INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

ÁREA DEPARTAMENTAMENTAL DE ENGENHARIA DE ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES

Licenciatura em Engenharia Informática e Multimédia

Inteligência Artificial para Sistemas Autónomos

Documento Síntese 1

2018/2019

ALUNOS:

39275, Ana Sofia Oliveira

ÍNDICE

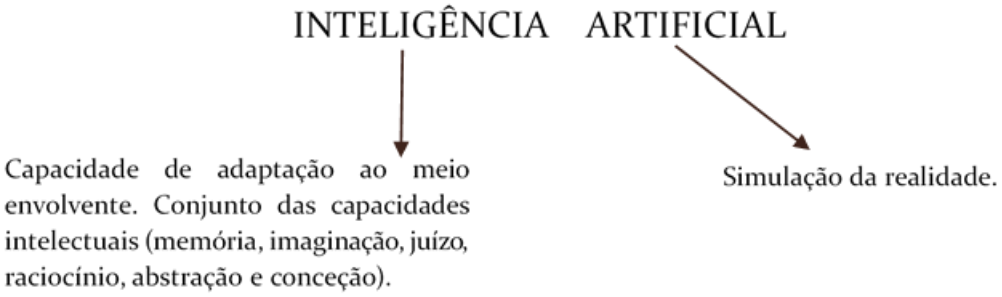
1 Enquadramento Teórico

1.1 Inteligência Artificial

"***Artificial Intelligence**, or **AI**, is the field that studies the synthesis and analysis of computational agents that act intelligently."*

[Poole & Mackworth, 2010]

A inteligência artificial é a área de desenvolvimento responsável pelo estudo e contrução de sistemas com capacidades semelhantes a um ser inteligente.



1.1.1 Processos base para a definição de Inteligência Artificial

No limite, a inteligencia artificial simula a inteligencia de qualquer ser real. Assim, é necessário definir processo base inerentes a esse processo de inteligencia.

Para que seja gerada uma acção, qualquer ser observa, interpreta o resultado da sua observação e gera uma resposta adquada. Assim, definimos os conceitos Percepção, Processamento e Atuação.

- Percepção: responsável por observar o meio ambiente, interpretando um estímulo a ser processado.
- Processamento: responsável por processar o estímulo proviniente da percepção e ponderar a resposta possível a ser dada. Pode subdividir-se em duas vertentes internas: construção do modelo do mundo e deliberação.
- Atuação: reponsável pela execução da acção ponderada no processamento.

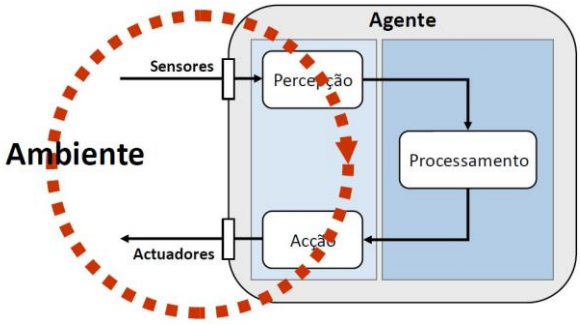


Fig 1 – Realimentação de sistema autonomo

1.1.2 Paradigmas

1.1.2.1 Paradigma Simbólico

Resulta de processos computacionais sobre estruturas simbólicas. A construção de significado está intrinsecamente associadas à relação entre diversos elementos.

Inteligências é obtida através da combinação de processos/ regras definidas e de um conjunto de estruturas simbólicas.

1.1.2.2 Paradigma Conexionista

Resulta das propriedades emergentes de interações de um número elevado de unidades elementares de processamento.

Baseado nos sistemas neuronais e nas suas ligações/ conecxões. Recorre a memórias associativas e reconhecimento de padrões para gerar um comportamento.

1.1.2.3 Paradigma Comportamental

Resulta da dinamica comportamental individual e/ou conjunta entre multiplos sistemas com diferentes escalas de organização.

Baseado em comportamentos já adquiridos e aprendidos por parte de um determinado agente.

1.2 Modelo Computacional

Um modelo computacionar consiste na definição comportamento de um sistema. Um sistema é constituido por um determinado conjunto de entradas que sofrem uma transformação e é gerado um conjunto de saidas.

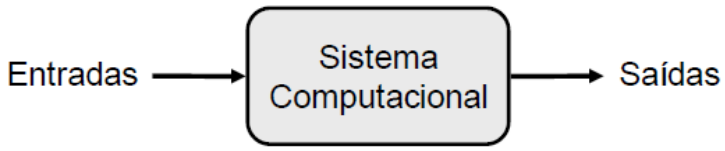


Fig 2 – Modelo Computacional

Pode estar organizado de duas formas:

- No espaço: Corresponde à estrutura do sistema.
- No tempo: Corresponde à dinâmica do sistema.

