Universidade do Porto
FEUP Faculdade de Engenharia

JADEX Tutorial

Agentes e Inteligência Artificial Distribuída

MIEIC 4º Ano, 1º Semestre

Última atualização: 03/10/2012
jose.pedro.silva@fe.up.pt

Índice

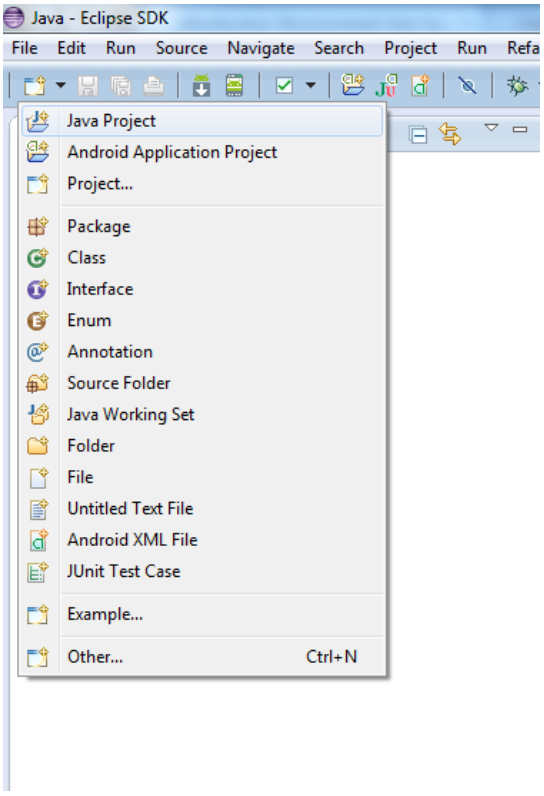
Plataforma Jadex

A versão utilizada neste tutorial é 2.2. Todos os nomes usados que façam referência à versão poderão necessitar de serem alterados caso complete este tutorial com outra versão. Existe um *bug* com *paths* que contenham espaços nesta versão, problema que segundo os autores da plataforma será corrigido na versão 2.2.1.

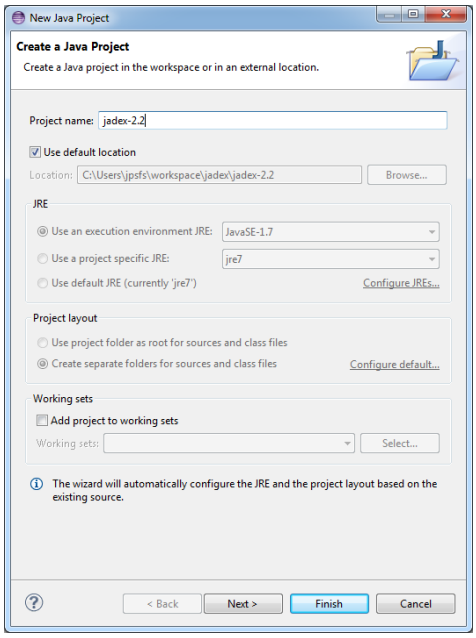
Como lançar a plataforma

Para lançar a plataforma Jadex, abrindo o seu componente Jadex Control Center, basta abrir o Eclipse num *workspace* à escolha, e colocar dentro desse mesmo *workspace* a plataforma Jadex, que pode ser descarregada em: http://sourceforge.net/projects/jadex/files/latest/download

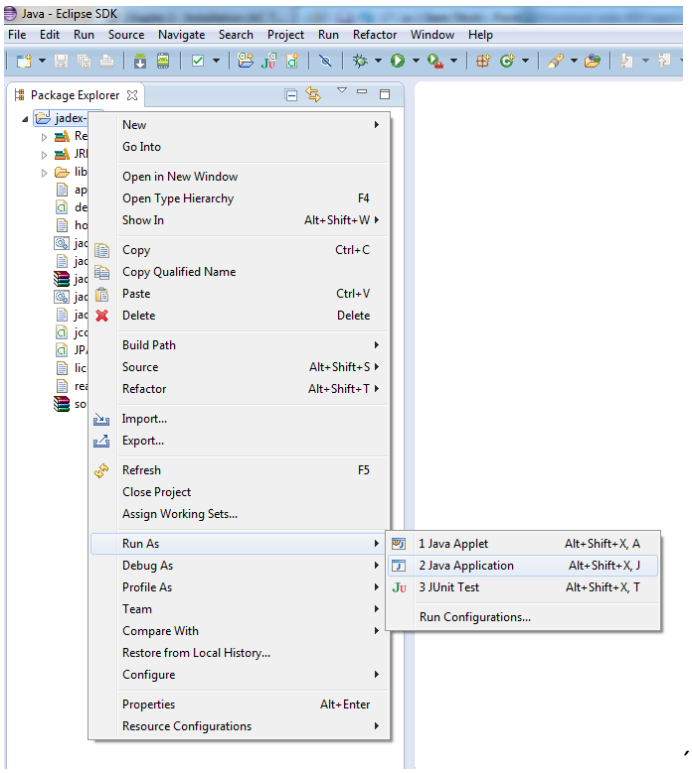
No Eclipse, vamos criar uma Java Project:



Neste caso vamos dar o mesmo nome ao projeto que a pasta que já se encontra dentro do *workspace*.

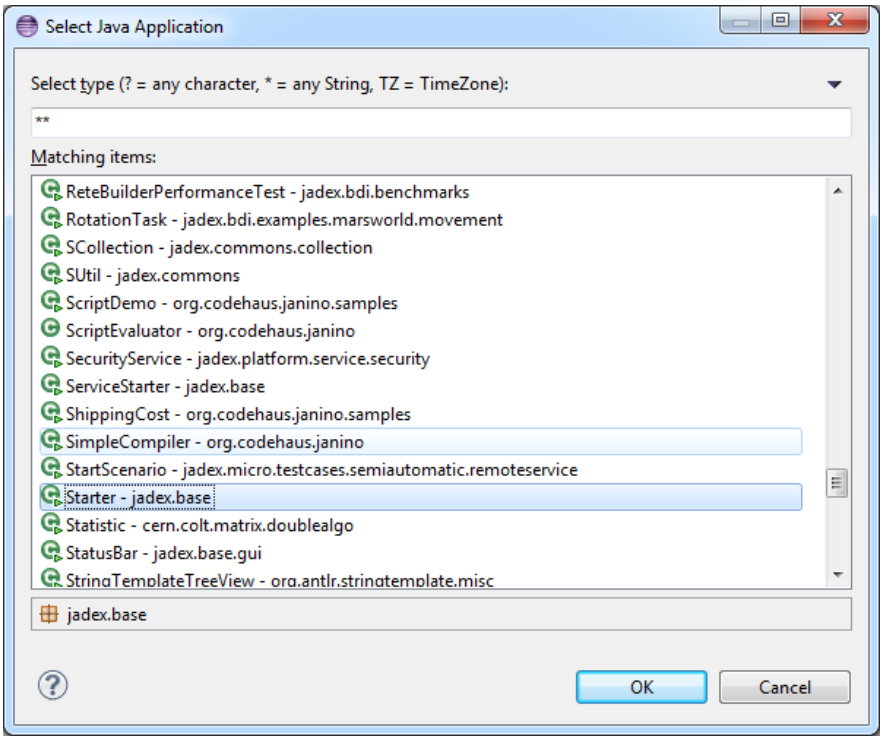


Clicando em *Finish* irá aparecer o projeto no *Package Explorer* do Eclipse.



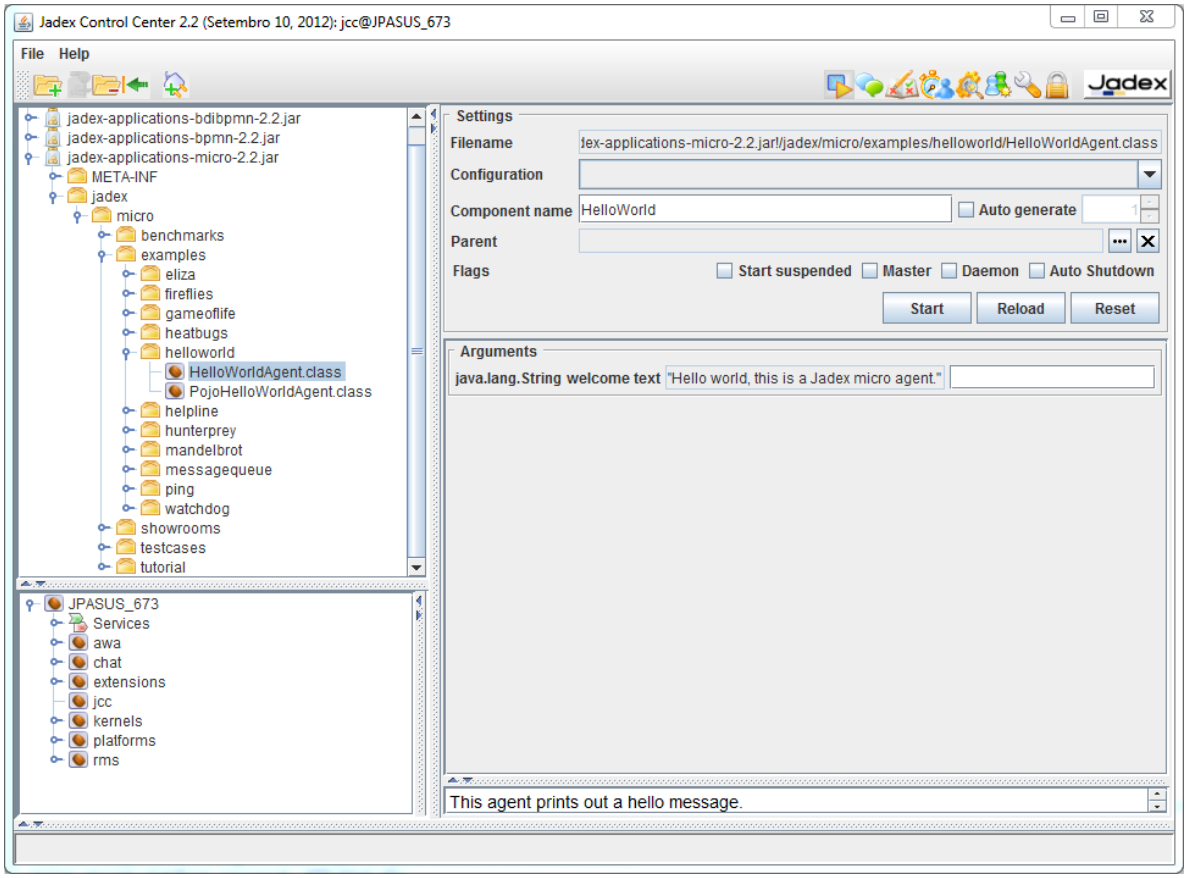
Clique direito no projeto recém-criado, *Run As*, *Java Application*.

