*UNIVERSIDADE NOVA DE LISBOA*
FCT - Departamento de Engenharia Electrotécnica
Secção de Robótica e Manufatura Integrada

# Integração de Sistemas

2016 / 2017

Trabalho 3

Sistemas Multiagente (JADE) e XML

Duração: 4 aulas acompanhadas por docente

Entrega: 4 de Junho de 2017

## 1. INTRODUÇÃO

A definição de agente não é consensual. Desta forma, surgem múltiplas definições que variam de acordo com a cultura dos diversos proponentes. A noção de que um agente deve ter alguma autonomia é um dos poucos pontos de contacto entre as diferentes definições. No âmbito deste trabalho considerar-se-á que um agente é um programa que, sobre um determinado ambiente onde está situado, é capaz de tomar ações autónomas com base nos seus objetivos e na perceção de alterações ambientais, no sentido de cumprir o objetivo para que foi desenvolvido. Esta definição suporta a existência de um mundo não determinista pelo que a informação que o agente recolhe do meio é uma parcela da realidade. Assim, o agente deve estar preparado para lidar com informação incerta e incompleta. De acordo com esta definição um agente deve denotar algumas das seguintes características:

- Autonomia: um agente pode operar em autocontrolo sem interação de terceiros;
- Sociabilidade: os agentes podem interagir com outros agentes e entidades cumprindo as regras sociais que pautam a sua interação;
- Reatividade: os agentes são capazes de reagir a mudanças no ambiente.
- Pró-Atividade: os agentes não são entidades puramente reativas, isto é, têm um comportamento orientado a objetivos podendo tomar iniciativa durante uma operação;
- Adaptabilidade: os agentes têm a capacidade de se adaptarem a alterações no ambiente.

As interações com o meio ambiente (mundo e outros agentes) são fundamentais no contexto dos sistemas multiagente. Neste sentido, um sistema multiagente pode ser definido como uma rede de “solucionadores” de problemas que cooperam na resolução de um problema global do qual apenas conhecem partes. Está implícita a noção de que uma sociedade de agentes fornece um nível de funcionalidade em que o todo é maior do que a soma da contribuição das partes. Este conjunto de características torna os conceitos de agente e sistema multiagente poderosas ferramentas de modelação de sistemas de natureza distribuída.

# 2. Apresentação do Problema

## *2.1. Jogo 2D baseado num Sistema Multiagente*

Neste trabalho é pedido que se modele e implemente um pequeno jogo 2D. O jogo decorre num *tilemap* com dimensão *n*m*, onde cada *tile* representa uma entidade diferente, nomeadamente:

- **Goblin** – Inimigo básico, é uma criatura que ataca todos os jogadores que tentem ocupar ou usar o mesmo tile onde o Goblin se encontra.
- **Healer** – Cura o jogador, restaurando os seus *Health Points* (HP), ou adicionando um bónus aos mesmos.
- **Trap** – Uma armadilha no chão que mata o jogador se este se mover para esse *tile*.
- **Treasure** – Representa o objetivo do jogo. O jogador ganha ao encontrar o tesouro.

Uma visão geral do mapa bem como da representação de cada uma destas entidades pode ser observada na Figura 1.