Aplicando um modelo simbólico, podemos referenciar a função de transformação é descrita com base em duas funções:

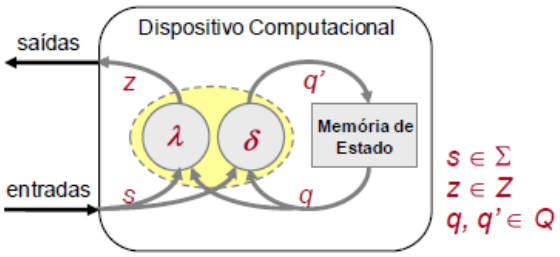


Fig 3 – Funções de Transformacão

- Função de transição de estado: dada pela relação entre o conjunto das entradas e o estado atual do sistema, atualizando o estado interno do sistema (com memória).

$$\delta = Q \times \sum \quad Q$$

- Função de saída: dada pela relação entre o conjunto das entradas e o estado atual do sistema, gerando as saídas do sistema (sem memória).

$$\lambda = Q \times \sum \quad Z$$

1.3 Engenharia de Software

1.3.1 Métricas

1.3.1.1 Complexidade

"Consigo explicar o sistema? Tenho domínio sobre ele?"

Métrica responsável por determinar o estado de organização de um sistema. Resulta do caracter combinatório das partes de um determinado sistema.

Existem dois tipos de complexidades: Organizada e Desorganizada.

A complexidade organizada é, normalmente, a pretendida para o sistema em análise dado garantir que um sistema contem apenas as partes estritamente necessárias para a sua subsistência.

A complexidade desorganizada promove um estado com maior estropia no sistema e, por consequência, uma perda de controlo sobre as partes que constituem esse sistema.

1.3.1.2 Acoplamento

"Cada parte do sistema tem apenas as ligações estritamente necessárias?"

Métrica responsável por determinar o grau de interdependência entre as partes de um sistema.

Existem diversos tipos de acoplamento, dois dos quais iremos falar com maior detalhe: Estrutural e Funcional.

O acoplamento estrutural define as interligações estruturais entre as partes de um sistema. Por exemplo, a porta está ligada à ombreira através das dobradiças. A porta, a ombreira e as dobradiças são elementos estruturais deste sistema.

O acoplamento funcional define as interligações funcionais entre as partes de um sistema. Por exemplo, uma chave depende de uma fechadura, da mesma forma que a fechadura depende da chave, para poder cumprir a sua função. Embora a chave e a fechadura sejam elementos estruturais neste sistema, dependem de uma interligação funcional para que possam cumprir o seu propósito/função.

Quando menor o acoplamento, menor a interdependencia entre as partes de um determinado sistema, logo menor a complexidade associada a esse sistema. Por consequência maior o domínio de conhecimento sobre esse mesmo sistema.

1.3.1.3 Coesão

"Cada sub-parte do sistema cumpre apenas a sua função?"

Métrica responsável pelo nível de coerência funcional.

Define o número de sub-partes necessárias para que o sistema cumpra o seu propósito. Indica até que ponto cada parte está definida para o seu propósito ou se tem múltiplos propósitos.

Quanto menor o nível de coesão maior o número de ligações entre as partes de um sistema (maior acoplamento). Sempre que uma das partes seja alterada será necessário alterar todas as restantes que de si dependam. Assim, quanto menor o nível de coesão, maior o acoplamento e, por consequência, maior a complexidade.

Quanto maior o nível de coesão menor o número de ligações interdependentes das restantes partes do sistema (menor acoplamento). Desta forma, sempre que alterada uma parte do sistema apenas é necessário alterar as partes que obrigatóriamente dependam desta. Assim, quanto maior o nível de coesão, menor o acoplamento e, por consequência, menor complexidade no sistema.

2 Enquadramento Prático

2.1 Enunciado

“ 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 objectivo impedir que inimigos entrem numa zona à sua guarda.

Quando o jogo se inicia a personagem fica a patrulhar a zona. Quando detecta algum ruído aproxima-se e fica a inspeccionar a zona, procurando a fonte do ruído. Quando volta a haver silêncio a personagem continua a patrulhar a zona.

Quando detecta um inimigo a personagem aproxima-se e fica a proteger a zona, avisando o inimigo para este se retirar. Nesta situação, se o inimigo fugir a personagem fica a inspeccionar a zona, à procura de uma fonte de ruído. Caso o inimigo persista, a personagem defende-se e passa a combater o inimigo atacando-o. Na situação de combate, caso ocorra a fuga do inimigo ou a vitória da personagem, esta fica novamente a patrulhar, caso ocorra a derrota da personagem, o jogo é reiniciado.

A interacção com o jogador é realizada em modo de texto.”

2.2 Concretização

Conforme descrito no enunciado, iremos implementar um personagem virtual que interage com um jogador humano.

Através da definição de sistema de um agente autónomo (Fig.1), conseguimos encontrar algumas semelhanças com o nosso personagem e começar a definir conceitos. Assim:

- **Agente –** Corrresponde ao nosso Personagem. Contempla todo o processamento interno do personagem. Constituido pela percepção dos eventos do sensor, processamento interno e tomada de ação via atuadores.
- **Ambiente –** Corresponde ao nosso Ambiente. Contempla todo o processamento externo ao personagem. Constituido por 2 componentes:
  - **Sensores:** Responsáveis pelo input de eventos do ambiente. Ou seja, pelo input dado pelo jogador humano ao jogador vitual.
  - **Atuadores:** Reponsáveis pelo output de eventos para o ambiente, sendo a aplicação da ação no ambiente. Ou seja, output sob a forma de acção gerada para o ambiente.

Sabendo que iremos construir um jogo que irá conter um personagem virtual que, por sua vez, irá interpretar e agir num ambiente, teremos que ter as classes Jogo, Personagem e Ambiente. Posto isto, conseguimos estrapular uma interação funcional entre estas classes (Fig. 4).