Após aguardar alguns segundos, serão apresentadas as várias Java Applications disponíveis neste projeto. Para iniciarmos o *Jadex Control Center* deve escolher a aplicação *Starter* do *jadex.base package.*



Deverá surgir o *Jadex Control Center* (JCC). Aqui já deverão surgir diversos jars que estão presentes na pasta *lib* do projeto criado. Caso o mesmo não aconteça, clique no botão "*Add Path" (*primeiro da *toolbar* a contar da esquerda) e adicione o pacote *jadex-applications-micro-2.2.jar*.

Para teste, podemos correr o exemplo *HelloWorld.* Basta para isso escolher o ficheiro *HelloWorldAgent.class*, como ilustrado, e carregar em *Start*.



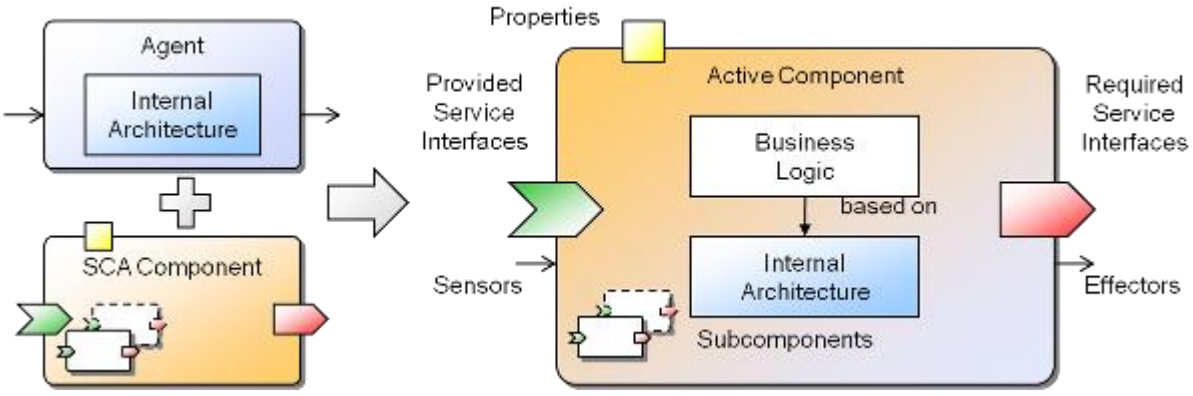
A consola do Eclipse deverá mostrar o seguinte texto:

```
Hello world, this is a Jadex micro agent.
Good bye world.
```

Pode correr outros restantes exemplos. Destes destacam-se os exemplos *heatbugs* e *hunterprey*. Estes exemplos por envolverem outros componentes, devem ser executados a partir dos respetivos **.application.xml*.

Active Components

A abordagem através do conceito de *Active Component* (AC) integra os conceitos de agentes, serviços e componentes de forma a construir uma visão do mundo útil para a modelação e programação de vários tipos de sistemas distribuídos. SCA[1] combina, de forma natural, a arquitetura orientada a objetos (SOA) com uma orientação ao componente, introduzindo comunicação através de serviços. AC parte da base de SCA, e progride na direção de agentes de *software*. A ideia é a de transformar componentes SCA passivos em provedores de serviços autónomos de forma a melhor refletir a realidade, como se pode perceber através da figura seguinte:



Criação de Componentes

XML

Crie um ficheiro XML, com o nome *ChatB1.component.xml*. Neste ficheiro são definidas as propriedades de arranque do agente. A primeira propriedade é o nome que deve ser o mesmo do nome do ficheiro (à semelhança dos *class files* em Java). Adicionalmente pode ser adicionado o atributo *package*, neste caso "tutorial".

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Chat component. -->
<componenttype xmlns="http://jadex.sourceforge.net/jadex"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jadex.sourceforge.net/jadex
                      http://jadex.sourceforge.net/jadex-component-2.2.xsd"
  name="ChatB1" package="tutorial">

</componenttype>
```

Este ficheiro pode ser testado abrindo o JCC e adicionando ao *path* a pasta que contém o *package*

[1] Service Component Architecture: http://en.wikipedia.org/wiki/Service_Component_Architecture

"tutorial". Se não contiver erros, ao ser arrancado o componente deve aparecer no módulo *Component Tree* (parte inferior esquerda). Para terminar a sua execução pode fazer clique direito sobre o componente em execução e selecionar *Kill component.*

Java

Há vários tipos de componentes em Jadex, entre eles *applications, BPMN workflows, micro* e *BDI agents.* A maioria destes componentes é baseado em XML, contudo os *micro agents* utilizam apenas Java.

Exemplo de um *micro agent*:

```
package tutorial;

import jadex.micro.MicroAgent;

/**
 *  Chat micro agent.
 */

public class ChatB2Agent extends MicroAgent
{
}
```

BDI

Para criar um agente BDI é necessário criar um ficheiro XML, contendo os parâmetros de arranque e definições do agente e um ficheiro em Java para cada plano especificado no ficheiro XML.

Agentes

Beliefs

Conhecimento que o agente possui sobre si mesmo e sobre o ambiente. Podem ser representados por qualquer tipo de objeto Java. Podem ser acedidos ou alterados pelos planos utilizando a interface *beliefbase*, herdados de uma *capability* através da utilização do sufixo *ref*, ou referenciados em expressões.

```
<agent>
	...
	<beliefs>
		<!-- Belief declarado -->
		<belief name="juice" class="boolean">
			<fact>false</fact>
		</belief>
		<!-- Herdado de capability -->
		<beliefref name="myself">
			<concrete ref="movement.myself" />
		</beliefref>
		...
	</beliefs>
	...
</agent>
```

Goals

Representam os objetivos, motivações, do agente e que irão levar ao despoletar de ações. Podem ser de quatro tipos:

- *Perform goal:* algo que precisa ser feito, mas não necessariamente um objetivo;
- *Archieve goal:* representa um estado a atingir, não especificando o caminho para lá chegar;
- *Query goal:* semelhante ao *Archieve Goal*, mas que pretende obter informação;
- *Maintain goal:* pretende manter uma, ou mais propriedades sempre satisfeitas.

```
<agent>
	...
	<goals>
		<!-- Belief declarado -->
		<performgoal name="goal_name" retry="true">
		    <!-- Todos os tipos de goal podem ter drop conditions.-->
		    <dropcondition></dropcondition>
		</performgoal>
		<archievegoal name="goal_name" retry="false">
			<parameter name="parameter_name" class="parameter_class">
			    <value>value_of_parameter</value>
			</parameter>
			<!-- Condição para o goal ficar ativo -->
			<createcondition></createcondition>
		</archievegoal>
		<querygoal name="goal_name" retry="true">
		    ...
		</querygoal>
		<maintaingoal name="goal_name">
		    ...
		</maintaingoal>
		...
	</goals>
	...
</agent>
```

Plans

Descrevem como as ações do agente se processam. São selecionados de acordo com a ocorrência de eventos e *goals*. A seleção de *plans* é feita de forma automática pelo sistema. No *Jadex* os *plans* dividem-se em duas partes: um *head* declarado em XML e um *body* declarado em Java.

Head

```
<agent>
	...
	<plan name="plan_name">
		...
		<body class="JavaClass">
		    <trigger>
		        <goal ref="goal_name"/>
		    </trigger>
		</body>
	</plan>
	...
</agent>
```

Body

