LICENCIATURA EM ENGENHARIA INFORMÁTICA E MULTIMÉDIA

Inteligência Artificial para Sistemas Autónomos

4º Semestre 2023/2024

Relatório - Entrega 1

Docente: Engº Luís Morgado

28/04/2024

Turma: 41D

Trabalho realizado por:
Daniela Cafum: A50032

Conteúdo

Lista de Figuras

1 Introdução

Como projeto da unidade curricular de Inteligência Artificial para Sistemas Autónomos, foi proposto aos alunos o desenvolvimento de um jogo com diversas partes onde fosse englobada toda a matéria lecionada durante este semestre letivo de Verão 2023/24.

Ao longo do semestre, foram apresentados os fundamentos essenciais do ramo de inteligência artificial. A Inteligência Artificial (IA ou AI) investiga a elaboração de sistemas informáticos capazes de possuir um comportamento inteligente. Para sistemas autónomos, IA concentra-se na capacidade das máquinas executarem tarefas de uma forma independente.

Essencialmente, tais sistemas destacam-se pela sua habilidade de tomar decisões e agir no conceito de mundo real com o mínimo de intervenção humana possível. É com base nesse conceito que o projeto para a disciplina de IASA será desenvolvido.

Durante o semestre, foram desenvolvidas várias bibliotecas com o propósito da criação de aplicações para agentes autónomos.

2 Enquadramento teórico

Primeiramente, é necessário explicar teoricamente, de uma forma resumida, cada parte que será abrangida por este trabalho prático.

2.1 Introdução à Inteligência Artificial

Tal como foi dito na introdução deste relatório, a Inteligência Artificial é uma área científica que estuda o desenvolvimento de sistemas computacionais capazes de comportamento inteligente.

O desenvolvimento de sistemas com base em inteligência artificial é caracterizado pela elevada complexidade desses sistemas. Requer métodos adequados de engenharia de software, nomeadamente, no que se refere à modelação de sistemas, bem como a linguagens de modelação adequadas, como é o caso da linguagem UML.

A IA adopta duas perspectivas principais:

- **Analítica:** por via empírica (baseada na prática, com experiências e observações) desenvolver modelos e teorias para explicar os fenómenos observados e reproduzir esses fenómenos em sistemas artificiais (dispositivos criados para o efeito);
- **Sintética:** com base nesses modelos e teorias, desenvolver sistemas capazes de apresentar características e comportamentos associados ao conceito de inteligência.

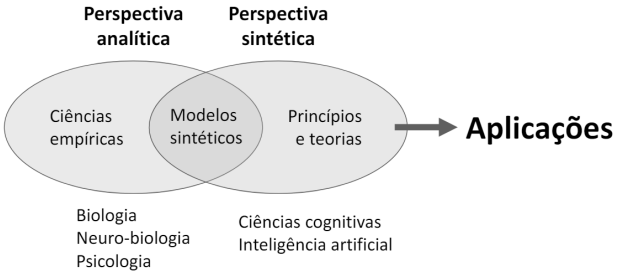


Figura 1: Perspetivas de IA

2.2 Arquitetura de Agentes Autónomos

Nesta aquitetura, um sistema autónomo inteligente opera num ciclo de percepção-processamento-acção, através do qual é realizado o controlo da função do sistema de modo a concretizar a finalidade desse sistema (por exemplo, manter uma distância mínima a outro veículo).

Neste ciclo de percepção-processamento-acção, um agente percebe o seu ambiente, processa as perceções, toma decisões e age de forma autónoma, ou seja, sem qualquer intervenção direta de um ser humano.

Os sistemas autónomos são capazes de operar de forma independente em ambientes complexos e dinâmicos, adaptando-se facilmente às mudanças e tomando decisões em tempo real com base nas informações disponíveis, por isso estes sistemas são frequentemente utilizados numa variedade de aplicações, como veículos autónomos, robótica industrial, sistemas de navegação, entre outros.

O ciclo de um agente autónomo pode ser observado na figura 2.

