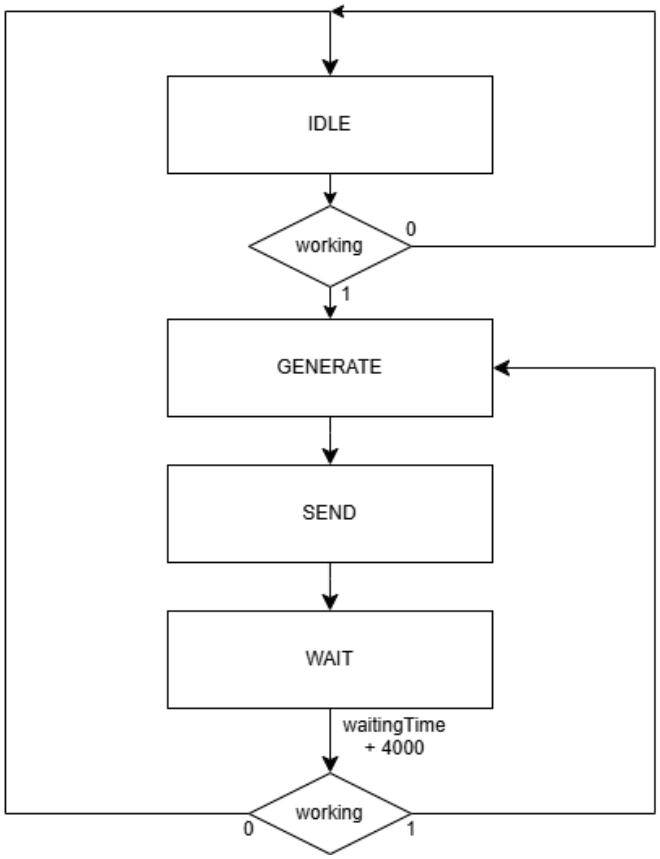

Figura 2: Diagrama de Estados: Movimentos Aleatórios

2.5 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*, 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
           movement) {
15         buffer.add(0, movement);
16         notifyAll();
17     }
18
19     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 }
```

2.5.1 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      }
```

2.6 Classe: Movement

2.6.1 Construtores

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

```
public Movement(RobotLegoEV3 robot, ILogger logger,
    MovementEnum movement, int distance) {
     this.logger = logger;
     this.movement = movement;
     this.distance = distance;
     this.robot = robot;
}

public Movement(RobotLegoEV3 robot, ILogger logger,
    MovementEnum movement, int radius, int angle) {
     this.logger = logger;
     this.movement = movement;
     this.radius = radius;
     this.angle = angle;
     this.robot = robot;
}
```

2.6.2 Método: doMovement

Este método é usado para executar o movimento do robô.

```
public void doMovement() {
     switch (this.movement) {
     case FORWARD:
          robot.Reta(this.distance);
          //log
          break;
     case BACKWARDS:
          robot.Reta(-this.distance);
          //log
          break;
     case RIGHT:
          robot.CurvarDireita(this.radius, this.angle);
          //log
          break;
     case LEFT:
          robot.CurvarEsquerda(this.radius, this.angle);
          //log
          break;
     }
}
```

2.6.3 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.

```
public int getDelayStraightLine() {
    return (int) ((distance / 0.02) + 100);
}

public int getDelayCurve() {
    return (int) (((Math.toRadians(this.angle) * this.radius)
       / 0.02) + 100);
}
```

2.7 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.

```
public class BufferManager {
    private boolean inUse = false;

    public BufferManager() {
    }

    public synchronized void entrar() {
        while (inUse) {
            try {
                wait();
            } catch (InterruptedException e) {
               e.printStackTrace();
                Thread.currentThread().interrupt();
            }
        }
        inUse = true;
    }

    public synchronized void sair() {
        inUse = false;
        notifyAll();
    }
}
```