```
public class MyPlan extends Plan {

    public void body() {
        // Application code goes here.
        ...
    }

    public void passed() {
        // Clean-up code for plan success.
        ...
    }

    public void failed() {
        // Clean-up code for plan failure.
        ...
        getException().printStackTrace();
    }

    public void aborted() {
        // Clean-up code for an aborted plan.
        ...
        System.out.println("Goal achieved? "+ isAbortedOnSuccess());
    }
}
```

Capabilities

No Jadex uma capability representa um módulo de agente composto por *beliefs, goals* e *plans*. Esta funcionalidade permite a reutilização de código, podendo qualquer agente herdar uma ou mais *capabilities*. Para aceder aos *beliefs* ou *goals* de uma *capability* o agente tem que incluir uma referência para a mesma no seu ficheiro XML.

Exemplo de uma *capability*:

```
<capability xmlns="http://jadex.sourceforge.net/jadex-bdi"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://jadex.sourceforge.net/jadex-bdi
                                http://jadex.sourceforge.net/jadex-bdi-2.2.xsd"
  name="CapabilityName"
  package="package.name">
  ...
  <beliefs>...</beliefs>
  <plans>...</plans>
  <goals>...</goals>
</capability>
```

No ficheiro do agente:

```
<agent>
	<capabilities>
		<capability
		    name="CapabilityName"
		    file="package.name/CapabilityName.capability.xml"
		/>
		...
	</capabilities>
	...
</agent>
```

Eventos

Os agentes têm a capacidade de reagir a eventos internos ou externos (mensagens). Os eventos internos são utilizados para sinalizar uma ocorrência dentro de um agente, enquanto as mensagens representam a comunicação entre dois, ou mais, agentes. Os eventos podem despoletar *plans*.

Expressões

As expressões permitem especificar valores dos parâmetros e *beliefs* numa sintaxe semelhante a *SQL*. Esta funcionalidade facilita a seleção de objetos.

Aplicações

Quando se pretende construir um sistema que contenha mais do que um agente simples, é aconselhável que se invista um pouco mais de tempo para desenhar de forma correta a aplicação.

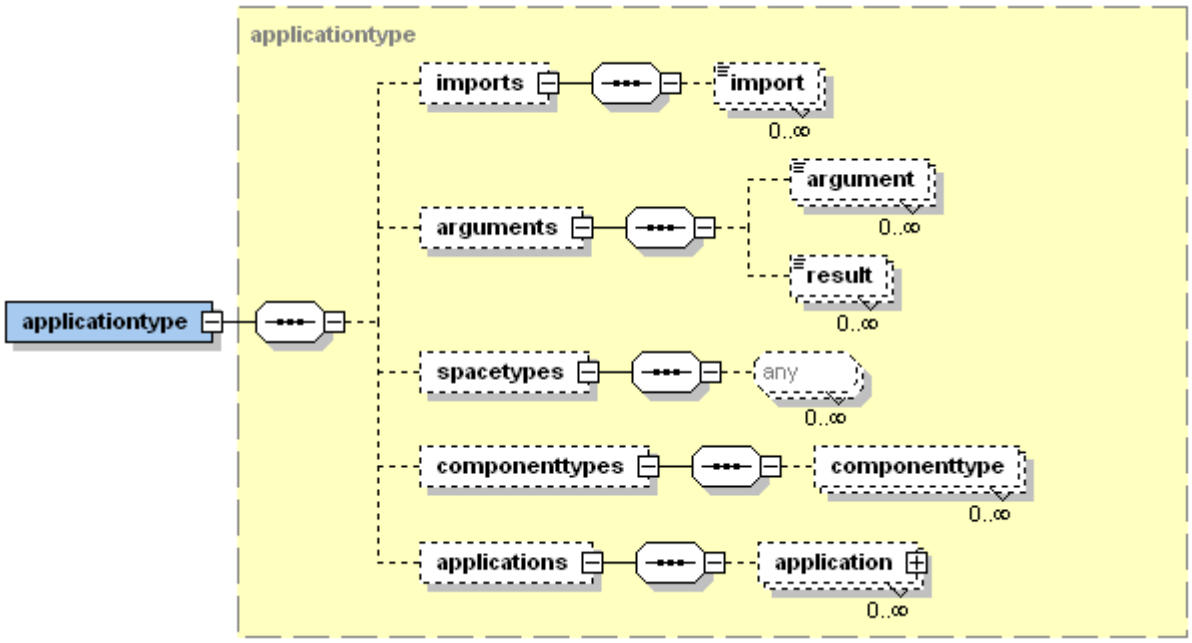


Ilustração 1 - Possível modelação de uma aplicação

Neste tutorial vamos proceder à construção de uma pequena aplicação, com base no exemplo de Ping/Pong visto nas aulas teóricas para a plataforma JADE. Esta aplicação faz apenas uso de micro agentes, promovendo a sua comunicação. O código fonte deste exemplo pode ser encontrado em: http://jpsfs.com/data/PingPong.zip

1. Criar um projeto Java, com o nome PingPong, através do Eclipse;
2. Criar uma pasta *lib* dentro do projeto, contendo a plataforma Jadex;
3. Adicionar a plataforma Jadex ao *Java Build Path;*

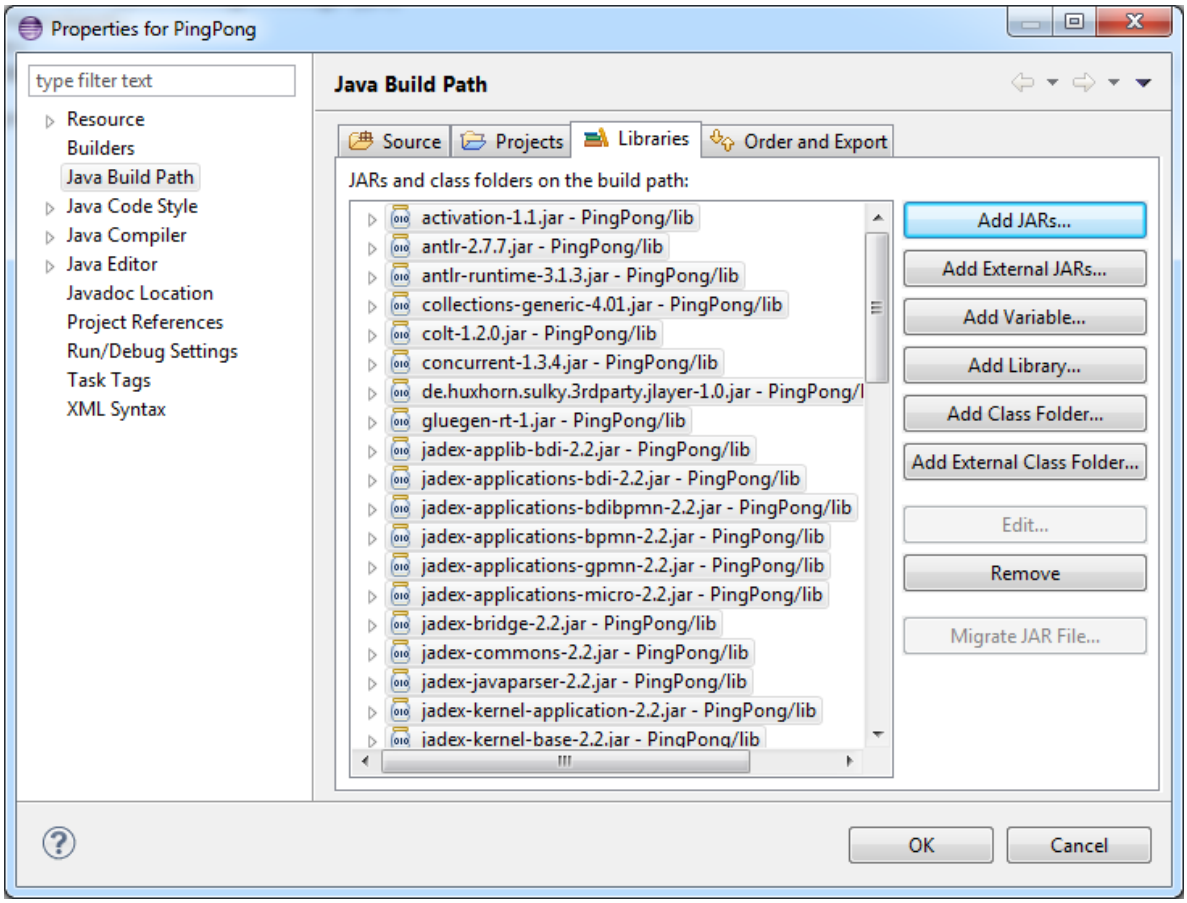


Ilustração 2 - [3] Adicionar o Jadex ao Java Build Path

4. Crie a classe *PongAgent* que herda da classe *MicroAgent*, no *package ping*. Este agente será o responsável por responder "Pong" a mensagens que contenham "Ping" como conteúdo. O código desta classe é o que se reproduz a seguir:

```
package ping;

import jadex.bridge.service.types.message.MessageType;
import jadex.micro.MicroAgent;
import java.util.Map;

public class PongAgent extends MicroAgent {
	public void messageArrived(Map msg, MessageType mt){
		//Get the message performative
		String perf = (String)msg.get("performative");

		switch (perf) {
		case "query-if":
		case "query-ref":
			//Check if the message contains a 'ping'
			if(msg.get("content").equals("ping")){
				//Great! Message contains ping. Reply with Pong.
				Map reply = createReply(msg, mt);
				reply.put("content", "pong");
				reply.put("performative", "inform");
				reply.put("sender", getComponentIdentifier());
				sendMessage(reply, mt);
				System.out.println("[" + this + "] Just sended a
Pong!");
			}
			break;
		default:
			getLogger().severe("Unknown Performative");
			break;
		}
	}
}
```

5. Crie agora o agente *Ping*. Este agente será o responsável por começar a comunicação. Para tal crie a classe *PingAgent* que herda de *MicroAgent*, no *package ping*. O código desta classe é o que se reproduz de seguida:

```
package ping;

import jadex.bridge.ComponentIdentifier;
import jadex.bridge.IComponentIdentifier;
import jadex.bridge.IComponentStep;
import jadex.bridge.IInternalAccess;
import jadex.bridge.fipa.SFipa;
import jadex.bridge.service.types.message.MessageType;
import jadex.commons.SUtil;
import jadex.commons.future.Future;
import jadex.commons.future.IFuture;
import jadex.micro.MicroAgent;
import jadex.micro.annotation.Arguments;
import jadex.micro.annotation.Description;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import java.util.logging.Logger;

@Description("A simple agent that sends pings to another agent and waits for
replies.")
@Arguments({@jadex.micro.annotation.Argument(name="receiver",
clazz=IComponentIdentifier.class, description="The component receiver of the
ping target."), @jadex.micro.annotation.Argument(name="missed_max",
clazz=int.class, description="Maximum number of allowed missed replies",
defaultvalue="3"), @jadex.micro.annotation.Argument(name="timeout",
clazz=long.class, description="Timeout for reply", defaultvalue="1000"),
@jadex.micro.annotation.Argument(name="content", clazz=String.class,
description="Ping message content", defaultvalue="\"ping\"")})
public class PingAgent extends MicroAgent
{
	protected IComponentIdentifier receiver;
	  protected int dif;
	  protected Set sent;

	  public IFuture<Void> executeBody()
	  {
	    final Future ret = new Future();

	    //Get the parameters
	    this.receiver = ((IComponentIdentifier)getArgument("receiver"));
	    final int missed_max =
((Number)getArgument("missed_max")).intValue();
	    final long timeout = ((Number)getArgument("timeout")).longValue();
	    final Object content = getArgument("content");
	    this.sent = new HashSet();

	    //Ping agent will start
	    IComponentStep step = new IComponentStep()
	    {
	      public IFuture<Void> execute(IInternalAccess ia)
	      {
	       //After send "missed_max" if there is no answer, than terminate.
	        if (PingAgent.this.dif > missed_max)
	        {
	          PingAgent.this.getLogger().warning("Ping target does not
respond: " + PingAgent.this.receiver);
```

```
                ret.setResult(null);
              }
              else
              {
                String convid =
SUtil.createUniqueId(PingAgent.this.getAgentName());
                Map msg = new HashMap();
                msg.put("content", content);
                msg.put("performative", "query-if");
                msg.put("conversation_id", convid);
                msg.put("receivers", new IComponentIdentifier[] {
PingAgent.this.receiver });

                PingAgent.this.dif += 1;
                PingAgent.this.sent.add(convid);
                PingAgent.this.sendMessage(msg, SFipa.FIPA_MESSAGE_TYPE);
                PingAgent.this.waitFor(timeout, this);
              }
              return IFuture.DONE;
            }
          };
          if (this.receiver == null)
          {
            this.receiver = new ComponentIdentifier("Pong",
getComponentIdentifier().getParent());
          }

          scheduleStep(step);

          return ret;
        }

        public void messageArrived(Map msg, MessageType mt)
        {
            //If a message arrives, clear dif and missed messages.
          if (mt.equals(SFipa.FIPA_MESSAGE_TYPE))
          {
            String convid = (String)msg.get("conversation_id");
            if (this.sent.remove(convid))
            {
              this.dif = 0;
              this.sent.clear();
               System.out.println("[" + this + "] Just received a Pong!");
            }
          }
        }
}
```

O método *executeBody()* é executado a quando da criação do agente. É por isso então que neste método colocamos o envio da mensagem "Ping" para o agente *Pong*. Esta mensagem será enviada várias vezes, caso não se obtenha resposta o agente *Ping* cessará comunicações.

6. Falta neste momento criar a aplicação, isto é, agregar os dois agentes e lançar a sua execução. Para tal vamos proceder à criação do ficheiro *PingPongScenario.application.xml*. O ficheiro de definição da aplicação deve ter sempre um nome do tipo **.application.xml* de forma a que ferramentas como o JCC possam identifica-los facilmente.

Transcreve-se abaixo o conteúdo do ficheiro *PingPongScenario.application.xml:*

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
	<H3> The PingPong Scenario </H3>

	Starts Ping agent, and sends a Ping message to Pong Agent. Pong agent
will reply with pong to Ping Agent.
-->
<applicationtype xmlns="http://jadex.sourceforge.net/jadex"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://jadex.sourceforge.net/jadex
                                    http://jadex.sourceforge.net/jadex-
application-2.2.xsd"
	name="PingPongScenario" package="ping">

	<componenttypes>
		<componenttype name="Pong" filename="bin/ping/PongAgent.class"
master="true"/>
		<componenttype name="Ping" filename="bin/ping/PingAgent.class"
master="true"/>
	</componenttypes>

	<configurations>
		<configuration name="Ping-Pong">
			<components>
				<component type="Pong" name="Pong" />
				<component type="Ping" name="Ping" />
			</components>
		</configuration>

	</configurations>

</applicationtype>
```

No primeiro bloco de comentário do ficheiro temos colocado uma descrição da aplicação. Podem ser utilizadas tags HTLM para melhor dispor a informação. No nó *applicationtype* é declarado o modelo do XML (apesar de não ser obrigatório é recomendável que o modelo seja indicado para que ferramentas como o JCC possam avaliar o ficheiro), é ainda indicado o nome da aplicação e o pacote. No nó *componenttypes* são indicados quais os agentes disponíveis. Neste caso como ambos são micro agentes, indica-se o *path* para os *class files* respectivos, noutros caso será indicado o ficheiro XML que define o agente a incluir.

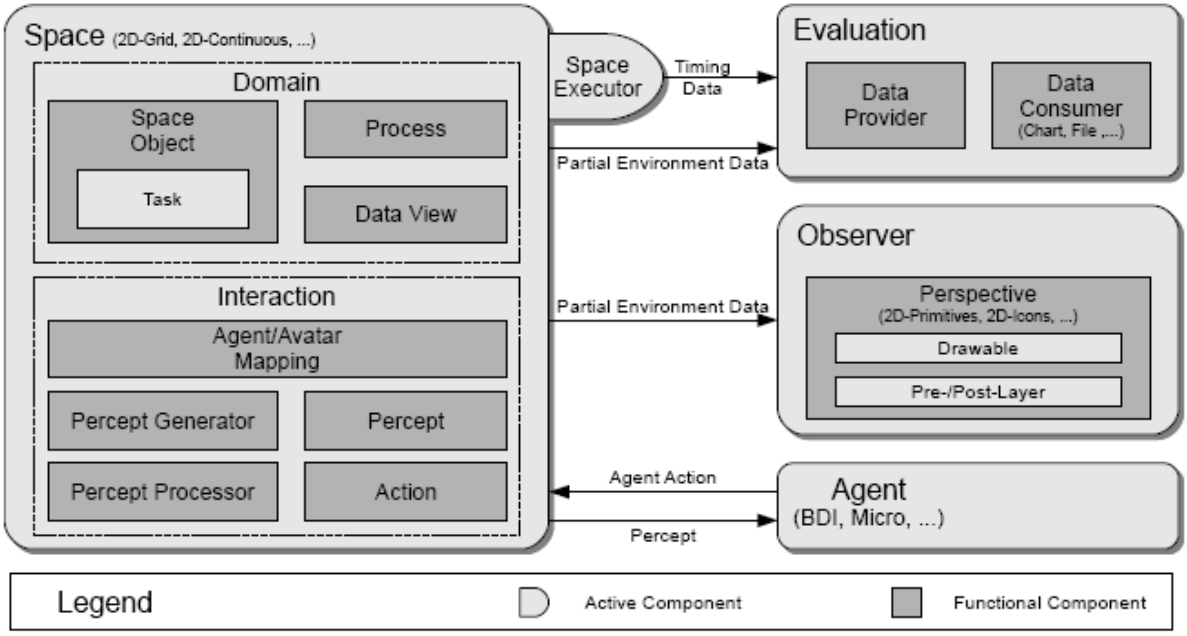
No nó *configurations* inclui-se as configurações disponíveis para a nossa aplicação. Neste caso basta uma configuração, indicando que devem ser executados ambos os agentes.

7. Abra o JCC através do Eclipse como explicado no primeiro módulo deste tutorial e inclua o projeto no *path.* Execute de seguida a aplicação *PingPongScenario.application.xml.* Poderá ver na consola do Eclipse as mensagens a serem trocadas entre os dois agentes.
   Para um maior detalhe, poderá aceder ao módulo de *debug* do JCC (quarto ícone a contar da direita na *toolbar*).