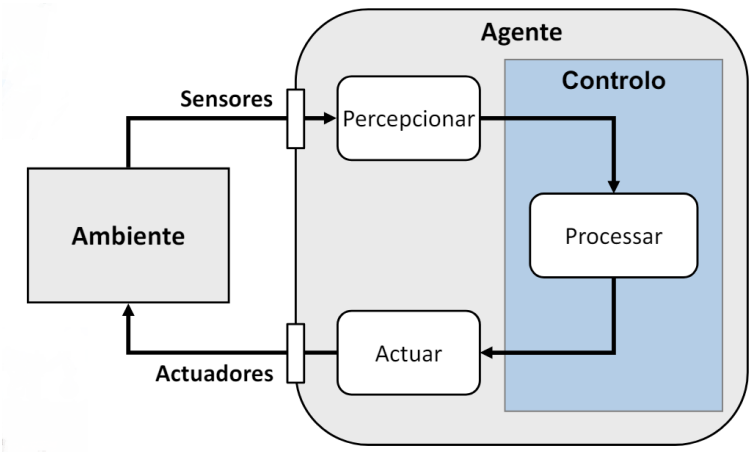


Figura 2: Modelo geral de um agente autónomo

Como podemos observar, o agente percebe o seu ambiente através de sensores, processa essas informações no controlo para tomar decisões, e então executa ações com atuadores para atingir o seu objetivo.

As características principais, dependentes do tipo de arquitectura de agente são:

- **Autonomia:** capacidade de um sistema operar por si próprio, de modo independente de outros sistemas;
- **Reatividade:** Capacidade de um sistema reagir aos estímulos do ambiente;
- **Pró-Atividade:** capacidade de um sistema tomar a iniciativa de acção em função dos seus objectivos;
- **Sociabilidade:** Capacidade de um sistema interactuar e agir em conjunto com outros agentes para concretizar objectivos individuais ou comuns a outros agentes.

Todas estas características do agente estão associadas à concretização da finalidade do agente, ou seja, do seu propósito, expresso na função que realiza.

2.3 Arquitetura de Agentes Reativos

Um agente reativo é, essencialmente, aquele que executa uma tarefa seguindo padrões de comportamento já definidos, como reflexos, baseados na associação entre estímulos e respostas.

Os objetivos desse tipo de agente não são expressos explicitamente, mas encontram-se nas associações entre estímulos e respostas que determinam o seu comportamento, por exemplo, se um agente é estimulado a se aproximar de um local específico, podemos deduzir que o seu objetivo é alcançar esse local, mesmo que não seja diretamente declarado.

Em resumo, uma arquitectura de agentes reactivos define um ciclo percepção-reacção-acção, onde as reacções definem de forma modular as associações entre estímulos (derivados da percepção) e respostas (geradoras de acção).

Esta arquitetura de agentes reativos é útil em ambientes dinâmicos e imprevisíveis, nos quais as respostas imediatas são necessárias para lidar com mudanças rápidas no ambiente. A abordagem simplificada e eficiente desta arquitetura, permite que os agentes atuem de forma rápida e adaptativa, mesmo sem uma compreensão profunda do ambiente em que estão a agir.

Os agentes reativos são também frequentemente utilizados nos sistemas robóticos, jogos eletrônicos e em ambientes de simulação, onde a capacidade de resposta imediata é crucial para o desempenho eficaz do agente.

O ciclo de um agente reativo pode ser observado na figura 3.