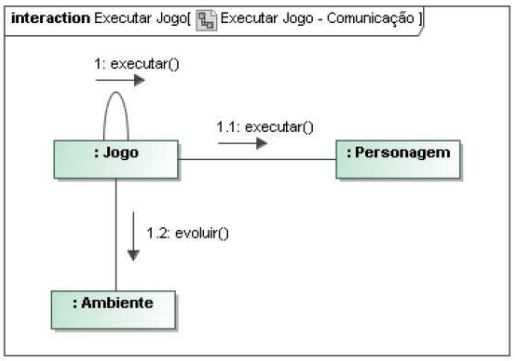


Fig 4 – Diagrama de comunicacão executar Jogo

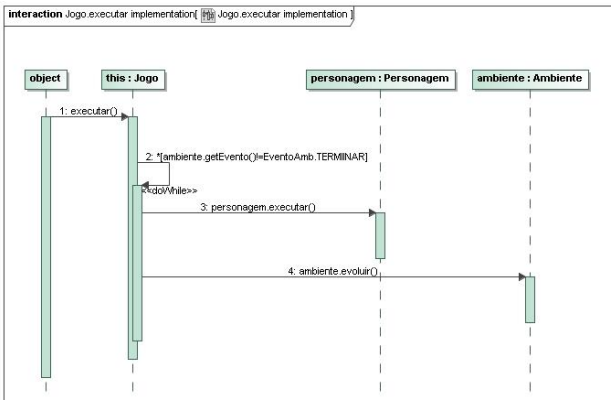


Fig 5 – Diagrama de atividades executar Jogo

Na Fig. 5 encontra-se promenorizada a interação funcional entre as classes decritas acima, demonstrada na Fig. 4. Leia-se: o Jogo irá fazer execuar sobre si mesmo, depois irá executar o Personagem e, posteriormente, irá evoluir o Ambiente. Este processo será ciclico até que o jogo termine (evento TERMINAR).

Podemos, agora sim, iniciar a construção estrutural do código de acordo com o UML da figura 6.

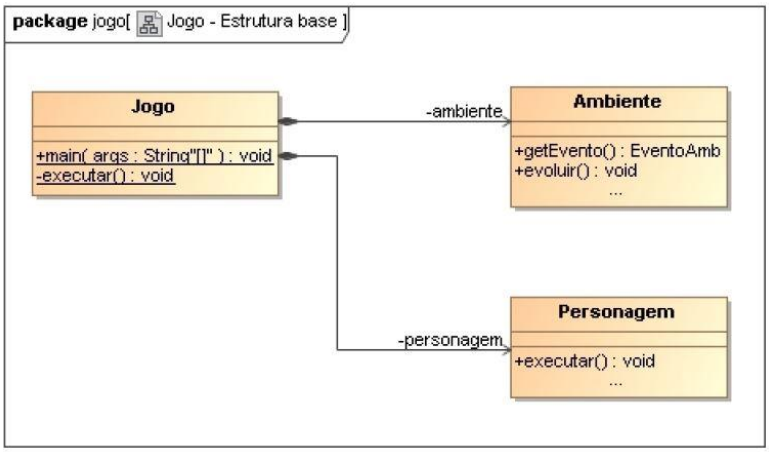


Fig 6 – Diagrama de Classes jogo

Após estar definido o código estrutural iniciamos a implementação funcional do executar do Jogo, de acordo com o apresentado anteriormente (Fig. 4 e 5).

Através dos processos base, mencionados na componente teórica deste relatório, sabemos que o nosso agente irá percepcionar um estímulo, processar esse estímulo numa ação e atuar no ambiente (Fig. 1). Podemos agora complementar o nosso UML estrutural para o UML apresentado na figura abaixo (Fig. 8), bem como, podemos implementar o método executar do Personagem.