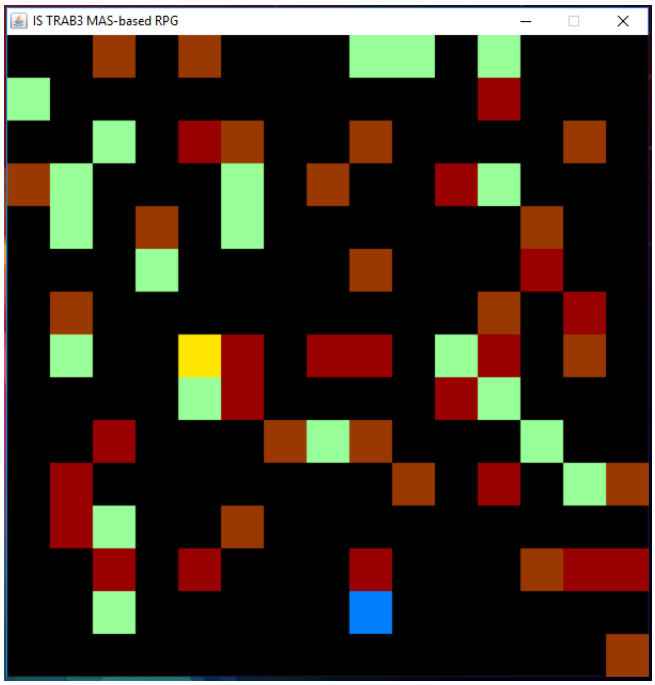


*Figura 1 - Vista geral do mapa de jogo.*

*Legenda: Vermelho - Goblin, Verde - Healer, Castanho - Trap, Amarelo - Treasure, Azul - Player.*

O jogador, representado a azul, pode assim mover-se ortogonalmente, ou seja, na direção horizontal ou vertical. Adicionalmente, antes de se movimentar o jogador pode ainda procurar vestígios (*tracks*) no *tile* para o qual está virado, podendo assim identificar que tipo de entidade nele presente.

Desta forma, para além de se movimentar, o jogador é também capaz de atacar ou usar as entidades presentes em determinado *tile*. O resultado de cada uma das interações resultantes das jogadas possíveis está definido na Tabela 1.

*Tabela 1- Resultado das interações do Player com as diferentes entidades*

| ***Character*/Agente** | **Mover** | **Atacar** | **Usar** |
|---|---|---|---|
| ***Goblin*** | O jogador é atacado pelo Goblin e perde HP. Se a HP do jogador acabar, perde. Caso contrário volta para o *tile* onde se encontrava. | O jogador ataca o Goblin. Se o Goblin for derrotado o jogador conquista o seu *tile* no mapa. Caso contrário permanece onde se encontra. | O jogador é atacado pelo Goblin e perde HP. Se o não perder o jogo, fica no mesmo *tile* em que estava anteriormente. |
| ***Healer*** | Desaparece e o jogador ocupa a sua posição no mapa. | O jogador perde o jogo. | O Healer dá HP ao jogador e desaparece do mapa. O jogador ocupa a sua posição |
| ***Trap*** (incluí zona fora do mapa) | O jogador perde o jogo. | O jogador perde o jogo. | O jogador perde o jogo. |
| ***Treasure*** | O tesouro é realocado para outro tile. O jogador ocupa a sua posição no mapa. | O jogador perde o jogo. | O jogador ganha e o jogo termina. |
| ***Terrain*** | O jogador ocupa a nova posição no mapa. | O jogador ocupa a nova posição no mapa, mas perde pontos na arma. | O jogador ocupa a nova posição no mapa, mas perde pontos na arma. |

Existe ainda um tipo de agente adicional, nomeadamente o *WorldAgent*. Este agente tem como responsabilidade abstrair o ambiente de jogo, coordenando os pedidos dos jogadores e respetivas interações com os elementos no mapa. É também a entidade responsável pela atualização da interface gráfica.

O objetivo final deste projeto é que o agente responsável por abstrair o jogador seja capaz de operar de forma autónoma, adaptando dinamicamente as suas jogadas de acordo com a visão parcial do ambiente que o rodeia.

# 3. Troca de Mensagens

A estrutura que permite ao agente *Player* comunicar com o *WorldAgent*, bem como receber a atualização do seu estado após cada jogada, é fornecida pelo corpo docente, sob a forma de um *schema* num ficheiro "*.xsd.*". Este *schema* está representado na Figura 2.

```
<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://xml.netbeans.org/schema/characterSchema"
           xmlns:tns="http://xml.netbeans.org/schema/characterSchema"
           elementFormDefault="qualified">

    <xsd:complexType name="tPositionAndOrientation">
        <xsd:sequence>
            <xsd:element name="longitude" type="xsd:int" maxOccurs="1" minOccurs="1"/>
            <xsd:element name="latitude" type="xsd:int" maxOccurs="1" minOccurs="1"/>
            <xsd:element name="orientation" type="xsd:int" maxOccurs="1" minOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="tCharacter">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="description" type="xsd:string"/>
            <xsd:element name="healthPoints" type="xsd:int"/>
            <xsd:element name="weaponDurability" type="xsd:int"/>
            <xsd:element name="type" type="xsd:int"/>
            <xsd:element name="status" type="xsd:int"/>
            <xsd:element name="tracks" type="xsd:int"/>
            <xsd:element name="previousPAndO" type="tns:tPositionAndOrientation"/>
            <xsd:element name="currentPAndO" type="tns:tPositionAndOrientation"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="myCharacter" type="tns:tCharacter"/>

</xs:schema>
```

