instituto
superior de
engenharia
de lisboa

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Licenciatura em Engenharia Informática e Multimédia

Trabalho Prático 2

Fundamentos de Sistemas Operativos

Alunos:
A51417 - David Santos
A52483 - Bernardo Aguiar

Docentes:
Prof. Carlos Gonçalves

Outubro de 2025

Conteúdo

Lista de Figuras

Listings

1 Introdução

Este projeto, no âmbito da disciplina de Fundamentos de Sistemas Operativos, tem como objetivo principal o desenvolvimento de uma aplicação multitarefa em Java para o controlo avançado de um robô. O sistema é concebido para demonstrar e aplicar conceitos essenciais de programação concorrente, como a sincronização entre tarefas no acesso a um robô. O trabalho é composto por duas fases (Trabalho Prático 1 e 2), visando a criação de um sistema robusto e reativo, capaz de executar comandos de forma controlada, gerar movimentos autónomos e, simultaneamente, reagir a eventos externos, como obstáculos.

A aplicação inclui uma Interface Gráfica de Utilizador (GUI), desenvolvida utilizando o editor WindowBuilder. Esta interface constitui o ponto de contacto primário do utilizador com o sistema de controlo. Através da GUI, o utilizador pode interagir diretamente com o robô, emitindo comandos manuais de movimento essenciais, como FRENTE, TRÁS, PARAR, ESQUERDA e DIREITA. Adicionalmente, a GUI permite ativar ou desativar a tarefa de Movimentos Aleatórios , uma funcionalidade que gera sequências de comandos de forma autónoma. A interface é complementada por uma Consola , uma área de texto essencial para exibir as ações realizadas pelo robô e mensagens informativas ou de debugging do programa.

No final da execução do projeto, espera-se obter um robô capaz de se movimentar em todas as direções e que integre eficazmente os mecanismos de controlo implementados. O trabalho culmina com a adição de duas tarefas críticas. A primeira tarefa "Evitar Obstáculo": Uma funcionalidade de segurança que monitoriza continuamente o sensor de toque do robô e, em caso de colisão, pára imediatamente o robô, executa um recuo e uma curva aleatória para prevenir o bloqueio. A segunda tarefa "Gravador": Uma funcionalidade que permite gravar e reproduzir sequências de comandos executados pelo robô, com manipulação obrigatória de ficheiros.

2 Estrutura

O projeto está estruturado com o padrão de arquitetura Model-View-Controller (MVC), onde a classe GUI funciona como a View, gerindo a apresentação e as interações do utilizador. A classe Controller desempenha o papel de Controller, centralizando o fluxo de execução do programa, recebendo as entradas e orquestrando as ações. E todas as restantes classes constituem o Model, sendo responsáveis pela lógica e a manipulação dos dados.

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.