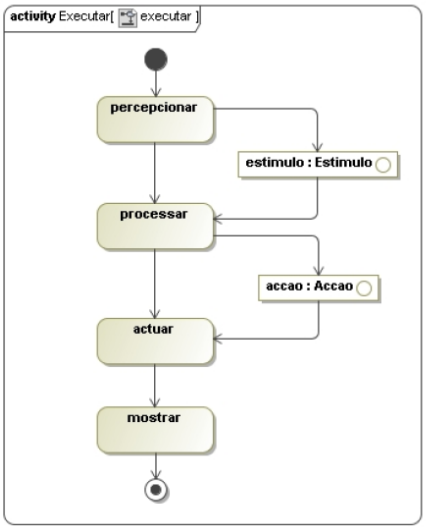


Fig 7 – Diagrama de atividades executar Jogo

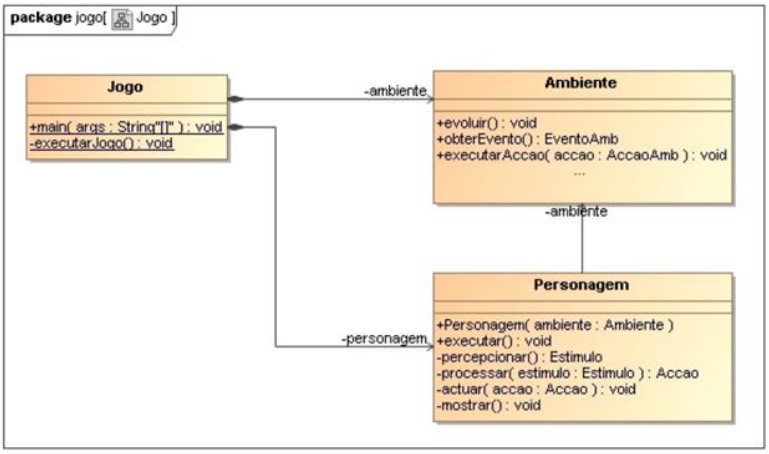


Fig 8 – Diagrama de Classes jogo

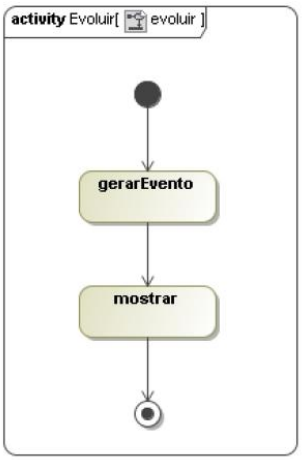
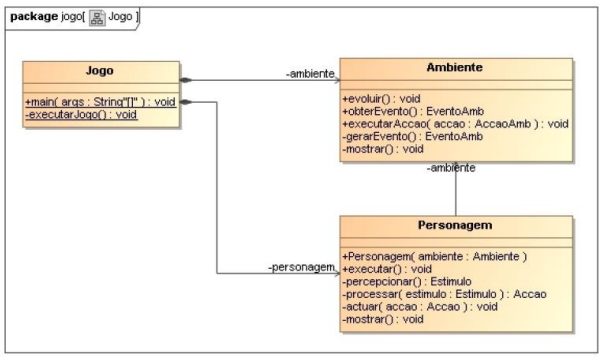


Fig 9 – Diagrama de atividades evoluir Ambiente

Observando a estrutura da classe ambiente podemos concluir que evoluir o ambiente é uma amostragem textual do evento gerado, portanto terá que ser gerado um evento e de seguida teremos que mostrar esse evento ao utilizador. Com este raciocínio constuimos o diagrama de atividades do método evoluir e completamos o digrama de classes geral (Fig. 9).

Fig 10 – Diagrama de Classes jogo



Após termos definido a mecanica do jogo, precisamos de estruturar qual irá sem o comportamento do personagem. Para cada estado existem diversas reações possíveis, ou seja, cada estado terá que conter todas as possíveis reações sob a forma de estimulo-resposta. Estas regras de comportamento estimulo-resposta foram inseridas na componente reação, classe Reaccao. Para que essas reações possam fazer parte de um comportamento, criamos o comportamento hierarquico, classe ComportHierarq.

Sendo que cada estado tem mais que uma reação possível e o nosso personagem pode estar em diversos estados, como iremos ver mais à frente, necessitamos de construir também uma estrutura que suporte o comportamento do personagem numa máquina de estados. Assim, criamos o comportamento máquina estados, classe ComportMaqEst. Todo este raciocício deu origem ao diagrama apresentado na Fig. 11.

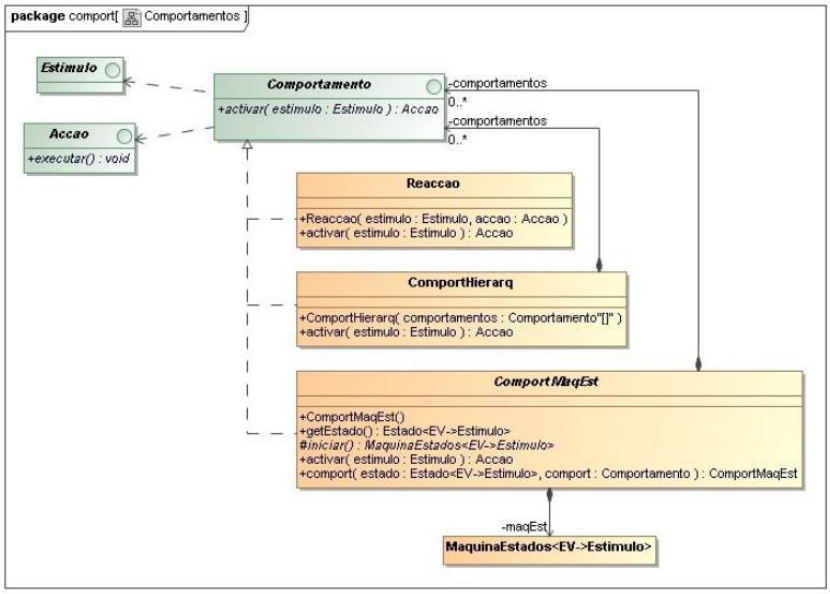


Fig 11 – Diagrama de Classes comport

De forma a manter o acoplamento baixo, a complexidade baixa e a coesão elevada, e visto que tanto a reação, o comportamento hierárquico e o comportamento da máquina de estados teriam um método comum “activar” , implementamos a classe abstrata Comportameto que define um contrato sobre a implementação do método ativar.