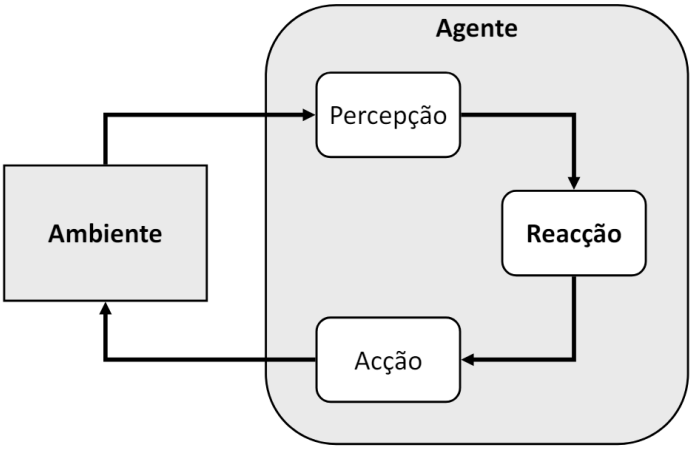


Figura 3: Modelo geral de um agente reativo

3 Realização do Projeto

Como foi mencionado anteriormente, o projeto desta cadeira foi realizado em diversas partes.

3.1 Parte 1

O objetivo inicial da primeira parte do projeto da disciplina foi dar uma introdução à arquitetura com base num modelo UML e a interpretação de máquinas de estado.

Pretende-se implementar um jogo com uma personagem virtual que interage com um jogador humano.

O jogo consiste num ambiente onde a personagem tem por objetivo registar a presença de animais através de fotografias

Quando o jogo começa, a personagem encontra-se numa situação de procura de animais, ao detetar algum ruído, a personagem aproxima-se e investiga a área à procura da fonte do ruído.

Quando o ambiente volta a ficar em silêncio, a personagem retoma a sua procura por animais. Ao detetar um animal, a personagem aproxima-se e observa-o, se o animal permanecer no local, a personagem continua a observa-lo e prepara-se para tirar a fotografia. Caso o animal fuja, a personagem volta a investigar a área á procura da fonte do ruído.

Durante o processo de captura, se o animal estiver presente, a personagem tira uma fotografia e se o animal fugir ou se a fotografia for bem-sucedida, a personagem retoma o processo de procura.

Este primeiro trabalho foi desenvolvido em linguagem *Java* no *Visual Studio Code*.

3.1.1 Componentes Realizados

Para realizar esta parte 1 do projeto seguindo o conceito dado, foi necessário programar por partes, organizadas em pastas, sendo estas:

- **Agente:** onde se encontram todas as classes feitas para o funcionamento do agente;
- **Ambiente:** onde temos as classes que irão simular o nosso ambiente e os eventos que decorrem no mesmo;
- **Jogo:** onde se encontram as classes feitas para o controlo do jogo em sí;
- **Máquina de Estado (MaqEst):** onde estão todas as classes que irão simular o modelo computacional de uma máquina de estados.

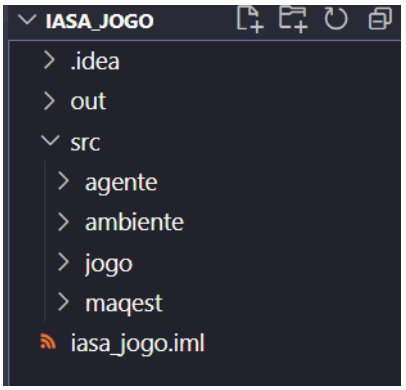


Figura 4: Organização da Parte 1

3.1.2 Agente

Para podermos traduzir a representação conceptual de agente que se encontra na figura 2, seguimos um modelo estrutural de agente na linguagem UML da figura 5 dada pelo docente.

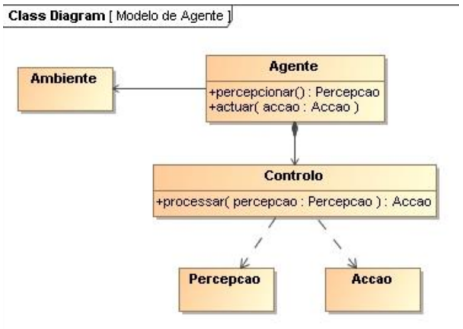


Figura 5: Modelo Estrutural do Agente em UML

Seguindo então o UML dado criámos as classes pedidas.

Para a classe **Agente** tivemos de traduzir o funcionamento do subsistema agente, sendo necessário um método para percepcionar o ambiente, um método para realizar uma ação e o método que irá então executar o próprio agente.