2.8 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.

```
public void run() {
        while(true) {
            switch(STATE) {
            case IDLE:
                synchronized(this) {
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    /* MODO ROBO */
                    if(robotController.robotOn &&
                       robotController.obstacleFound())
                        STATE = StateEnum.SEND;

                    /* MODO TESTE
                    if(robotController.obstacleFound())
                        STATE = StateEnum.GENERATE;
                    */

                }
                break;
            case GENERATE:
                waitingTime = 0;

                movementList[0] = new Movement(this.robot,
                   this.logger, MovementEnum.STOP);
                movementList[1] = new Movement(this.robot,
                   this.logger, MovementEnum.BACKWARDS, 20);

                while(true) {
                    MovementEnum[] movement =
                       MovementEnum.values();
                    int direction =
                       random.nextInt(movement.length);

                    if(movement[direction] ==
                       MovementEnum.LEFT ||
                       movement[direction] ==
                       MovementEnum.RIGHT) {
                        movementList[2] = new
```

```
                            Movement(this.robot, this.logger,
                            movement[direction], 0, 90);
                        break;
                    }
                }

                for(Movement movement : movementList)
                    waitingTime += movement.getTime();

            case SEND:
                bufferManager.acquire();
                for(int i = MOVEMENT_NUMBER - 1; i >= 0; i--)
                   {
                    robotController.
                    putBufferHigherPriority(movementList[i]);
                }
                bufferManager.release();
                STATE = StateEnum.WAIT;
                break;
            case WAIT:
                try {
                    Thread.sleep(waitingTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                STATE = StateEnum.IDLE;

                break;
            default:
                break;
            }
        }
    }
```

2.9 UML

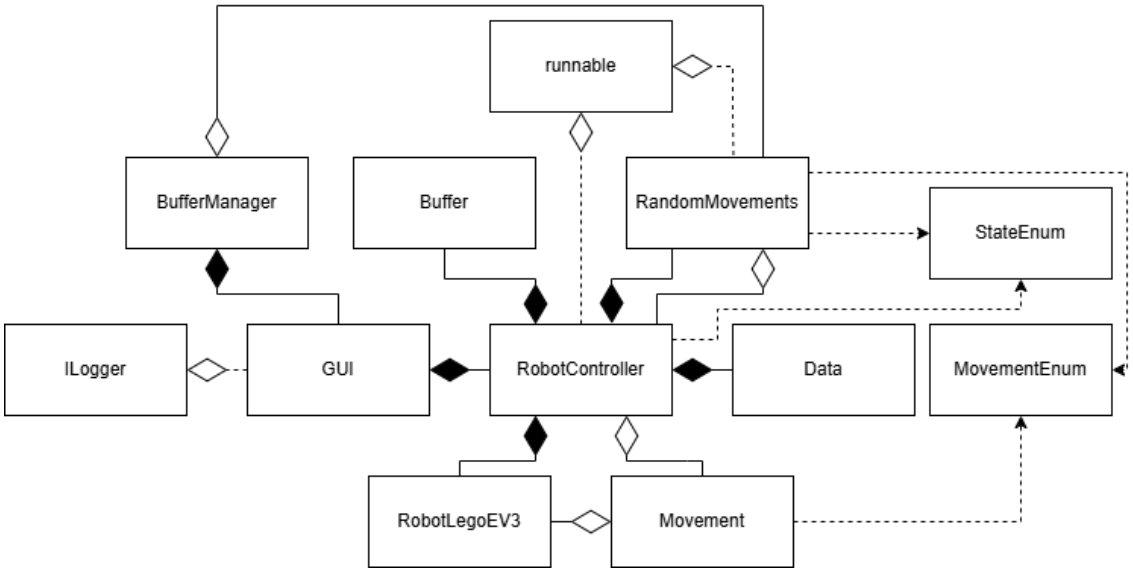

Figura 3: UML

3 Conclusão

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.