Assim, cada tipo de comportamento implementa a suas especificidades na ativação:

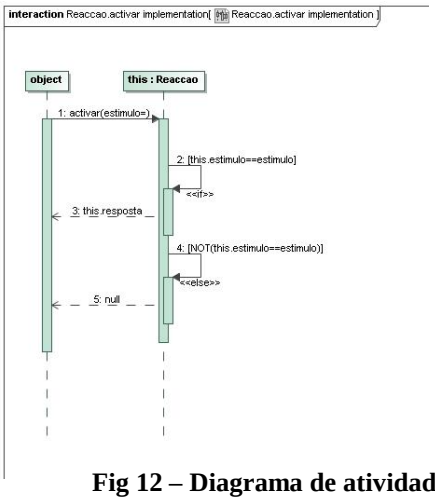


Fig 12 – Diagrama de atividades activar Reaccao

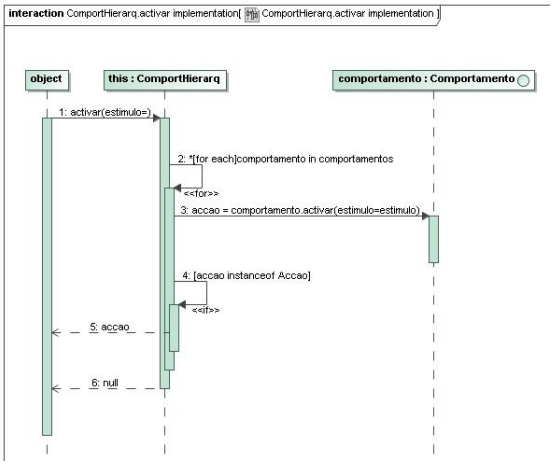


Fig 13 – Diagrama de atividades activar ComportHierarq

NOTA: A implementação funcional da Classe ComportMaqEst será apresentada na sequecia do relatório.

Recorrendo novamente à leitura do enunciado e ao modelo computacional (Fig. 3), anuímos que teremos:

$\sum$ = {ruido, silencio, inimigo, fuga, vitoria, derrota, terminar}

Q = {Patrulha, Inspeção, Defesa, Combate}

Z = {patrulhar, aproximar, avisar, defender, atacar, procurar, iniciar}

Sendo,

Função de transição de estado: $\delta = Q \times \sum \rightarrow Q$

Função de saida: $\lambda = Q \times \sum \rightarrow Z$

Para encontrar o novo estado dependendo do estado anterior em função do estímulo, foi definida a tabela representativa da função de transição de estado (δ):

	Patrulha	**Inspeção**	**Defesa**	**Combate**
ruído (r)	Inspeção	Inspeção	-	-
silêncio (s)	Patrulha	Patrulha	-	-
inimigo (i)	Defesa	Defesa	Combate	Combate
fuga (f)	-	-	Inspeção	Patrulha
victória (v)	-	-	-	Patrulha
derrota (d)	-	-	-	Patrulha

Definimos também uma relação entre o conjunto de entrada (∑) com o conjunto de saída (Z), obtendo a seguinte função de saída (λ):

$$\lambda = \begin{Bmatrix} ruído\ (r) \rightarrow aproximar \\ ruído\ (r) \rightarrow procurar \\ silêcio\ (s) \rightarrow patrulhar \\ inimigo\ (i) \rightarrow aproximar \\ inimigo\ (i) \rightarrow defender \\ fuga\ (f) \rightarrow procurar \\ vitória\ (v) \rightarrow patrulhar \\ derrota\ (d) \rightarrow patrulhar \\ terminar\ (t) \rightarrow patrulhar \end{Bmatrix}$$

Podemos então facilitar a interpretação através do diagrama de atividades representativo desta máquina de estados (Fig. 14).

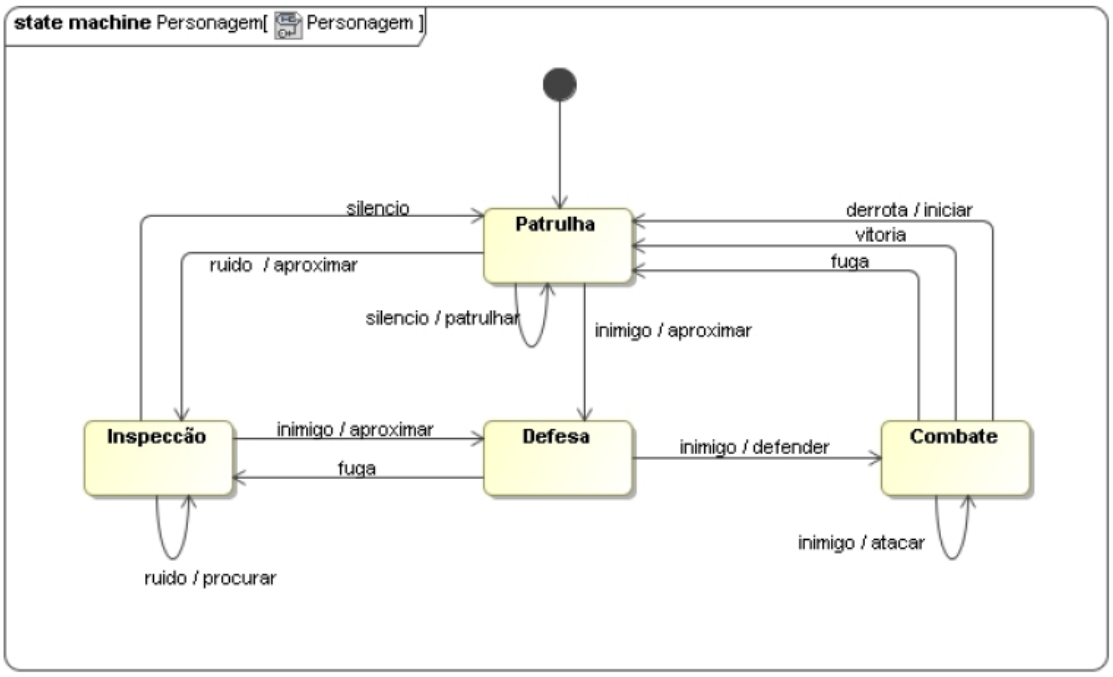


Fig 14 – Maquina de Estados

Será necessário construir uma estrutura de dados que suporte o modelo acima. Para isso, construimos as classes MaquinaEstados e Estado.