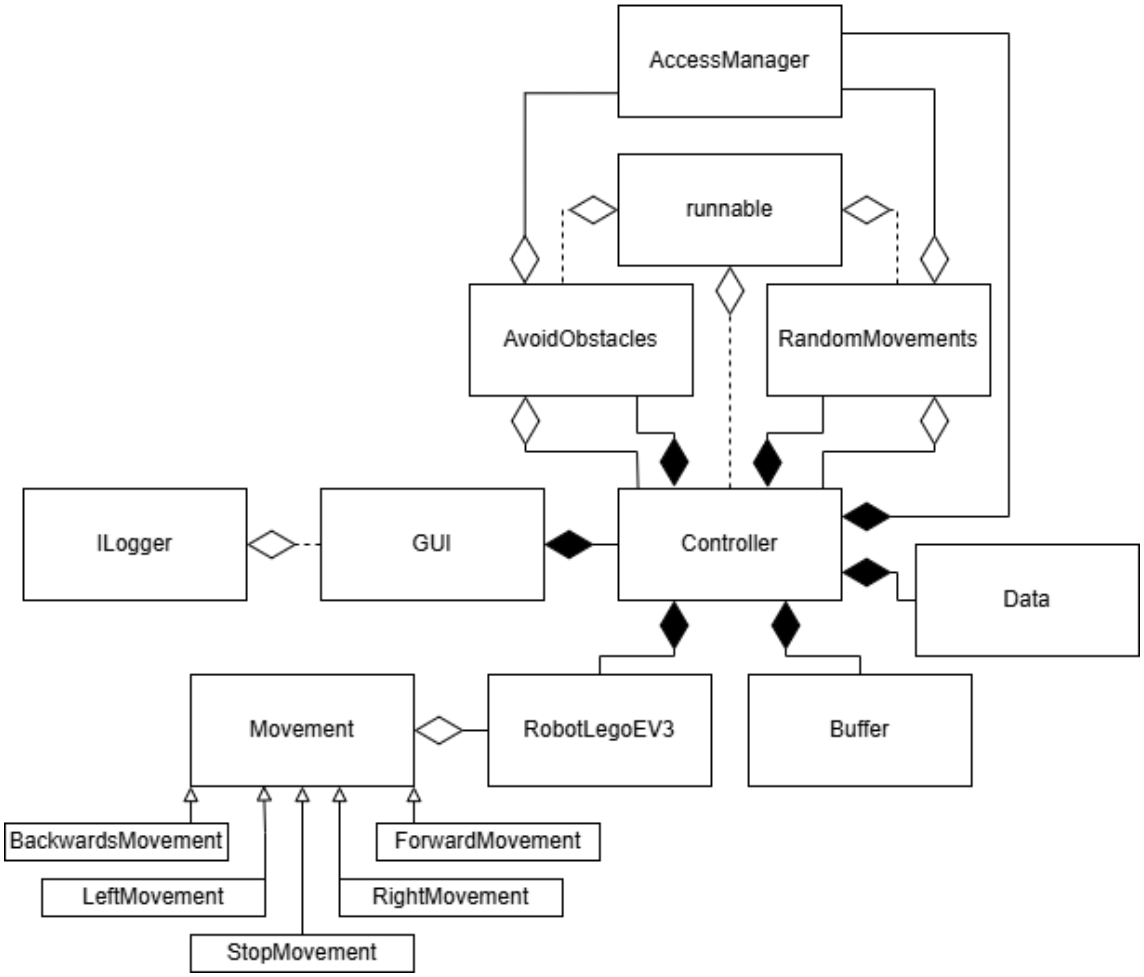

Figura 2.1.1: UML

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

Esta classe constitui a interface primária de interação para o utilizador, servindo como o componente View essencial do sistema de controlo do robô. A sua responsabilidade é apresentar o estado do sistema e capturar as entradas e comandos do utilizador. Sempre que uma interação do utilizador ocorre através de um elemento da interface, esta classe encaminha os comandos correspondentes diretamente para a classe RobotController, garantindo assim a execução das ações pretendidas no robô.

3.2 Classe: Data

Esta classe funciona como o Model de dados do sistema e possui a finalidade estrita de armazenamento e agregação de informações. A sua principal função é encapsular num único objeto todos os parâmetros de configuração e entrada provenientes da View (GUI), tais como distância, ângulo, raio e número de ações aleatórias.

Listing 1: Construtor da Classe Data

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

3.3 Classe: Movement

Esta classe serve como uma classe base abstrata fundamental para a definição do contrato de movimento do robô. O seu propósito é estabelecer a estrutura e os métodos obrigatórios que todas as classes de movimento específicas deverão implementar ao herdarem Movement.

Listing 2: Classe Abstrata para Movimentos

```
public abstract class Movement {
    private ILogger logger;

    public Movement(ILogger logger) {
        this.logger = logger;
    }

    public abstract void doMovement();
```

```
9
10      public abstract int getTime();
11
12      protected void log(String message) {
13          if (logger != null) {
14              logger.logMessage(message);
15          }
16      }
17  }
```

3.4 Classe: ForwardMovement

Esta classe herda da classe abstrata Movement e atua como uma implementação concreta dessa interface. A sua responsabilidade primária é fornecer a lógica específica e necessários para realizar a ação de movimento em frente do robô, preenchendo assim os métodos obrigatórios definidos pela classe base.