*Figure 2 – Schema definindo a estrutura de um Character*

Este *schema* será utilizado para serializar o objeto para uma String e assim poder ser trocado entre os agentes nas diversas mensagens.

# 4. Implementação

Na implementação pedida será utilizado o seguinte material:

- Linguagem JAVA no IDE Netbeans;
- Plataforma JADE;
- JAXB: utilizado para criar as classes *TCharacter* e *TPositionAndOrientation* através do *schema,* bem como para serializar objetos destes tipos;
- Windows 7, 8 ou 10 recomendado (também é possível em Linux);
- Código fornecido pelos docentes da cadeira;

# 5. Planeamento das Aulas

## *5.1. Aula 1 – Instalação e Tutorial*

1. Siga as instruções fornecidas pelo docente, instale e teste o JADE.
2. Implemente o tutorial de JADE fornecido pelo corpo docente.

   Tutoriais para ambos os passos podem ser encontrados na página da cadeira de Integração de Sistemas, no CLIP.

## *5.2. Aula 2 – Deployment de Agentes e Manipulação do DF*

1. Estude o código fornecido pelos docentes.

   a. Leia atentamente todo o código fornecido familiarizando-se com as funcionalidades implementadas e com os diferentes pontos a desenvolver.

2. Passar argumentos a um agente no JADE:

   a. No package "WorldAgent" e na classe com o mesmo nome, implemente o método **setup()** de forma a que o agente receba por argumento os valores de **maxLongitude**, **maxLatitude**, **numberOfHealers**, **numberOfGoblins**, e **numberOfTraps**.
   O agente deve verificar se recebeu exactamente 5 argumentos. Caso contrário o agente deverá abandonar a plataforma.

   b. Nas propriedades do projeto, selecione **Run** e defina os parâmetros "Main Class" e "Arguments" de forma a que ao correr o projeto, a plataforma JADE seja lançada (mostrando a interface gráfica), bem como um agente do tipo "WorldAgent". Os argumentos sugeridos para o agente são:
      i. maxLongitude = 10
      ii. maxLatitude = 10
      iii. numberOfHealers = 5
      iv. numberOfGoblins = 5
      v. numberOfTraps = 5

3. Registo no Directory Facilitator (DF) do JADE

   a. No package Common e na classe DFInteraction preencha o método static, "**RegisterInDF**" que permite qualquer agente inscrever-se no DF e aos seus serviços.

b. No package “WorldAgent”, na classe com o mesmo nome, adicione ao método **setup**() a chamada a “RegisterInDF”, implementado no ponto 3a, para que o agente WorldAgent se registe no DF.

4. Procura no Directory Facilitator (DF) do JADE

   a. No package Common e na classe DFInteraction preencha o método static, “**findAgents**” que permite procurar no DF por agentes que disponibilizem um determinado serviço.

   b. No package “Entities”, na classe “Character”, implemente o método **findWorldInDF**() utilizando o método desenvolvido no ponto 4a. Este método tem de procurar o agente mundo no DF, utilizando um template de pesquisa equivalente ao de registo do agente WorldAgent. Se um agente for encontrado a variável **world** (também presente nessa classe) deve tomar o valor encontrado na pesquisa, sendo retornado o valor **true**. Em todos os outros casos o método deverá retornar **false**.

5. Remover um agente do Directory Facilitator (DF) do JADE

   a. No package Common e na classe DFInteraction preencha o método static, “**DeregisterFromDF**” que permite limpar o registo de um agente do DF.

   b. No package “WorldAgent”, na classe com o mesmo nome, implemente o método **takedown**() de forma a que o agente remova o seu registe do DF sempre que abandonar a plataforma.