A classe Estado é responsável por tratar toda a informação inerente ao estado, ou seja, processar o estado e definir as transicções possíveis desse estado. A classe MaquinaEstados é responsável por processar toda a máquina de estado com todos os estados possíveis.

Assim, construimos o diagrama de classes que descreve a nossa estrutura de dados (Fig. 15).

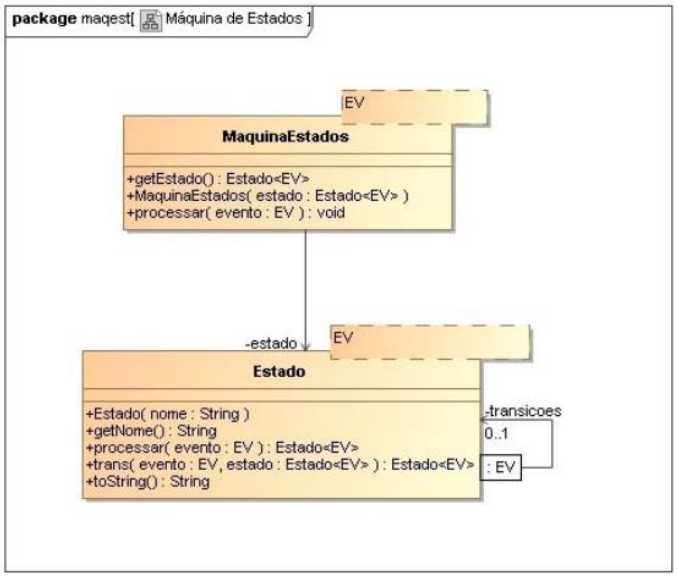


Fig 15 – Diagrama de Classes maqest

Passamos agora para a implementação do código estrutural e posteriormente, para o código funcional.

A função transicao da Classe Estado representa a codificação de uma transição possível para aquele estado. Assim, deverá receber como argumentos o evento e o próximo estado. Para simplificar a codificação de todas as transições possíveis para determinado estado, esta função retorna o próprio objeto. Desta forma, aumentamos a coesão e diminuimos o acoplamento na nossa implementação.

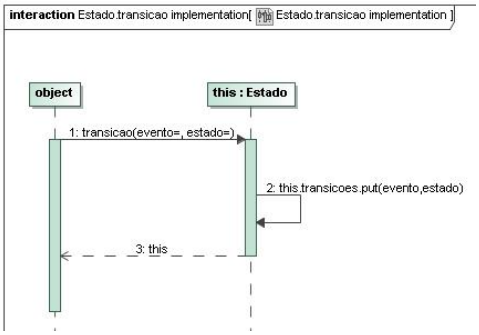


Fig 16 – Diagrama de atividades transicao Estado

Após estarem codificadas todas as transições na nossa maquina de estados, teremos de ter forma de obter o próximo estado gerado por input de um evento. Assim, criámos a função processar. Esta função recebe como augumento um dado evento e, mediante o estado em análise, retorna o próximo estado.

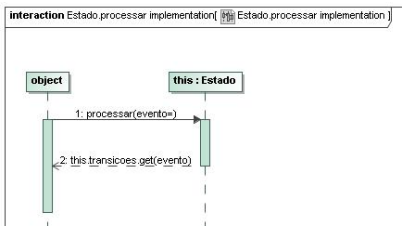


Fig 17 – Diagrama de atividades processar Estado

A classe MaqEst sendo responsável pela maquina de estado deverá ter o domínio do estado atual do sistema e de como processar o evento dado. Assim, também esta classe terá implementada uma função processar responsável por dado um evento retornar o próximo estado. Deverá ainda atualizar o estado atual da máquina de estados.

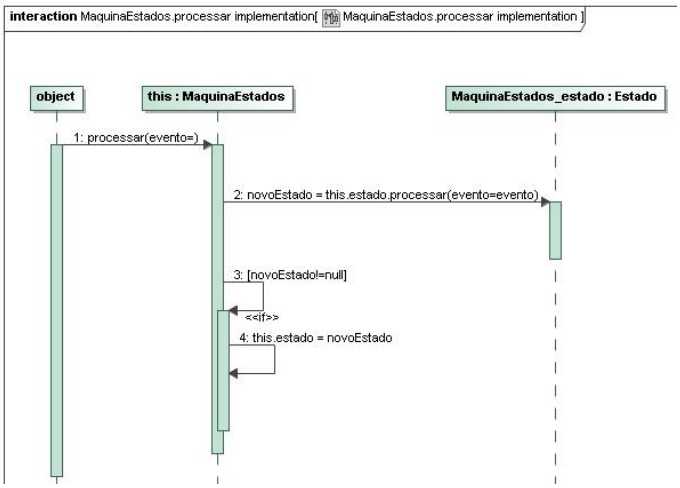


Fig 18 – Diagrama de atividades processar MaquinaEstados

Após a construção da nossa máquina de estado teremos que atualizar o comportamento baseado na mesma. Assim, a classe ComportMaqEste deverá conseguir ativar a maquina de estados, codificar comportamentos e obter o estado atual do sistema.