Listing 3: Classe ForwardMovement

```
1  public class ForwardMovement extends Movement {
2
3      private int distance;
4      private RobotLegoEV3 robot;
5
6      public ForwardMovement(int distance, RobotLegoEV3 robot,
           ILogger logger) {
7          super(logger);
8          this.distance = distance;
9          this.robot = robot;
10     }
11
12     public void doMovement() {
13         robot.Reta(this.distance);
14         log("O rob  andou para a frente " +
               Math.abs(this.distance) + " cent metros.\n");
15     }
16
17     public int getTime() {
18         return (int) ((distance / 0.02) + 100);
19     }
20 }
```

3.5 Classe: BackwardsMovement

Esta classe herda da classe abstrata Movement e atua como uma implementação concreta dessa interface. A sua responsabilidade primária é fornecer a lógica específica e necessários para realizar a ação de movimento para trás do robô, preenchendo assim os métodos obrigatórios definidos pela classe base.

Listing 4: Classe BackwardsMovement

```
public class BackwardsMovement extends Movement {

    private int distance;
    private RobotLegoEV3 robot;

    public BackwardsMovement(int distance, RobotLegoEV3
       robot, ILogger logger) {
        super(logger);
        this.distance = distance;
        this.robot = robot;
    }

    public void doMovement() {
        robot.Reta(-this.distance);
        log("O rob  andou para tr s " +
           Math.abs(this.distance) + " cent metros.\n");
    }

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

3.6 Classe: RightMovement

Esta classe herda da classe abstrata Movement e atua como uma implementação concreta dessa interface. A sua responsabilidade primária é fornecer a lógica específica e necessários para realizar a ação de movimento de uma curva para a direita do robô, preenchendo assim os métodos obrigatórios definidos pela classe base.

Listing 5: Classe RightMovement

```
public class RightMovement extends Movement {

    private int radius, angle;
    private RobotLegoEV3 robot;

    public RightMovement(int radius, int angle, RobotLegoEV3
       robot, ILogger logger) {
        super(logger);
        this.radius = radius;
        this.angle = angle;
        this.robot = robot;
    }

    public void doMovement() {
        robot.CurvarDireita(this.radius, this.angle);
```

```
15          log("O rob  curvou   direita com um  ngulo  de " +
               this.angle + " graus e com um raio de " +
               this.radius
16                  + " cent metros.\n");
17      }
18
19      public int getTime() {
20          return (int) (((Math.toRadians(this.angle) *
               this.radius) / 0.02) + 100);
21      }
22
23  }
```

3.7 Classe: LeftMovement

Esta classe herda da classe abstrata Movement e atua como uma implementação concreta dessa interface. A sua responsabilidade primária é fornecer a lógica específica e necessários para realizar a ação de movimento de uma curva para a esquerda do robô, preenchendo assim os métodos obrigatórios definidos pela classe base.

Listing 6: Classe LeftMovement

```
1   public class LeftMovement extends Movement {
2
3       private int radius, angle;
4       private RobotLegoEV3 robot;
5
6       public LeftMovement(int radius, int angle, RobotLegoEV3
           robot, ILogger logger) {
7           super(logger);
8           this.radius = radius;
9           this.angle = angle;
10          this.robot = robot;
11      }
12
13      public void doMovement() {
14          robot.CurvarEsquerda(this.radius, this.angle);
15          log("O rob  curvou   esquerda com um  ngulo  de " +
               this.angle + " graus e com um raio de " +
               this.radius
16                  + " cent metros.\n");
17      }
18
19      public int getTime() {
20          return (int) (((Math.toRadians(this.angle) *
               this.radius) / 0.02) + 100);
21      }
22
23  }
```

3.8 Classe: StopMovement

Esta classe herda da classe abstrata Movement e atua como uma implementação concreta dessa interface. A sua responsabilidade primária é fornecer a lógica específica e necessários para realizar a paragem imediata do robô, preenchendo assim os métodos obrigatórios definidos pela classe base.

Listing 7: Classe StopMovement