Espaços

Aplicações que fazem uso de Agentes consistem não só em agentes mas também no ambiente em que estão situados.

Conceitos



Na figura acima é possível observar a divisão conceptual do suporte a espaços do Jadex, bem como a suas interligações. Este suporte a espaços consiste nos módulos de *espaço* (*space), agentes (agents), observadores (observers)* e *avaliação (evaluators)*.

Espaço

No modelo do domínio são declarados os elementos constituintes do espaço. No ambiente do Jadex estes elementos são chamados de *space objects* que podem representar qualquer artefacto do mundo que se pretende descrever.

Observadores

Os *observadores* representam interfaces para olhar um ambiente. Tipicamente permite observar o estado atual do ambiente bem como dos seus objetos.

Avaliador

Para além do espaço em si mesmo e da sua visualização, um avaliador por, opcionalmente, ser configurado. Esta avaliação permite a recolha de dados específicos do espaço para que possa ser posteriormente analisada.

Exemplo

Para este exemplo irá ser criado um pequeno mapa onde serão colocados objetos que serão considerados lixo. A tarefa de o agente será recuperar esse lixo.

Este exemplo ter por base o exemplo *GarbageCollector* fornecido pela equipa do Jadex, onde foram feitas algumas simplificações para um melhor entendimento do mesmo. O código fonte deste exemplo pode ser encontrado em: http://jpsfs.com/data/GarbageCollector.zip

Criar um projeto Java, com o nome *GarbageCollector*, através do Eclipse;

1. Criar uma pasta *lib* dentro do projeto, contendo a plataforma Jadex;
2. Adicionar a plataforma Jadex ao *Java Build Path;*
3. Vamos começar por construir o manifesto da nossa aplicação. Para tal criamos um ficheiro com o nome *GarbageCollector.application.xml* na pasta *src*.
   Este ficheiro, como já foi explicado anteriormente, define a nossa aplicação. Depois de dados os passos iniciais para a construção da nossa aplicação, vamos declarar o espaço dentro da nossa aplicação, dentro do nó *extensiontypes,* como se demonstra a seguir:

   ```
   <!-- Declaration of a grid environment as space type. -->
   <env:envspacetype name="gc2dspace" width="10" height="10" class="Grid2D">

   </env:envspacetype>
   ```

   Neste caso declaramos um objeto da classe *Grid2D*, quadrado, de tamanho 10.
4. Agora que está criado o espaço, vamos declarar os objetos presentes nesse espaço. Para este exemplo temos objetos do tipo lixo (*garbage)*, um robô que apanha o lixo (*collector)* e um robô para de desfazer do lixo (*burner*).

   ```
   <env:objecttypes>
   	<env:objecttype name="collector">
   		<env:property name="rotation" dynamic="true">
   			$object.lastmove==null || $object.lastmove.equals("right")?
   null:
   			$object.lastmove.equals("left")? New
   Vector3Double(Math.PI,0,Math.PI):
   			$object.lastmove.equals("up")? new
   Vector3Double(0,0,Math.PI*3/2):
   			new Vector3Double(0,0,Math.PI/2)
   		</env:property>
   		<env:property name="vision_range">0</env:property>
   		<env:property name="position" event="true"/>
   	</env:objecttype>
   	<env:objecttype name="burner">
   		<env:property name="vision_range">0</env:property>
   	</env:objecttype>
   	<env:objecttype name="garbage">
   		<env:property name="position" event="true"/>
   	</env:objecttype>
   </env:objecttypes>
   ```

   Neste caso, para tornar o comportamento do robô *collector* mais real, quando este muda o sentido da sua marcha é executada uma rotação do seu objeto, de forma a que ele esteja sempre virado para o local onde se dirige.

5. Posto isto, falta ainda adicionar o “lixo” ao nosso mapa. Para tal criamos um processo que correrá de forma independente e que irá criar lixo ao longo do tempo em posições aleatórias.

```
<env:processtypes>
	<env:processtype name="create" class="DefaultObjectCreationProcess">
		<env:property name="tickrate">5</env:property>
		<env:property name="type">"garbage"</env:property>
		<env:property name="properties" dynamic="true">
			jadex.commons.SUtil.createHashMap(
				new String[]{Space2D.PROPERTY_POSITION},
				new
Object[]{((Space2D)$space).getRandomPosition(Vector2Int.ZERO)})
		</env:property>
	</env:processtype>
</env:processtypes>
```

6. Para visualizarmos estes dados, é necessário criar uma *dataview*.

```
<env:dataviews>
	<env:dataview name="view_all" class="GeneralDataView2D" />
</env:dataviews>
```

7. A próxima etapa será definir as perceções disponíveis no nosso mundo, nomeadamente os geradores dessas perceções e respetivos processadores dessas mesmas perceções.

```
<env:percepttypes>

	<!-- Percepttypes that the environment produces and agents can consume. -->
	<env:percepttype name="garbage_appeared" objecttype="garbage"/>
	<env:percepttype name="garbage_disappeared" objecttype="garbage" />

	<!-- Percept generators define which percepts should be created -->
	<env:perceptgenerator name="visiongen" class="DefaultVisionGenerator">
		<env:property name="maxrange">0</env:property>
		<env:property name="range_property">"vision_range"</env:property>
	<!-- percept types are declared with associated action types:
					percepttype, actiontype1, actiontype2, ... --
>
		<env:property name="percepttypes">
			new Object[]
			{
				new String[]{"garbage_appeared", "appeared", "created"},
				new String[]{"garbage_disappeared", "disappeared",
"destroyed"}
			}
		</env:property>
	</env:perceptgenerator>

	<!-- Percept processors consume percepts and push them into the agents. -->
	<env:perceptprocessor componenttype="Collector"
class="DefaultBDIVisionProcessor" >
		<env:property name="percepttypes">
			new Object[]
			{
				new String[]{"garbage_appeared", "add", "garbages"},
				new String[]{"garbage_disappeared", "remove", "garbages"}
			}
		</env:property>
	</env:perceptprocessor>

	<env:perceptprocessor componenttype="Burner"
class="DefaultBDIVisionProcessor" >
		<env:property name="percepttypes">
			new Object[]
			{
				new String[]{"garbage_appeared", "add", "garbages"},
				new String[]{"garbage_disappeared", "remove", "garbages"}
			}
		</env:property>
	</env:perceptprocessor>
</env:percepttypes>
```

8. De seguida definimos as ações que podem ser executadas no ambiente.
   Essas ações são o levantar, o pousar e o queimar o lixo, e depois a ação de o robô se deslocar até determinada posição.

```
<!-- Actions that can be performed in the environment. -->
<env:actiontypes>
	<env:actiontype name="pickup" class="PickupWasteAction">
		<env:property name="actor_id" class="IComponentIdentifier"/>
	</env:actiontype>
	<env:actiontype name="drop" class="DropWasteAction" >
		<env:property name="actor_id" class="IComponentIdentifier"/>
	</env:actiontype>
	<env:actiontype name="burn" class="BurnWasteAction" >
		<env:property name="actor_id" class="IComponentIdentifier"/>
	</env:actiontype>
	<env:actiontype name="go" class="GoAction" >
		<env:property name="object_id" class="Object"/>
		<env:property name="direction" class="String"/>
	</env:actiontype>
</env:actiontypes>
```

9. Passamos agora à definição da perspetiva, indicando os *avatars* para cada objeto. Definimos então os icons para o *collector,* para o *burner*, para o *garbage*, e o fundo, para se ter uma melhor perceção do espaço onde os agentes se movimentam.

```
<!-- Perspectives define the user-interface appearance. -->
<env:perspectives>
	<env:perspective name="icons" class="Perspective2D" objectplacement="center">
		<env:drawable objecttype="collector" width="1.0" height="1.0"
rotation="rotation">
			<env:property name="vision_size" dynamic="true">new
Vector2Double($object.vision_range*2+1)</env:property>
			<env:texturedrectangle layer="2" x="0" y="-0.1" width="0.8"
height="0.8" imagepath="garbagecollector/images/collector.png" />
			<env:texturedrectangle layer="3" width="0.5" height="0.5"
x="0.3" y="-0.3" imagepath="garbagecollector/images/garbage.png">

	<env:drawcondition>$object.garbage!=null</env:drawcondition>
			</env:texturedrectangle>
			<env:rectangle layer="-1" size="vision_size" color="#ffff007f"
/>
		</env:drawable>
		<env:drawable objecttype="burner" width="1" height="1">
			<env:property name="vision_size" dynamic="true">new
Vector2Double($object.vision_range*2+1)</env:property>
			<env:texturedrectangle width="0.8" height="0.8" layer="1"
imagepath="garbagecollector/images/burner.png" />
				<env:texturedrectangle width="0.6" height="0.6" layer="2"
imagepath="garbagecollector/images/flames.png" >
					<env:drawcondition>$object.garbage!=null</env:drawc
					ondition>
			</env:texturedrectangle>
			<env:rectangle layer="-1" size="vision_size" color="#ffff007f"
/>
		</env:drawable>
		<env:drawable objecttype="garbage" width="0.7" height="0.7">
			<env:texturedrectangle layer="0"
imagepath="garbagecollector/images/garbage.png" />
		</env:drawable>
		<!-- Draw tiles to have better perception of the world -->
		<env:prelayers>
			<env:tiledlayer width="3.5" height="3.5"
imagepath="garbagecollector/images/bg.jpg" />
			<env:gridlayer width="1.0" height="1.0" color="white" />
		</env:prelayers>
	</env:perspective>
</env:perspectives>
```