A função ativar do ComportMaqEst é responsável por ativar um determinado estimulo e atualizar a máquina de estados. Assim, deverá receber o estímulo e idenficar a açõa a tomar, e ainda fornecer esse estímulo como evento à máquina de estados para atualizar o estado atual do sistema.

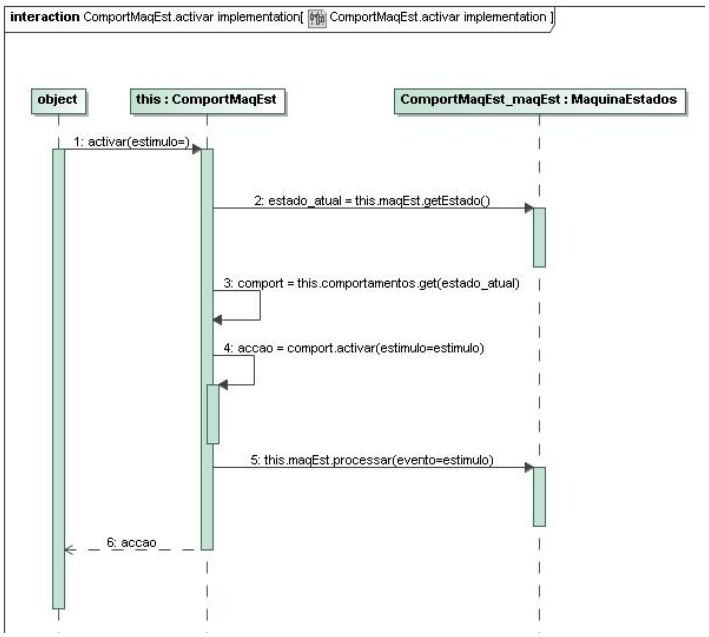


Fig 19 – Diagrama de atividades activar ComportMaqEst

A função comport da classe ComportMaqEst deverá codificar os comportamentos.

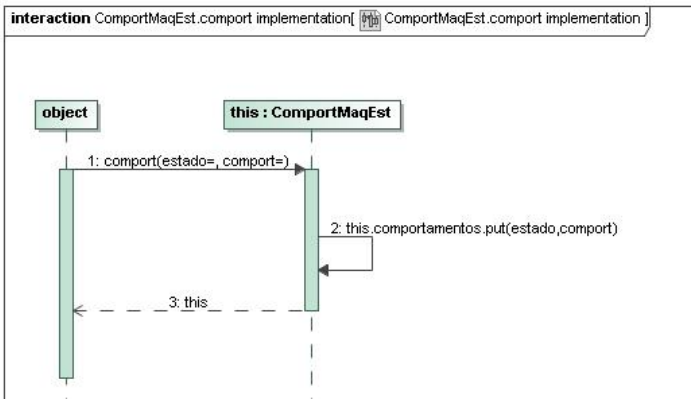


Fig 20 – Diagrama de atividades comport ComportMaqEst

Tendo todas as estruturas de dados para suporte do nosso projeto implementadas teremos agora que iniciar a implementação no domínio do problema.

Assim, implementamos todas as reações associadas a cada estado:

- Patrulhar:
  - Reação com estimulo: Ruido e acção Aproximar;
  - Reação com estimulo: Silencio e acção Patrulhar;
  - Reação com estimulo: Inimigo e acção Aproximar.

- Inspecionar:
  - Reação com estimulo: Inimigo e acção Aproximar;
  - Reação com estimulo: Ruido e acção Procurar.

- Defender:
  - Reação com estimulo: Inimigo e acção Defender.

- Combater:
  - Reação com estimulo: Inimigo e acção Atacar;
  - Reação com estimulo: Derrota e acção Iniciar.

Criámos a classe ComportPersonagem que irá ser responsável por codificar os diversos comportamentos do nosso personagem.