6. Adicionar um Behaviour ao agente

   a. No package “WorldAgent”, na classe com o mesmo nome, adicione ao método **setup**() o código necessário para que o agente WorldAgent execute o behaviour implementado na classe **ActionResponder**, pertencente ao mesmo package.

### *5.3. Aula 3 – Comunicação*

1. Iniciar uma conversa entre dois agentes no JADE

   a. No package Player, na classe ActionInitiator, complete o método **createInitialMessage**. Este método estático deverá devolver uma ACLMessage que deve ser usada para iniciar a comunicação entre agentes.
   O parâmetro actionOntology contém o valor a definir no campo **ontology** das mensagens, o parâmetro c deverá ser convertido

para o tipo TCharacter utilizando o método convertCharacter presente na classe Serialization do package Common e serializado utilizando o método serializeCharacter, da mesma classe. Posteriormente deverá ser adicionado ao campo **content** da mensagem.
Finalmente, o valor de world (afetado na Aula 2, ponto 4b) deve ser adicionado como o **receiver** da mensagem.

2. Receber uma conversa no JADE

    a. Retorne ao package WorldAgent e à classe ActionResponder. Complete o método **handleRequest** de forma a que a variável c receba um Character a partir do conteúdo da mensagem, utilizando um processo inverso ao descrito no ponto 1a (utilize o método deserializeCharacter da classe Serialization no package Common).

    b. Ainda no mesmo método, inicialize um objeto msg de forma a que seja uma resposta válida para a mensagem recebida.

3. Serialização das mensagens utilizando JAXB

    a. No package Common, na classe Serialization, complete os métodos **serializeCharacter** e **deserializeCharacter**, responsáveis por serializar um objecto TCharacter numa String, e vice-versa.

4. Preparar a lógica do agente WorldAgent para uma jogada "Use"

    a. Implemente a função **commitUse** que deve conter a mecânica de jogo na situação da jogada "Use" (tenha em atenção a especificação da Tabela 1, e inspire-se nos métodos **commitX** já implementados). Faça a alteração correspondente na classe Responder.

## *5.4. Aula 4 – Implementação de Behaviours*

1. Implementação de behaviours específicos de cada agente

    a. No package Entities, adicione um *behaviour* à classe Healer de forma a que este perca 10 pontos da sua HP de 10 em 10 segundos. Adapte a lógica do WorldAgent para quando o Player tentar usar um Healer que já não tenha HP (A mensagem deverá ser diferente, alertando o jogador de que não ganhou pontos bónus).
    b. Implemente um behaviour que imprime no ecrã a HP e os pontos da arma de cada entidade de 5 em 5 segundos. Todas as entidades

relevantes deverão ter este behaviour. Assim, escolha a classe mais adequada para a sua implementação.

### *5.5. Funcionalidades Extra*

1. Complete o protocolo FIPA Request

    a. Modifique a classe Responder para que no tratamento de request envie uma mensagem do tipo Agree para o iniciador. Transfira o processamento e subsequente envio do inform para o método **prepareResultNotification**.

    b. Modifique a classe **Initiator** para receber e processar este **Agree**. Esta classe deves apenas imprimir o conteúdo do Agree.

2. Confira Autonomia ao agente Player

    a. Na versão disponibilizada pelo corpo docente, o Player reage aos comandos inseridos pelo utilizador através de uma interface gráfica. Implemente os comportamentos necessários para que o agente Player jogue de forma **completamente autónoma** e **inteligente** (evitar *Traps*, manter-se dentro dos limites do mapa, atacar *Goblins* e encontrar o *Treasure*).

# 6. Avaliação

A avaliação do trabalho tem a seguinte ponderação:

- Correta implementação e demonstração de funcionamento do trabalho previsto para as aulas 2, 3 e 4:
  - 16 valores
- Correta implementação e demonstração de funcionalidade extra definida no ponto 1:
  - 1 valor
- Correta implementação e demonstração da funcionalidade extra definida no ponto 2:
  - 3 valores

**Docentes**

Ricardo Silva Peres <ricardo.peres@uninova.pt>

Pedro Lima Monteiro <pedro.monteiro@uninova.pt>

José Barata <jab@uninova.pt>