10. Para terminar o nosso ambiente, falta declarar a forma como o espaço é atualizado:

```
<!-- A space executor defines the space execution policy (e.g. roundbased vs.
continuous). -->
<env:spaceexecutor class="RoundBasedExecutor">
	<env:property name="space">$space</env:property>
</env:spaceexecutor>
```

11. Para concluir o nosso manifesto basta declarar os agentes e definir a configuração da aplicação.

```
<componenttypes>
	<componenttype name="Collector"
filename="garbagecollector/GarbageCollector.agent.xml"/>
	<componenttype name="Burner"
filename="garbagecollector/GarbageBurner.agent.xml"/>
</componenttypes>

<!-- Application instance definitions. -->
<configurations>
	<configuration name="One Burner / One Collector">
		<extensions>
			<env:envspace name="mygc2dspace" type="gc2dspace">
				<env:processes>
					<env:process type="create">
						<env:property name="tickrate">
                                                15
                                                </env:property>
					</env:process>
				</env:processes>

				<env:observers>
					<env:observer name="world" dataview="view_all"
perspective="abstract"/>
				</env:observers>
			</env:envspace>
		</extensions>

		<components>
			<component type="Collector"/>
			<component type="Burner"/>
		</components>
	</configuration>

</configurations>
```

12. O passo seguinte é definirmos os nossos agentes.
    a. Agente *Collector:*
        i. O agente necessita de "acreditar" no espaço onde está inserido, ter consciência de si próprio, da posição onde se encontra, de saber o que significa "lixo", de identificar uma localização com "lixo" e saber se ele próprio está ou não já a transportar lixo. Para tal identificamos os seus *beliefs*:

```
<beliefs>
	<!-- Environment. -->
	<belief name="env" class="Grid2D">
		<fact class="IFuture">
			$scope.getParentAccess().getExtension("mygc2dspace")
		</fact>
	</belief>

	<!-- Myself. -->
	<belief name="myself" class="ISpaceObject">
		<fact>
			$beliefbase.env.getAvatar($scope.getComponentDescription(),
		$scope.getAgentModel().getFullName())
		</fact>
	</belief>

	<!-- The actual position on the grid world. -->
	<belief name="pos" class="IVector2" evaluationmode="push">
		<fact language="jcl">
			$beliefbase.myself.position
		</fact>
	</belief>

	<!-- The garbages at the current position. -->
	<beliefset name="garbages" class="ISpaceObject"/>

	<!-- When at least one piece of waste is present on the
		current position the actual position is perceived as dirty. -->
	<belief name="is_dirty" class="boolean" evaluationmode="push">
		<fact language="jcl">
			$beliefbase.garbages.length>0
		</fact>
	</belief>

	<!-- Flag indicating if the agent currently carries garbage. -->
	<belief name="has_garbage" class="boolean" evaluationmode="push">
		<fact language="jcl">
			$beliefbase.myself.garbage!=null
		</fact>
	</belief>
</beliefs>
```

ii. De seguida identificamos os seus objetivos. Aqui há que distinguir dois tipos de objetivos: os *Archieve Goal* (movimentar-se e pegar em "lixo") e *Perform Goal* (verificar continuamente se existe "lixo" na sua posição).

```
<goals>

	<!-- Goal for picking up a piece of waste, bringing it
		to some burner and going back. A new goal is created
		whenever the actual position is dirty and there is no
		burner present. -->
	<achievegoal name="take">
		<unique/>
		<creationcondition language="jcl">
			$beliefbase.is_dirty &amp;&amp;
$beliefbase.env.getSpaceObjectsByGridPosition($beliefbase.pos, "burner")==null
		</creationcondition>
		<deliberation>
			<inhibits ref="check"/>
		</deliberation>
	</achievegoal>

	<!-- Goal for running around on the grid and searching for garbage. -->
	<performgoal name="check" exclude="never"/>

	<!-- Goal for going to a specified position. -->
	<achievegoal name="go">
		<parameter name="pos" class="IVector2"/>
	</achievegoal>

	<!-- The goal for picking up waste. Tries endlessly to pick up. -->
	<achievegoal name="pick" exclude="never" retrydelay="100">
		<dropcondition language="jcl">
			!$beliefbase.is_dirty &amp;&amp; !$beliefbase.has_garbage
		</dropcondition>
	</achievegoal>
</goals>
```

iii. Falta ainda indicar os seus planos, neste caso, indicando como deve proceder para atingir os objetivos traçados anteriormente.

```
<plans>
	<!-- Plan for taking some waste, bringing it to a burner
		and going back. -->
	<plan name="take_plan">
		<body class="TakePlanEnv"/>
		<trigger>
			<goal ref="take"/>
		</trigger>
	</plan>

	<!-- Plan for running on a predefined route and searching waste. -->
	<plan name="checking_plan">
		<body class="CheckingPlanEnv"/>
		<trigger>
			<goal ref="check"/>
		</trigger>
	</plan>

	<!-- Plan for picking up some garbage. -->
	<plan name="pickup_plan">
		<body class="PickUpPlanEnv"/>
		<trigger>
			<goal ref="pick"/>
		</trigger>
	</plan>

	<!-- Plan for going to a target position. -->
	<plan name="go_plan">
		<parameter name="pos" class="IVector2">
			<goalmapping ref="go.pos"/>
		</parameter>
		<body class="GoPlanEnv"/>
		<trigger>
			<goal ref="go"/>
		</trigger>
	</plan>
</plans>
```

iv. Está apenas a falta a configuração do agente, indicando qual o objetivo inicial.

```
<configurations>
	<configuration name="default">
		<goals>
			<!-- Initial goal for searching for garbage. -->
			<initialgoal ref="check"/>
		</goals>
	</configuration>
</configurations>
```

b. Agente *Burner*:

i. Na mesma base do Agente *Collector*, indicamos os *beliefs:*

```
<beliefs>
	<!-- Environment. -->
	<belief name="env" class="Grid2D">
		<fact
class="IFuture">$scope.getParentAccess().getExtension("mygc2dspace")</fact>
	</belief>

	<!-- Myself. -->
	<belief name="myself" class="ISpaceObject">

	<fact>$beliefbase.env.getAvatar($scope.getComponentDescription(),
$scope.getAgentModel().getFullName())</fact>
	</belief>

	<!-- The actual position on the grid world. -->
	<belief name="pos" class="IVector2" evaluationmode="push">
		<fact language="jcl">
			$beliefbase.myself.position
		</fact>
	</belief>

	<!-- The garbages at the current position. -->
	<beliefset name="garbages" class="ISpaceObject"/>
</beliefs>
```

ii. Os seus objetivos. Neste caso o nosso agente só terá dois objetivos, o de pegar no lixo presente no seu espaço, e o de o queimar.

```
<goals>
	<!-- The burn waste goal. For every garbage occurring at
		its position a new goal is craeted (see binding).
		The unique tag avoids creating more than one goal
		per specific piece of garbage. -->
	<achievegoal name="burn">
		<parameter name="garbage" class="ISpaceObject">
			<value>$garbage</value>
		</parameter>
		<unique/>
		<creationcondition language="jcl">
			ISpaceObject $garbage &amp;&amp;
Arrays.asList($beliefbase.garbages).contains($garbage)
		</creationcondition>
		<deliberation cardinality="1"/>
	</achievegoal>

	<!-- The goal for picking up waste. Tries endlessly to pick up. -->
	<achievegoal name="pick" exclude="never"/>
</goals>
```

iii. E planos:

```
<plans>
	<!-- Plan for burning garbage. -->
	<plan name="burn_plan">
		<body class="BurnPlanEnv"/>
		<trigger>
			<goal ref="burn"/>
		</trigger>
	</plan>

	<!-- Plan for trying to pick up waste. -->
	<plan name="pickup_plan">
		<body class="PickUpPlanEnv"/>
		<trigger>
			<goal ref="pick"/>
		</trigger>
	</plan>
</plans>
```

13. Como havia sido explicado anteriormente, os planos têm duas partes o *head* definido em XML e o *body* definido em Java. Estando já os planos identificados em cada um dos agentes, vamos então proceder à sua definição em Java.
    O nome do ficheiro de cada plano está definido no atributo *class* do nó *plan/body*.
    a. Plano *take_plan*: Este plano consiste em agarrar no "lixo", levá-lo até ao agente *Burner*, pousar o "lixo" nessa posição e regressar à posição inicial.
        i. Primeiro criamos uma classe Java, com o nome *TakePlanEnv* que herda da classe *Plan (jadex.bdi.runtime.Plan)*;
        ii. De seguida definimos o *body* do plano:

```
public void body()
	{
		Space2D grid = (Space2D)getBeliefbase().getBelief("env").getFact();

		// Pickup the garbarge.
		IGoal pickup = createGoal("pick");
		dispatchSubgoalAndWait(pickup);

		// Go to the burner.
		ISpaceObject myself =
			(ISpaceObject)getBeliefbase().getBelief("myself").getFact();
		IVector2 oldpos =
			(IVector2)myself.getProperty(Space2D.PROPERTY_POSITION);
		IGoal go = createGoal("go");
		ISpaceObject burner = grid.getNearestObject(oldpos, null, "burner");
		IVector2 pos =
			(IVector2)burner.getProperty(Space2D.PROPERTY_POSITION);
		go.getParameter("pos").setValue(pos);
		dispatchSubgoalAndWait(go);

		// Put down the garbarge.
		Map params = new HashMap();
		params.put(ISpaceAction.ACTOR_ID, getComponentDescription());
		SyncResultListener srl	= new SyncResultListener();
		grid.performSpaceAction("drop", params, srl);
		srl.waitForResult();

		// Go back.
		IGoal goback = createGoal("go");
		goback.getParameter("pos").setValue(oldpos);
		dispatchSubgoalAndWait(goback);
	}
```

b. Plano *checking_plan*:

O plano consiste em fazer movimentar o nosso agente, para que ele possa encontrar "lixo" para recolher.

```
public void body()
	{
		Space2D env = (Space2D)getBeliefbase().getBelief("env").getFact();
		IVector2 size = env.getAreaSize();
		IVector2 mypos =
			(IVector2)getBeliefbase().getBelief("pos").getFact();
		IVector2 newpos = computeNextPosition(mypos, size.getXAsInteger(),
size.getYAsInteger());

		//Moving from _mypos_ to _newpos_
		IGoal go = createGoal("go");
		go.getParameter("pos").setValue(newpos);
		dispatchSubgoalAndWait(go);
		//Moved to _newpos_
	}
```

Este plano assenta na mesma estrutura do anterior, contudo neste caso precisamos de uma função auxiliar para calcular a próxima posição do nosso agente *Collector*. Vamos chamar *computeNextPosition()* a essa função.

```
protected static IVector2 computeNextPosition(IVector2 pos, int sizex, int sizey)
{
	// Go right in even lanes
	if(pos.getXAsInteger()+1<sizex && pos.getYAsInteger()%2==0)
	{
		pos = new Vector2Int(pos.getXAsInteger()+1, pos.getYAsInteger());
	}
	// Go left in odd lanes
	else if(pos.getXAsInteger()-1>=0 && pos.getYAsInteger()%2==1)
	{
		pos = new Vector2Int(pos.getXAsInteger()-1, pos.getYAsInteger());
	}
	// Go down else
	else
	{
		pos = new Vector2Int(pos.getXAsInteger(),
				(pos.getYAsInteger()+1)%sizey);
	}
	return pos;
}
```

c. Plano *pick_up*:

Este plano consiste em levantar o "lixo" na posição atual.

```
public void body()
{

	IEnvironmentSpace env =
		(IEnvironmentSpace)getBeliefbase().getBelief("env").getFact();

	Map params = new HashMap();
	params.put(ISpaceAction.ACTOR_ID, getComponentDescription());
	SyncResultListener srl	= new SyncResultListener();
	env.performSpaceAction("pickup", params, srl);
	if(!((Boolean)srl.waitForResult()).booleanValue())
		fail();

}
```

d. Plano *go*:

Este plano recebe como parâmetro a nova posição para onde o agente *Collector* se deve dirigir, e encarrega-se de transmitir uma ação ao agente de forma a que este se movimente.

```
public void body()
{
	Grid2D env = (Grid2D)getBeliefbase().getBelief("env").getFact();

	//Get the position, as specified in the XML
	IVector2 target = (IVector2)getParameter("pos").getValue();
	ISpaceObject myself =
			(ISpaceObject)getBeliefbase().getBelief("myself").getFact();

	while(!target.equals(myself.getProperty(Space2D.PROPERTY_POSITION)))
	{
		String dir = null;
		IVector2 mypos =
				(IVector2)myself.getProperty(Space2D.PROPERTY_POSITION);

		IVector1 md = env.getShortestDirection(mypos.getX(),
							target.getX(), true);

		switch(md.getAsInteger()){
		case 1:
			dir = GoAction.RIGHT;
			break;
		case -1:
			dir = GoAction.LEFT;
			break;
		default:
			md = env.getShortestDirection(mypos.getY(),
							target.getY(), false);
			switch(md.getAsInteger()){
			case 1:
				dir = GoAction.DOWN;
				break;
			case -1:
				dir = GoAction.UP;
				break;
			}
		}


		Map params = new HashMap();
		params.put(GoAction.DIRECTION, dir);
		params.put(ISpaceAction.OBJECT_ID,
			env.getAvatar(getComponentDescription()).getId());
		SyncResultListener srl	= new SyncResultListener();
		env.performSpaceAction("go", params, srl);
		srl.waitForResult();
	}
}
```

e. Plano *burn_plan*:

Este plano diz respeito ao agente *Burner*, e visa queimar o “lixo” presente na posição deste agente.

```
public void body()
{
	IEnvironmentSpace env =
		(IEnvironmentSpace)getBeliefbase().getBelief("env").getFact();

	// Pickup the garbarge.
	IGoal pickup = createGoal("pick");
	dispatchSubgoalAndWait(pickup);

	SyncResultListener srl	= new SyncResultListener();
	Map params = new HashMap();
	params.put(ISpaceAction.ACTOR_ID, getComponentDescription());
	env.performSpaceAction("burn", params, srl);
	srl.waitForResult();
}
```

f. Plano *pickup_plan*:

Este plano consiste em pegar no "lixo" presente na posição do agente *Burner*.

```
public void body()
{

	IEnvironmentSpace env =
		(IEnvironmentSpace)getBeliefbase().getBelief("env").getFact();

	Map params = new HashMap();
	params.put(ISpaceAction.ACTOR_ID, getComponentDescription());
	SyncResultListener srl	= new SyncResultListener();
	env.performSpaceAction("pickup", params, srl);
	if(!((Boolean)srl.waitForResult()).booleanValue())
		fail();

}
```

g. Como já devem ter reparado, para que visualmente as alterações se processem é necessário definir ações do ambiente. Os planos até agora desenvolvidos já invocam estas ações, faltando agora defini-las.

i. Ação *PickupWasteAction*:

Esta ação estende *jadex.commons.SimplePropertyObject* e implementa a interface *ISpaceAction*, sendo por isso necessário implementar o método *perform().*

Neste método vamos verificar se existe "lixo" na posição atual do agente, e em caso afirmativo eliminamos esse "lixo" (no caso de haver mais do que um elemento deste tipo, eliminamos apenas o primeiro).

```
/**
 * Performs the action.
 * @param parameters parameters for the action
 * @param space the environment space
 * @return action return value
 */
public Object perform(Map parameters, IEnvironmentSpace space)
{
	boolean ret = false;

	Grid2D grid = (Grid2D)space;
	IComponentDescription owner =
		(IComponentDescription)parameters.get(ISpaceAction.ACTOR_ID);
	ISpaceObject so = grid.getAvatar(owner);

	assert so.getProperty("garbage") == null: so;

	Collection wastes =
				grid.getSpaceObjectsByGridPosition((IVector2)so.getPrope
			rty(Grid2D.PROPERTY_POSITION), "garbage");
	ISpaceObject waste = (ISpaceObject)(wastes!=null? wastes.iterator().next():
null);

	if(wastes!=null)
	{
		wastes.remove(waste);
		so.setProperty("garbage", waste);

		grid.setPosition(waste.getId(), null);
		ret = true;
	}

	return ret? Boolean.TRUE: Boolean.FALSE;
}
```

ii. Ação *DropWastAction:*

Esta ação faz o oposto da ação anterior, pousando o “lixo” na posição atual.

```
/**
* Performs the action.
* @param parameters parameters for the action
* @param space the environment space
* @return action return value
 */
public Object perform(Map parameters, IEnvironmentSpace space)
{

	Grid2D grid = (Grid2D)space;

	IComponentDescription owner =
		(IComponentDescription)parameters.get(ISpaceAction.ACTOR_ID);
	ISpaceObject so = grid.getAvatar(owner);
	IVector2 pos = (IVector2)so.getProperty(Grid2D.PROPERTY_POSITION);

	//If there is no garbage raise exception (if that is the case this
action shouldn't be called)
	assert so.getProperty("garbage") != null;

	ISpaceObject garb = (ISpaceObject)so.getProperty("garbage");
	grid.setPosition(garb.getId(), pos);
	so.setProperty("garbage", null);

	return null;
}
```

iii. Ação *Go*:

Esta ação está encarregue de mover o agente no mapa, calculando a nova posição com base nos parâmetros fornecidos.

```
public static final String UP = "up";
public static final String DOWN = "down";
public static final String LEFT = "left";
public static final String RIGHT = "right";
public static final String DIRECTION = "direction";
```

```
public Object perform(Map parameters, IEnvironmentSpace space)
{

	String dir = (String)parameters.get(DIRECTION);
	Object oid = parameters.get(ISpaceAction.OBJECT_ID);
	ISpaceObject obj = space.getSpaceObject(oid);
	IVector2 pos = (IVector2)obj.getProperty(Space2D.PROPERTY_POSITION);

	int px = pos.getXAsInteger();
	int py = pos.getYAsInteger();

	switch(dir){
	case UP:
		pos = new Vector2Int(px, py-1);
		break;
	case DOWN:
		pos = new Vector2Int(px, py+1);
		break;
	case LEFT:
		pos = new Vector2Int(px-1, py);
		break;
	case RIGHT:
		pos = new Vector2Int(px+1, py);
		break;
	}

	((Space2D)space).setPosition(oid, pos);
	obj.setProperty("lastmove", dir);

	return null;
}
```

iv. Ação *BurnWasteAction:*
Por fim só falta definir a ação de queimar o "lixo", fazendo desaparecer o lixo do ecrã.

```
public Object perform(Map parameters, IEnvironmentSpace space)
{
	Grid2D grid = (Grid2D)space;

	IComponentDescription owner =
		(IComponentDescription)parameters.get(ISpaceAction.ACTOR_ID);
	ISpaceObject so = grid.getAvatar(owner);

	assert so.getProperty("garbage")!=null;

	ISpaceObject garb = (ISpaceObject)so.getProperty("garbage");
	so.setProperty("garbage", null);
	space.destroySpaceObject(garb.getId());

	return null;
}
```

Resta-nos correr o nosso exemplo através do JCC, adicionando a pasta do *bin* do nosso projeto ao *path* e executando o ficheiro *GarbageCollector.application.xml*.