```
public class StopMovement extends Movement {
    private RobotLegoEV3 robot;

    public StopMovement(RobotLegoEV3 robot, ILogger logger) {
        super(logger);
        this.robot = robot;
    }

    public void doMovement() {
        robot.Parar(true);
        log("O rob  parou. ");
    }

    public int getTime() {
        return 100;
    }

}
```

3.9 Classe: Buffer

O buffer foi implementado utilizando o mecanismo de monitores para garantir a sincronização adqueada. Foi adotada uma estrutura de dados ArrayList (em detrimento de um Array de capacidade fixa) para o buffer. Esta escolha elimina potenciais vulnerabilidades associadas à limitação de capacidade, prevenindo erros críticos como a perda de dados ou a paragem/bloqueio indevido de Threads. Desta forma, esta solução demonstra ser a abordagem mais simples, eficiente e intuitiva para a gestão dinâmica do buffer, otimizando a utilização dos recursos do sistema.

Listing 8: Classe Buffer

```
public class Buffer {

    private ArrayList<Movement> buffer;

    public Buffer() {
        this.buffer = new ArrayList<>();
    }

    public synchronized void put(Movement movement) {
        buffer.add(movement);
```

```
        notifyAll();
    }

    public synchronized void higherPriorityPut(Movement
       movement) {
        buffer.add(0, movement);
        notifyAll();
    }

    public synchronized Movement get() {
        while (isEmpty()) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        return buffer.remove(0);
    }

    public boolean isEmpty() {
        return buffer.isEmpty();
    }

    //Resto do c digo
}
```

3.9.1 Método: cleanBuffer

Este método tem como função principal realizar a reinicialização do buffer de dados. A sua invocação ocorre de forma mandatória sempre que é acionado o comando stop do robô, garantindo que o buffer é limpo e fica num estado consistente e vazio antes de qualquer nova sequência de comandos.

Listing 9: Método de Limpeza do Buffer

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

3.10 Classe: AccessManager

Esta classe implementa um mecanismo de exclusão mútua simples com o objetivo de controlar o acesso a uma instância ou recurso crítico por parte de múltiplas threads.

Listing 10: Classe AccessManager

```
public class AccessManager {

```

```
    private boolean inUse = false;

    public AccessManager() {
    }

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

    public synchronized void release() {
        inUse = false;
        notify();
    }
}
```

3.11 Classe: Controller

Esta classe desempenha o papel de Controller dentro do padrão de arquitetura MVC. A sua função essencial é ser a ponte de comunicação entre os componentes lógicos da aplicação (View e Model) e o robô físico.

3.11.1 Métodos: Atualizações de Dados

Estes métodos atuam como uma ponte, facilitando a transferência controlada dos valores de entrada provenientes da GUI para a classe Data. O seu propósito é encapsular e gerir o fluxo de dados, promovendo uma organização modular e uma separação de responsabilidades mais eficaz entre a camada de apresentação e a camada de dados, o que é fundamental para a manutenção e escalabilidade do código.

Listing 11: Métodos de Atualização de Dados da Classe Data

```
public void updateRadius(int radius) {
    data.setRadius(radius);
}

public void updateAngle(int angle) {
    data.setAngle(angle);
}

public void updateDistance(int distance) {
    data.setDistance(distance);
```

```
}

public void updateActionNumber(int actionNumber) {
    data.setActionNumber(actionNumber);
}
```

3.11.2 Métodos: Comunicação com o Buffer

Estes métodos representam a interação com o buffer. Têm um design simples e claro com a finalidade de abstrair a complexidade das operações subjacentes.

Listing 12: Métodos de Envio para o Buffer