A classe **Controlo** apenas irá definir métodos para controlar o comportamento do agente, processando uma percepção recebida e retornando uma ação correspondente. Irá precisar de duas classes, Percepcão e Acção.

A classe **Acção** irá representa uma ação a ser realizada pelo agente, através de um comando recebido que identifica qual foi a ação escolhida.

Por ultimo, a classe **Percepção** representa uma percepção do ambiente obtida pelo agente, irá receber um evento percebido pelo agente.

3.1.3 Ambiente

Para poder simular o nosso ambiente onde o agente vai atuar seguimos o seguinte UML, dado pelo docente.

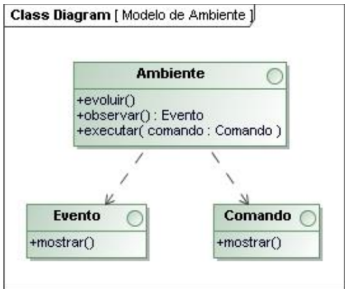


Figura 6: Modelo Estrutural do Ambiente em UML

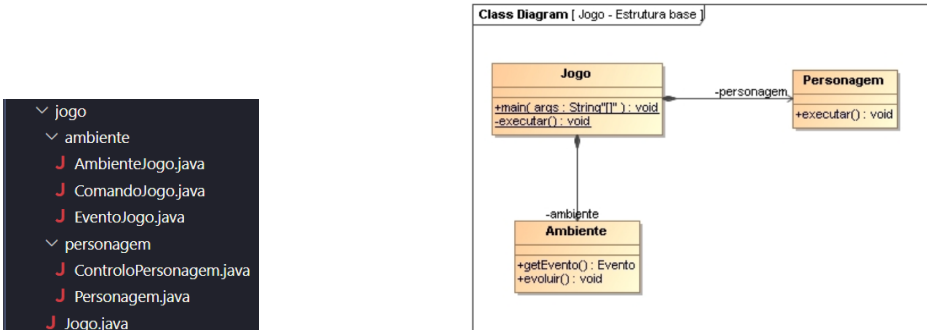
Seguindo o UML, foram criadas três classes.

A classe **Ambiente** irá representar o subsistema ambiente onde o agente se encontra, aqui teremos um método que irá evoluir o ambiente, outro método responsável por devolver um evento que foi observado no ambiente e um método irá executar uma ação no ambiente através de um comando especifico.

As classes **Evento** e **Comando** apenas irão mostrar um evento e um comando respetivamente.

3.1.4 Jogo

Como o jogo consiste num ambiente onde a personagem tem por objetivo registar a presença de animais através de fotografias, sendo que os conceitos do dominio do problema são: Ambiente e Personagem, é necessário dividir o jogo em duas partes. Podemos verificar isto seguindo o UML dado pelo docente.



(a) Organização do Jogo (b) UML do Jogo

Figura 7: Estrutura da Programação Jogo

Para o ambiente foi necessário criar três classe: **AmbienteJogo**, **ComandoJogo** e **EventoJogo**.

Na classe **AmbienteJogo** vamos evoluir o ambiente no tempo, onde é possível executar e mostrar comandos e eventos, sendo que é necessário fazer uma inicialização de cada evento para o utilizador, atribuindo uma letra a cada um.

```
//construtor da classe, inicializa o mapa "eventos" e preenche-o com os eventos que temos na classe EventoJogo
public AmbienteJogo(){

    //inicialização dos eventos
    eventos = new HashMap<String, EventoJogo>();
    eventos.put(key:"s", EventoJogo.SILENCIO);
    eventos.put(key:"r", EventoJogo.RUIDO);
    eventos.put(key:"a", EventoJogo.ANIMAL);
    eventos.put(key:"f", EventoJogo.FUGA);
    eventos.put(key:"p", EventoJogo.FOTOGRAFIA);
    eventos.put(key:"t", EventoJogo.TERMINAR);

}
```

Figura 8: Inicialização dos Eventos do jogo

Teremos um método que irá pegar na letra inserida pelo utilizador (o evento escolhido) e irá gerar o evento correspondente. Será necessário também um método que irá evoluir o estado do ambiente de jogo, um método que irá mostrar o evento observado e outro que executa um comando no ambiente de jogo.