Observadores e Avaliação

Neste módulo do tutorial vamo-nos focar em introduzir, no exemplo dado no capítulo anterior, capacidades ao observador já existente e um avaliador para tornar possível a recolha de alguns dados do espaço, para uma posterior análise. Uma descrição sobre o que são observadores e avaliadores pode ser encontrada nos tópicos ***Observadores*** e ***Avaliador*** deste tutorial.

O código fonte pode ser encontrado em: http://jpsfs.com/data/GarbageCollector_evaluators.zip

1. Abra o projeto *GarbageCollector* criado no capítulo anterior deste tutorial;
2. No ficheiro *GarbageCollector.application.xml* adicione os seguintes *imports*:

```
<import>jadex.extension.envsupport.evaluation.*</import>
<import>jadex.extension.envsupport.observer.gui.plugin.*</import>
```

   O primeiro diz respeito ao módulo de *evaluators* do *Jadex* e o segundo ao *plugin* para a interface gráfica, de forma a podermos visualizar os dados recolhidos.
3. Começamos por adicionar o parâmetro *burned* ao agente *burner*, para que aí fiquem registados o número de pedaços de lixo que este queimou. Para isso, procure o nó *env:objecttype* com o atributo *name="burner"* e adicione o parâmetro *burned*. No final o agente deverá estar indicado da seguinte forma:

```
<env:objecttype name="burner">
	<env:property name="vision_range">0</env:property>
	<env:property name="burned">0</env:property>
</env:objecttype>
```

4. No interior do nó *configurations/extensions/env:envspace*/env:*observers* encontrará o *observer* definido no exemplo anterior. A este *observer* vamos adicionar um *plugin* que adicionará um gráfico à nossa janela. Vamos dar o nome ao gráfico de *burned_chart* (este nome será nome que iremos declarar no nó *dataconsumer* mais à frente). O *observer* ficará assim:

```
<env:observer name="world" dataview="view_all" perspective="abstract">
	<env:plugin name="evaluation" class="EvaluationPlugin">
		<env:property name="component_0">
((AbstractChartDataConsumer)$space.getDataConsumer("burned_chart")).getChartPanel()
		</env:property>
	</env:plugin>
</env:observer>
```

5. Vamos indicar ao nosso modelo quais são os dados disponíveis no espaço, e onde os podemos encontrar. Para isso adicionamos o nó *env:dataproviders* dentro do nó *env:envspace name="mygc2dspace"*.

```
<env:dataproviders>
	<env:dataprovider name="garbage_burned">
		<env:source name="$burner" objecttype="burner"/>
		<env:data name="time">$time</env:data>
		<env:data name="burned_amount">$burner.burned</env:data>
	</env:dataprovider>
</env:dataproviders>
```

   Definimos então um *dataprovider* com o nome *garbage_burned*, que tem como origem o

agente *burner* e onde vamos retirar informação sobre o tempo decorrido e a quantidade de lixo queimado.

6. De seguida vamos adicionar os *dataconsumers*, adicionando o nó *env:dataconsumers* dentro do nó *env:envspace name="mygc2dspace"*. Declaramos um *dataconsumer* do tipo *XYChartDataConsumer*, que vai buscar os dados ao *dataprovider* declarado no ponto anterior, e indica alguns dados para a construção do gráfico como a série que o compõe e as legendas dos eixos.

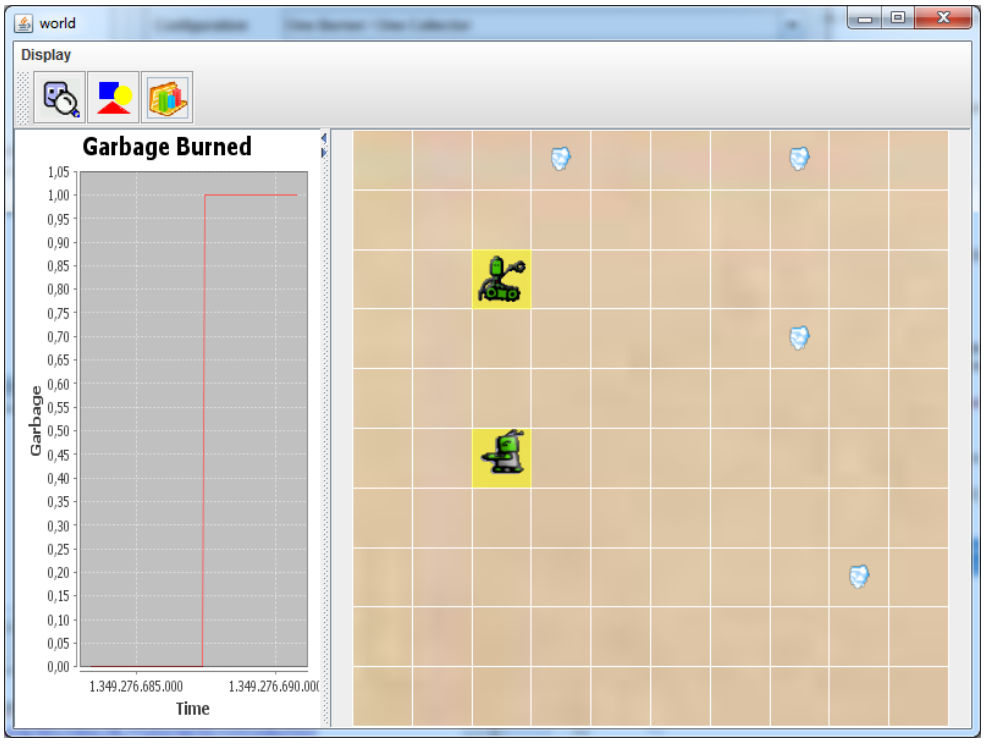
```
<env:dataconsumers>
	<env:dataconsumer name="burned_chart" class="XYChartDataConsumer">
		<env:property name="dataprovider">"garbage_burned"</env:property>
		<env:property name="title">"Garbage Burned"</env:property>
		<env:property name="labelx">"Time"</env:property>
		<env:property name="labely">"Garbage"</env:property>
		<env:property name="maxitemcount">100</env:property>
		<env:property name="legend">false</env:property>

		<!-- Serie -->
		<env:property name="seriesname">"Garbage"</env:property>
		<env:property name="valuex">"time"</env:property>
		<env:property name="valuey">"burned_amount"</env:property>
	</env:dataconsumer>
</env:dataconsumers>
```

Para este exemplo apenas foi declarado um *dataconsumer* mas mais poderão ser criados.

7. Para terminar, falta-nos atualizar o número de pedaços de lixo queimados pelo agente *burner*. Para tal vamos até ao ficheiro *BurnWasteAction*, e no final da função *perform()*, após destruir o objeto "lixo" do espaço, vamos atualizar o nosso contador.

```
int burned = (int)so.getProperty("burned");
so.setProperty("burned", burned + 1);
```

8. Resta-nos testar o nosso exemplo no JCC. Vão reparar que surgiu uma nova opção contento o gráfico declarado.



Materiais

- Aplicações, código fonte: http://jpsfs.com/data/PingPong.zip
- Espaços, código fonte: http://jpsfs.com/data/GarbageCollector.zip
- Observadores e Avaliação, código fonte: http://jpsfs.com/data/GarbageCollector_evaluators.zip

Referências

- [1] (20/09/2012) Jadex Documentation: http://www.activecomponents.org/bin/view/AC+Tutorial/01+Introduction
- [2] (20/09/2012) Jadex Documentation: http://www.activecomponents.org/bin/view/BDI+Tutorial/01+Introduction
- [3] (21/09/2012) Jadex Documentation: http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/Application+Guide/02+Simple+Applications
- [4] (21/09/2012) Jadex Documentation: http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/Environment+User+Guide/01+Introduction
- [5] (21/09/2012) SourceForge: http://sourceforge.net/projects/jadex/
- [6] (21/09/2012) Bruno Ferreira, Carlos Babo, Hélder Moreira (2011): Determinação de Percursos Usando Agentes BDI