```
public synchronized void bufferMoveForward() {
    bufferManager.acquire();
    buffer.put(new ForwardMovement(data.getDistance(), robot,
       logger));
    bufferManager.release();
    notify();
}

public synchronized void bufferMoveBackwards() {
    bufferManager.acquire();
    buffer.put(new BackwardsMovement(data.getDistance(),
       robot, logger));
    bufferManager.release();
    notify();
}

public synchronized void bufferMoveRightCurve() {
    bufferManager.acquire();
    buffer.put(new RightMovement(data.getRadius(),
       data.getAngle(), robot, logger));
    bufferManager.release();
    notify();
}

public synchronized void bufferMoveLeftCurve() {
    bufferManager.acquire();
    buffer.put(new LeftMovement(data.getRadius(),
       data.getAngle(), robot, logger));
    bufferManager.release();
    notify();
}

public synchronized void bufferStopMovement() {
    bufferManager.acquire();
    buffer.put(new StopMovement(robot, logger));
    bufferManager.release();
```

```
33      notify();
34  }
35
36  public synchronized void putBuffer(Movement movement) {
37      buffer.put(movement);
38      notify();
39  }
40
41  public synchronized void putBufferHigherPriority(Movement
        movement) {
42      buffer.higherPriorityPut(movement);
43      notify();
44  }
45
46  public void stopMovement() {
47      robotManager.acquire();
48      new StopMovement(robot, logger).doMovement();;
49      robotManager.release();
50      this.waitingTime = 0;
51  }
52
53  public void stopMovementSync() {
54      robotManager.acquire();
55      robot.Parar(false);
56      robotManager.release();
57  }
58
59  public void squareMovement() {
60      bufferMoveForward();
61      bufferMoveLeftCurve();
62      bufferMoveForward();
63      bufferMoveLeftCurve();
64      bufferMoveForward();
65      bufferMoveLeftCurve();
66      bufferMoveForward();
67      bufferMoveLeftCurve();
68  }
```

3.11.3 Métodos: Movimentos Aleatórios

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

Listing 13: Métodos de Configuração da Máquina de Estados dos Movimentos Aleatórios

```
1  public void startRandomMovements() {
2      randomMovements.setActionNumber(data.getActionNumber());
3      randomMovements.setWorking(true);
4  }
```

```
5
6 public void stopRandomMovements() {
7     randomMovements.setWorking(false);
8 }
```

3.11.4 Método: obstacleFound

Este método tem como função principal retornar um valor booleano que indica o estado de colisão atual do robô.

Listing 14: Método da Deteção de Obstáculos

```
1 public synchronized boolean obstacleFound() {
2     return robot.SensorToque(robot.S_1) == 1;
3 }
```

3.11.5 Controlador do Buffer

Esta Máquina de Estados é responsável pela execução sequencial dos comandos de movimento contidos no buffer. Opera num ciclo onde espera no estado IDLE até que o buffer contenha movimentos, e assim que um comando é detetado, inicia a sua execução no robô e espera até que esse movimento seja completamente concluído antes de processar o próximo ou regressar ao estado IDLE caso o buffer esteja vazio.

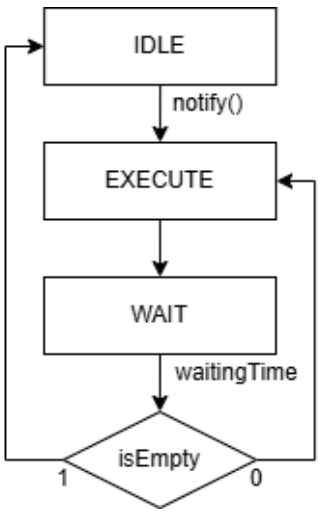

Figura 3.11.1: Diagrama de Estados: Buffer Controller

Listing 15: Máquina de Estados do Buffer

```
1 public void run() {
2     while (true) {
3         switch (bufferState) {
4         case IDLE:
5             synchronized (this) {
```