Tanto a classe **EventoJogo** como a classe **ComandoJogo** irão ser enumerados onde se encontram as diferentes constantes possiveis.

Para eventos teremos seis constantes: SILENCIO, RUIDO, ANIMAL, FUGA, FOTOGRAFIA, TERMINAR.

Para comandos teremos quatro constantes: PROCURAR, APROXIMAR, OBSERVAR, FOTOGRAFAR.

Para o personagem foi necessário criar duas classes: **ControloPersonagem** e **Personagem**.

Na classe **ControloPersonagem**, tal como o nome diz, vamos ter tudo o que é necessário para poder controlar a personagem no jogo. Aqui, definimos os estados que a personagem pode tomar, as ações que irá realizar e todas as transições possiveis para cada estado através de uma DSL (*domain-specific language*).

```
//Definição das transições através de uma DSL (domain-specific language)

/**
 * No contexto do nosso jogo, quando este se inicia a personagem fica numa situação de procura de animais.
 * Quando detecta algum ruído aproxima-se e fica em inspecção da zona, procurando a fonte do ruído.
 * Quando volta a haver silêncio a personagem volta a uma situação de procura de animais.
 * Quando detecta um animal a personagem aproxima-se e fica em observação. Caso o animal continue presente,
 * a personagem observa o animal e fica preparada para o registo, se ocorrer a fuga do animal
 * a personagem fica em inspecção da zona, à procura de uma fonte de ruído. Na situação de registo,
 * se o animal continuar presente fotografa-o, caso ocorra a fuga do animal ou a personagem
 * tenha conseguido uma fotografia do animal, a personagem fica novamente numa situação de procura.
 */

//Transições possiveis para o estado procura
procura
        .transicao(EventoJogo.SILENCIO, procura, procurar)
        .transicao(EventoJogo.ANIMAL, observacao, aproximar)
        .transicao(EventoJogo.RUIDO, inspeccao, aproximar);
```

Figura 9: Exemplo de transição de estado

A classe **Personagem** irá apenas representar a personagem do jogo, recebendo um novo ambiente e criando a personagem com o ControloPersonagem.

```
//classe Personagem que irá representar a personagem do jogo, estende a classe Agente
public class Personagem extends Agente {

    //construtor da classe que recebe uma instância de AmbienteJogo como parâmetro e
    //chama o construtor da classe Agente através do super(), passando como argumentos a instância de ambiente
    //e uma nova instância de ControloPersonagem
    public Personagem(AmbienteJogo ambiente){
        super(ambiente, new ControloPersonagem());
    }

}
```

Figura 10: Classe Personagem

3.1.5 Máquina de Estados

Esta parte irá ter tudo oque é necessário para traduzir o funcionamento de uma máquina de estados para código.

Uma máquina de estados é um modelo computacional que descreve o comportamento de um sistema através de um conjunto de estados, transições entre estados e ações associadas a essas transições.

Estados representam as condições em que um sistema pode ter, **transições** representam as mudanças de estado que ocorrem quando determinados eventos acontecem, **eventos** são acontecimentos que levam a transições entre estados, representam mudanças no ambiente ou entrada de dados e **ações** são o resultado de uma transição de estado.

Fizemos então o código, seguindo o UML dado pelo docente.

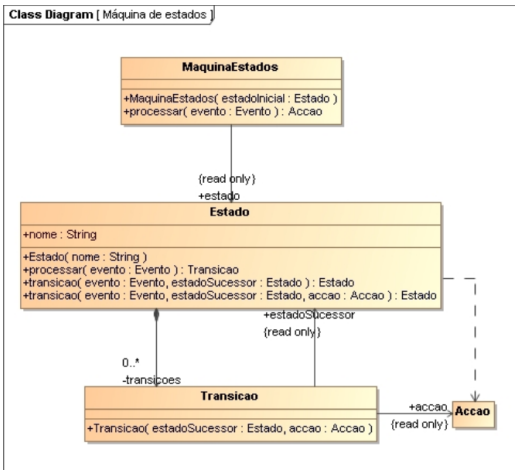


Figura 11: UML da Maquina de Estados

Portanto, na classe **MaquinaEstado** iremos ter um método que irá processar um evento e fazer uma transição do estado.