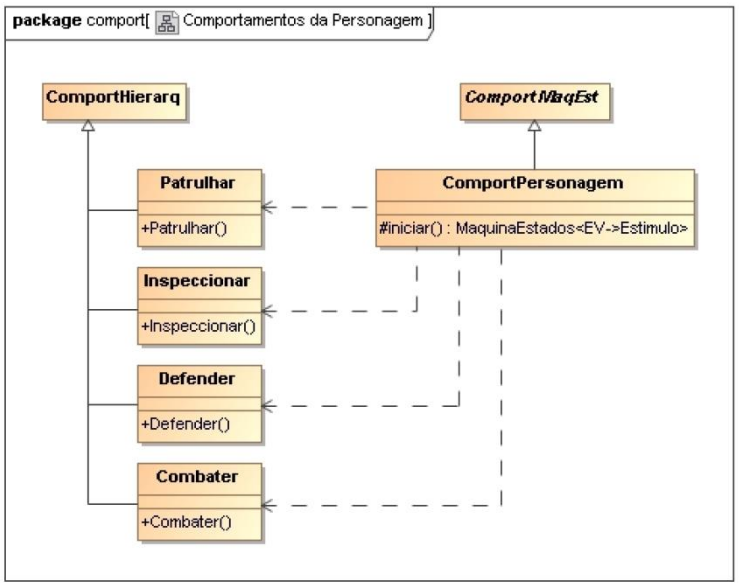


Fig 21 – Diagrama de Classes comport

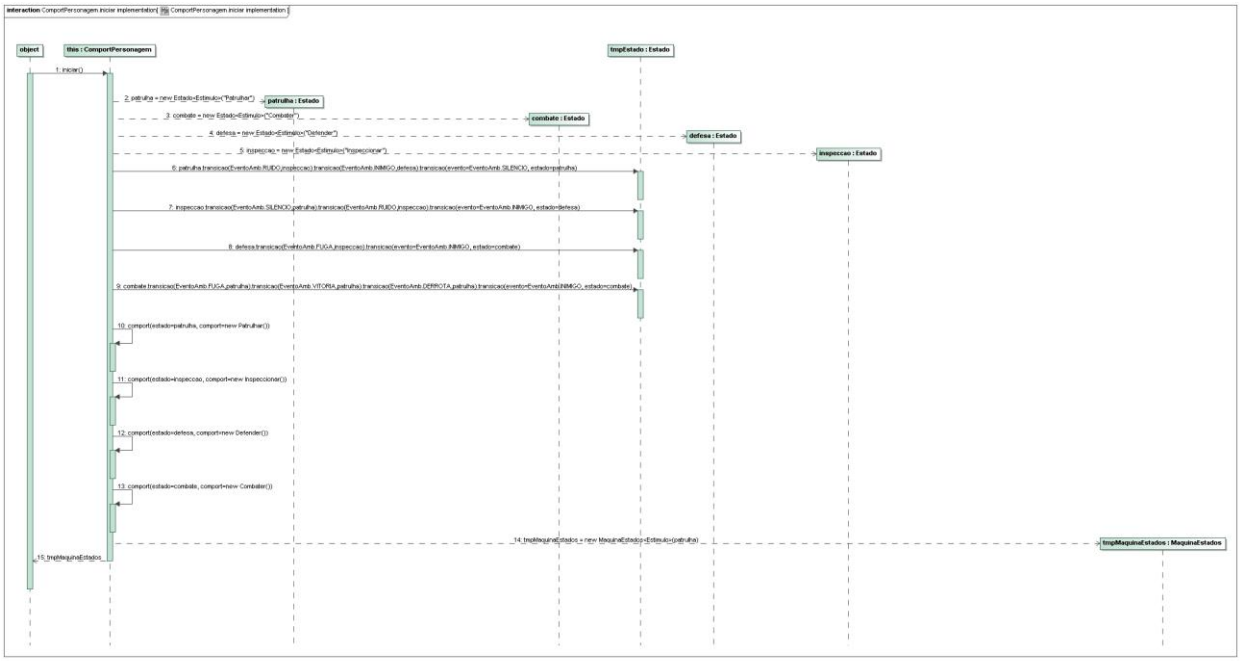


Fig 22 – Diagrama de atividades
iniciar ComportPersonagem

Poderemos agora implementar as funções que são intrinsecas ao personagem, nomeadamente, o percepcionar, o processar e o actuar.

A função percepcionar devera recolher um estimulo provieniente do ambiente e retornar o mesmo para que seja processado.

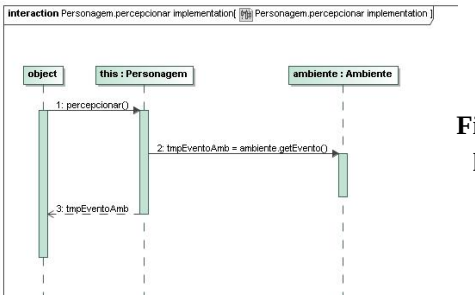
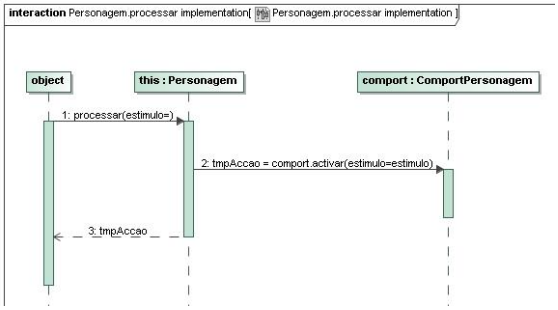


Fig 23 – Diagrama de atividades percepcionar Personagem

A função processar deverá receber, como argumento, o estimulo dado pelo percepcionar, ativar o comportamento do personagem e gerar uma ação.

Fig 24 – Diagrama de atividades processar Personagem



A função actuar deverá receber a acção gerada no processamento do estimulo e executar essa acção.

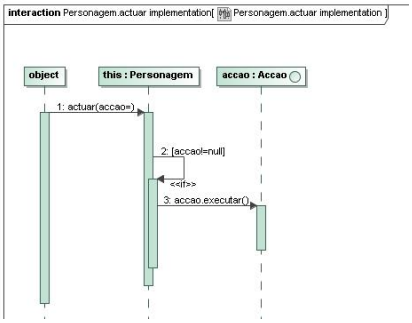


Fig 25 – Diagrama de atividades actuar Personagem

3 Bibliografia

- [Russel & Norvig, 2003] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2nd Edition, Prentice Hall, 2003.
- [Anabela Simões & Ernesto Costa] Inteligência Artificial, 2008.
- [Sérgio Guerreiro] Introdução à Engenharia de Software, 2015.