```
                    try {
                        this.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                bufferState = StateEnum.EXECUTE;
                break;
            case EXECUTE:
                bufferManager.acquire();
                movement = buffer.get();
                bufferManager.release();

                robotManager.acquire();
                movement.doMovement();
                robotManager.release();

                waitingTime = movement.getTime();
                bufferState = StateEnum.WAIT;
                break;
            case WAIT:
                try {
                    wait(waitingTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (buffer.isEmpty()) {
                    bufferState = StateEnum.IDLE;
                    stopMovementSync();
                } else
                    bufferState = StateEnum.EXECUTE;
                break;
            default:
                break;
            }
        }
}

public void clearBuffer() {
    buffer.clearBuffer();
}
```

3.12 Classe: RandomMovements

Esta classe implementa uma Máquina de Estados dedicada à geração autónoma de movimentos aleatórios para o robô. Esta é executada numa Thread, garantindo que a criação de novos movimentos não bloqueia o fluxo principal da aplicação. O objetivo principal desta é calcular e gerar movimentos de forma aleatória dentro de algumas regras, os quais

são subsequentemente enviados para o buffer através do RobotController.

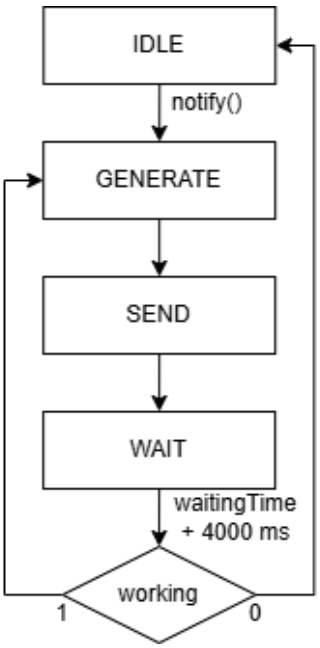

Figura 3.12.1: Diagrama de Estados: Movimentos Aleatórios

Listing 16: Máquina de Estados dos Movimentos Aleatórios

```
public void run() {
    while (true) {
        switch (STATE) {
        case IDLE:
            synchronized (this) {
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            STATE = StateEnum.GENERATE;

            break;
        case GENERATE:
            movementList = new Movement[actionNumber * 2];
            waitingTime = 0;

            for (int i = 0; i < this.actionNumber; i++) {
                MovementEnum[] movement =
                   MovementEnum.values();
                int direction =
                   random.nextInt(movement.length);

                if (lastDirection == movement[direction]) {
```

```
                    i--;
                    continue;
                }

                if (movement[direction] ==
                   MovementEnum.FORWARD)
                    movementList[i * 2] = (new
                       ForwardMovement((random.nextInt(40) +
                       10), robot, logger));
                else if (movement[direction] ==
                   MovementEnum.RIGHT)
                    movementList[i * 2] = (new
                       RightMovement(random.nextInt(20) + 10,
                       random.nextInt(70) + 20,
                            robot, logger));
                else if (movement[direction] ==
                   MovementEnum.LEFT)
                    movementList[i * 2] = (new
                       LeftMovement(random.nextInt(20) + 10,
                       random.nextInt(70) + 20, robot,
                            logger));
                else {
                    i--;
                    continue;
                }

                waitingTime += movementList[i * 2].getTime();

                movementList[i * 2 + 1] = (new
                   StopMovement(robot, this.logger));
                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 {
                wait(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
70                break;
71            default:
72                break;
73            }
74        }
75    }
```

3.12.1 Métodos: Configurações

O método setWorking tem como função mudar o estado da máquina de estados e, simultaneamente, acordar a Thread associada (se estiver em espera), permitindo que a máquina de estados retome a execução do seu ciclo de trabalho.

O método setActionNumber tem a função de definir o valor que corresponde ao número de ações aleatórias que a Máquina de Estados irá gerar para o robô.

Listing 17: Métodos de Configuração da Máquina de Estados dos Movimentos Aleatórios