Na classe **Estado** temos os métodos necessários para processar um estado e fazer a sua transição, e na classe **Transição** será onde iremos obter o estado seguinte e a ação associada à transição.

```java
/**
 * Para os métodos "transicao" utilizamos uma implementação polimorfica, isto é
 * com o uso do polimorfismo podemos ter vários métodos com o mesmo nome definidos dentro da mesma classe,
 * desde que eles tenham assinaturas diferentes.
 * No caso deste código os seguintes métodos têm parâmetros diferentes.
 */

//método que chama o método transição para os eventos sem ação
public Estado transicao(Evento evento, Estado estadoSucessor){
    return transicao(evento, estadoSucessor, accao:null);
}

//método que coloca o evento e a transicao no Hashmap "transicoes"
public Estado transicao(Evento evento, Estado estadoSucessor, Accao accao){
    transicoes.put(evento, new Transicao(estadoSucessor, accao));
    return this;
}
```

Figura 12: Métodos de transição de Estado

3.1.6 Funcionamento do Jogo

Realizáda a implementação podemos ver o funcionamento do jogo na consola.

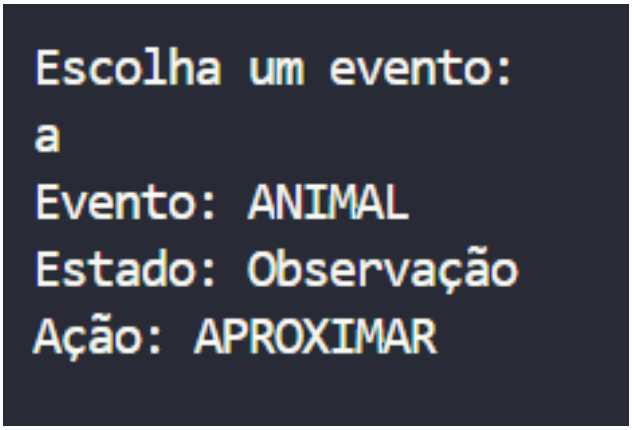


Figura 13: Funcionamento do Jogo da parte 1 na consola

Nesta caso o utilizador escolheu o evento ”Animal”onde o agente detetou um animal, sendo o estado correspondente a ”Observação”e a ação será ”Aproximar”. No conceito do jogo, isto quer dizer que a personagem detetou um animal e vai observar e tentar se aproximar para futuramente poder tirar foto.

3.2 Parte 2

Para esta segunda parte do projeto, desenvolvida em *Python*, foram feitas várias bibliotecas, as quais serão mencionadas nos pontos seguintes. Além disso, foi nos dado acesso a uma biblioteca já implementada pelos docentes da cadeira, denominada SAE, ou Simulação de Ambiente de Execução. Esta biblioteca permite visualizar o ambiente da nossa simulação e os seus elementos, como ilustrado na figura 14 (o agente é representado por um círculo amarelo, os alvos por quadrados verdes e os obstáculos por quadrados cinzentos).

Na parte direita da janela, encontra-se o elemento ”visão”do agente, que mostra os alvos e obstáculos nas direções norte, sul, este e oeste.

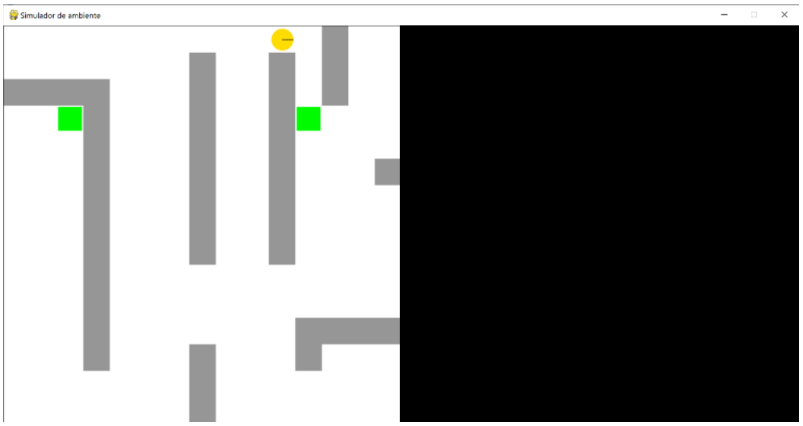


Figura 14: Janela da Simulação da Parte 2

3.2.1 Componentes Realizados

De modo a realizar esta parte do trabalho foram criadas, primeiramente, duas bibliotecas essenciais: ECR e *controlo_react*.

3.2.2 Biblioteca ECR

A primeira biblioteca implementada foi a ”ECR”(Esquemas Comportamentais Reativos), onde implementamos as três primeiras classes: **Reaccão**, **Estimulo**,**Resposta**.

Estas três classes são feitas para poder representar uma associação Estímulo - Resposta, onde uma perceção do agente vai ativar a deteção de um estimulo que, por sua vez, irá ativar uma resposta (dependendo da itensidade) e realizar uma ação.

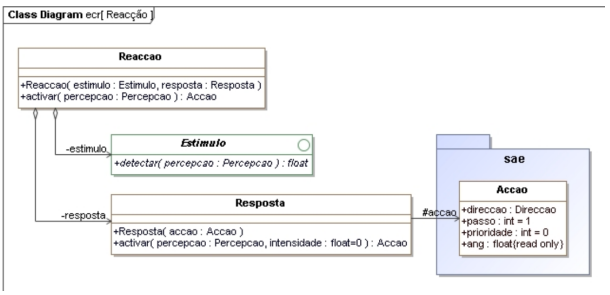


Figura 15: Biblioteca ECR

O nosso objetivo desta parte do projeto é ter um agente com controlo reativo, ou seja, tem uma perceção e com essa perceção efetua uma ação, ou seja, não ”pensa”, só age.

O modelo de interação seguinte descreve o funcionamento da associação Estímulo - Resposta.

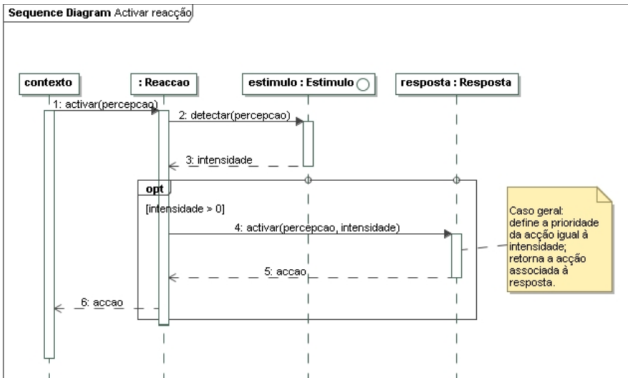


Figura 16: Modelo de Interação

O agente tem 3 tipos de comportamentos possiveis: aproximar do alvo, evitar obstáculos e explorar. Estes 3 comportamentos vão ser organizados através de uma hierarquia, onde o aproximar do alvo fica em primeiro lugar, o evitar obstáculos em segundo e o explorar em terceiro. Uma hierarquia refere-se à organização de classes numa estrutura de herança, a herança permite que uma classe herde atributos e métodos de outra classe.