```
1  public synchronized void setWorking(boolean working) {
2      this.working = working;
3      if (working)
4          this.notify();
5  }
6
7  public synchronized void setActionNumber(int actionNumber) {
8      this.actionNumber = actionNumber;
9  }
```

3.12.2 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*.

3.13 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.

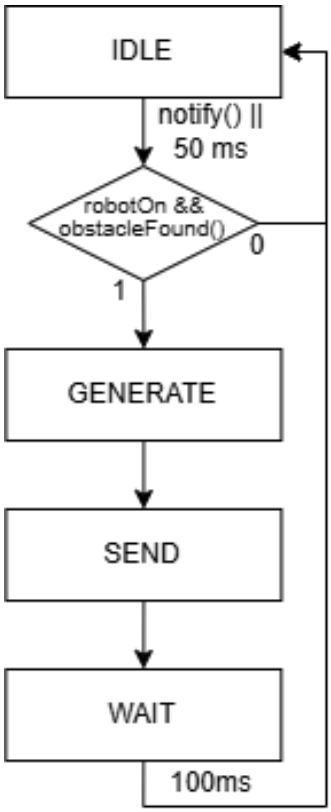

Figura 3.13.1: Diagrama de Estados: Avoid Obstacles

Listing 18: Máquina de Estados para Evitar Obstáculos

```
1 public void run() {
2     while (true) {
3         switch (STATE) {
4         case IDLE:
5             synchronized (this) {
6                 try {
```

```
                    wait(50);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                robotManager.acquire();
                if (robotController.robotOn &&
                   robotController.obstacleFound())
                    STATE = StateEnum.GENERATE;
                robotManager.release();
            }
            break;
        case GENERATE:
            waitingTime = 0;

            movementList[0] = new StopMovement(this.robot,
               this.logger);
            movementList[1] = new BackwardsMovement(20,
               this.robot, this.logger);

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

                if (movement[direction] == MovementEnum.LEFT)
                   {
                    movementList[2] = new LeftMovement(10,
                       90, this.robot, this.logger);
                    break;
                } else if (movement[direction] ==
                   MovementEnum.RIGHT) {
                    movementList[2] = new RightMovement(10,
                       90, this.robot, this.logger);
                    break;
                }
            }

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

            STATE = StateEnum.SEND;

        case SEND:
            bufferManager.acquire();
            for (int i = MOVEMENT_NUMBER - 1; i >= 0; i--)
                robotController.putBufferHigherPriority
                   (movementList[i]);
```

```
46              bufferManager.release();
47
48              STATE = StateEnum.WAIT;
49              break;
50          case WAIT:
51              try {
52                  wait(waitingTime);
53              } catch (InterruptedException e) {
54                  e.printStackTrace();
55              }
56
57              STATE = StateEnum.IDLE;
58
59              break;
60          default:
61              break;
62          }
63      }
64  }
```

4 Resultados

Foi possível verificar que o robô executou com precisão os comandos de movimento manual emitidos através da GUI (FRENTE, TRÁS, ESQUERDA, DIREITA e PARAR), confirmando a correta tradução das intenções do utilizador em comandos de hardware.

A Tarefa "Movimentos Aleatórios"mostrou o robô a executar uma sequência de movimentos (retas e curvas) gerados automaticamente. Mesmo durante a execução desta tarefa, a sincronização do sistema assegurou que nem os comandos manuais nem a tarefa de "Evitar Obstáculo"interferissem ou causassem perda de controlo sobre o robô.

A Tarefa "Evitar Obstáculo"demonstrou a sua eficácia no mundo real. Foi observado que o robô parava imediatamente (em milissegundos) assim que o sensor de toque (pára-choques) era ativado. Em seguida, efetuava um recuo de 20 cm, seguido de uma curva aleatória de 90 graus (à esquerda ou à direita). Esta sequência comprovou a capacidade de reatividade do sistema para evitar colisões e prosseguir a navegação de forma autónoma.

A Tarefa "Gravador"permitiu reproduzir fisicamente sequências de comandos que haviam sido previamente registados em ficheiro, garantindo que o robô repetia exatamente as trajetórias gravadas, validando a fidelidade da manipulação de dados em disco.

Em suma, as validações físicas atestaram que o robô se comportou de forma fiável e determinística, provando o sucesso da implementação do controlo multitarefa e dos mecanismos de sincronização.

5 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.