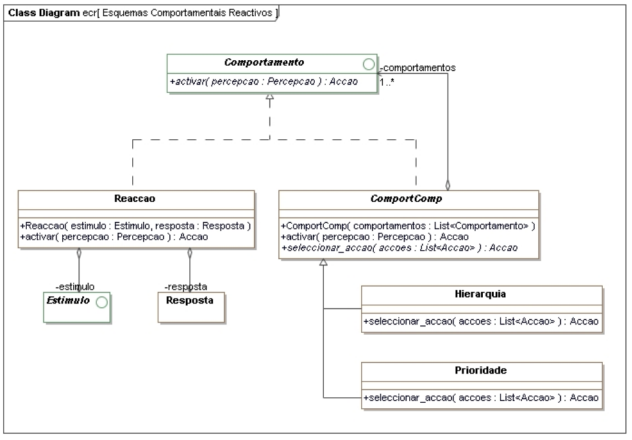


Figura 17: UML da biblioteca ECR

Para podermos então ter este sistema de comportamentos, foi necessário criar a classe **Hierarquia**, a classe **Comportamento**, uma classe **ComportComp** e uma classe **Prioridade**.

O **Comportamento** será uma interface que define a funcionalidade geral de um comportamento e a classe **ComportComp** será a implementação do mecanismo base de um comportamento composto.

A classe **Prioridade** é necessária porque para cada ação temos uma prioridade e é através desta prioridade que as ações são seleccionadas. Num sistema de processamento de eventos é selecionada primeiro a resposta com maior prioridade.

3.2.3 Biblioteca controlo_react

Para podermos ter um comportamento reactivo, temos de criar uma nova biblioteca.

Na biblioteca *"controlo_react"*, vamos implementar as nossas reações: **Aproximar**, **Evitar**, **Explorar** e **Resposta**.

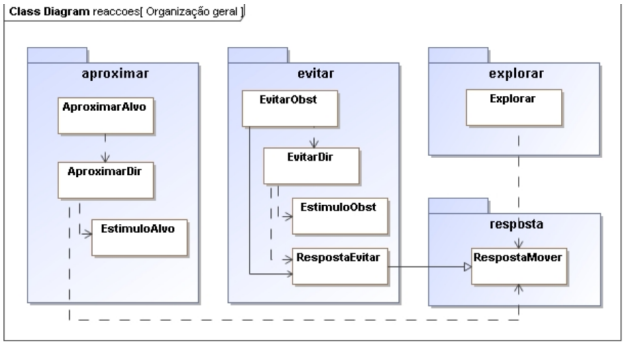


Figura 18: Organização das Reações

São estas respostas que vão permitir ao agente se comportar no ambiente, movendo-se no espaço nas quatro direções possiveis (NORTE, SUL, ESTE, OESTE).

3.2.4 Funcionamento do Programa

Uma vez implementada a biblioteca ECR e *controlo_react*, podemos testar o nosso programa criando uma nova simulação.

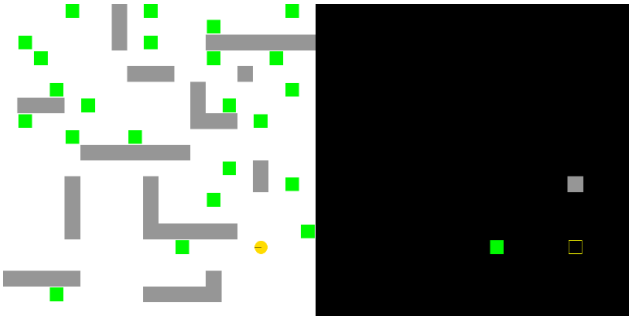


Figura 19: teste.py

4 Conclusões

Com este trabalho concluí-se que a inteligência artificial representa um campo fascinante na programação, com um papel vital no presente e no futuro. Ela impulsiona a inovação e o desenvolvimento de novas soluções tecnológicas que moldam o mundo à nossa volta.

Esta cadeira foi um bom ponto de partida para explorar o universo da inteligência artifical, algo que pretendo aprofundar no futuro. Além disso, aprendemos valiosas práticas de desenvolvimento de código, como evitar a repetição de código, entre outros aspetos importantes.

5 Bibliografia

Referências

[1] Morgado, L, *Introdução à inteligência artificial*, 2024.

[2] Morgado, L, *Introdução à engenharia de software - Parte 1, 2 e 3*, 2024.

[3] Morgado, L, *Arquitectura de Agentes Reactivos - Parte 1*, 2024.