INICIALIZAÇÃO AO JADE TUTORIAL

Robotics & Industrial Complex Systems

CRIAR UM PROJETO JADE EM NETBEANS

Um projeto JADE é na prática um projeto JAVA onde se adiciona uma biblioteca externa, neste caso o JADE. Portanto, o primeiro passo a ser realizado é a criação de um projeto em JAVA no Netbeans. O projeto criado não deverá ser criado com a *class main* automaticamente codificada pelo IDE (Netbeans).

Após a criação do projeto, falta-nos adicionar a biblioteca que permite a utilização e desenvolvimento de agentes JADE. (Na prática deve-se adicionar a biblioteca criada no tutorial onde é explicada a instalação do JADE e a configuração do Netbeans).

Para adicionar a biblioteca, clique com o botão direito no projeto, posteriormente em Propriedades (Properties), e aparecerá a seguinte janela, a baixo.

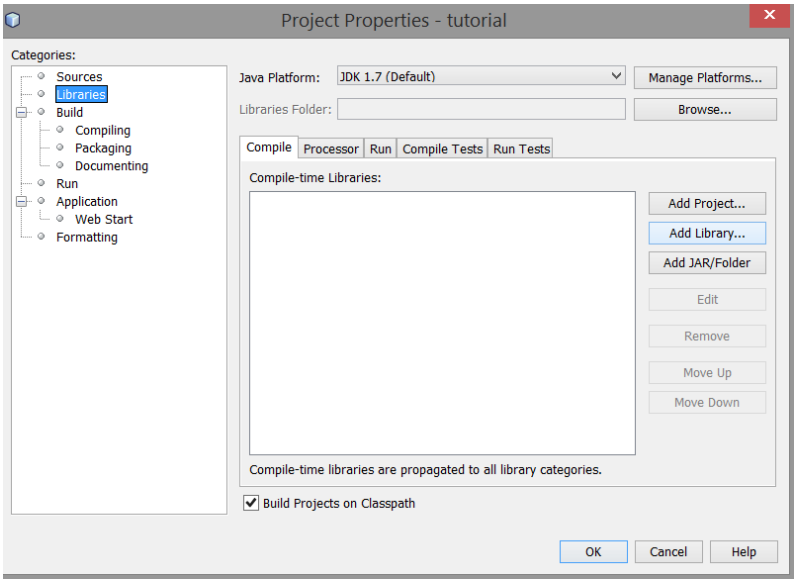

Figura 1 - Propriedades do projeto

Nas propriedades do projeto, selecione a categoria Bibliotecas (*Libraries*). Quando entrar nesta categoria selecione Adicionar Biblioteca (*Add Library…*).

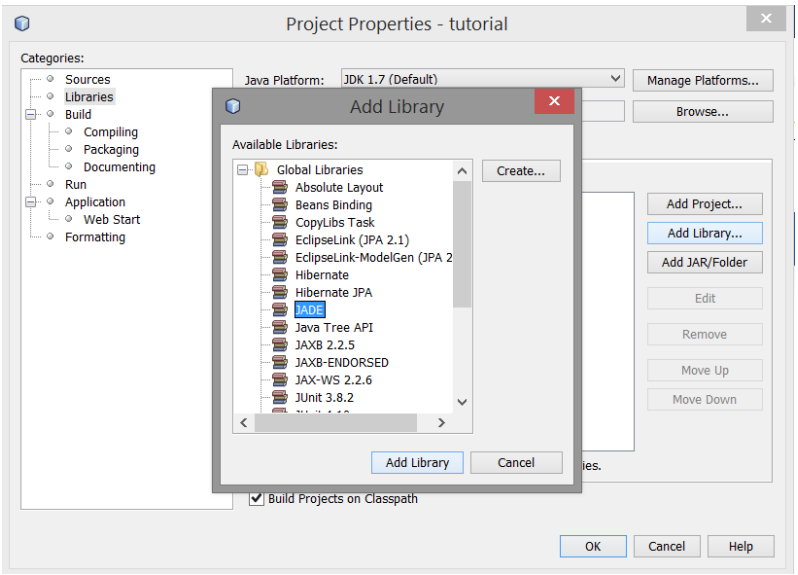

Figura 2 - Adicionar Biblioteca

Na nova janela deverá selecionar a biblioteca JADE, já criada e clicar em Adicional Biblioteca (*Add Library*). Após adicionar a biblioteca JADE o projeto está pronto para começar a ser utilizado como um projeto JADE.

CODIFICAR O PRIMEIRO AGENTE

Neste ponto é pedido que seja criado um primeiro agente muito simples, que apenas aparece e corre o método ***setup()***. Este método é o primeiro método que é chamado assim que um agente nasce, portanto deve ser utilizado para lançar os comportamentos que devem ser inicializados desde o inicio da execução do agente. Como tal crie uma nova ***class***, onde irá codificar o seu primeiro agente e faça o ***extends*** **Agent**. Não se esqueça de importar **jade.core.Agent**.

```
import jade.core.Agent;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        System.out.println("Hello World. ");
        System.out.println("My name is " + getLocalName());
    }
}
```

Figura 3 - Implementação do meu primeiro agente

No método ***setup()*** adicione dois ***println*** responsáveis por mostrar um ***Hello World*** com o nome do agente.

De forma a lançar a plataforma e um agente deste tipo é necessário alterar as propriedades do projeto. Depois de clicar com o botão direito em cima do projeto e abrir as propriedades, selecione Execução (*Run*). Aparecerá a seguinte janela, a baixo.

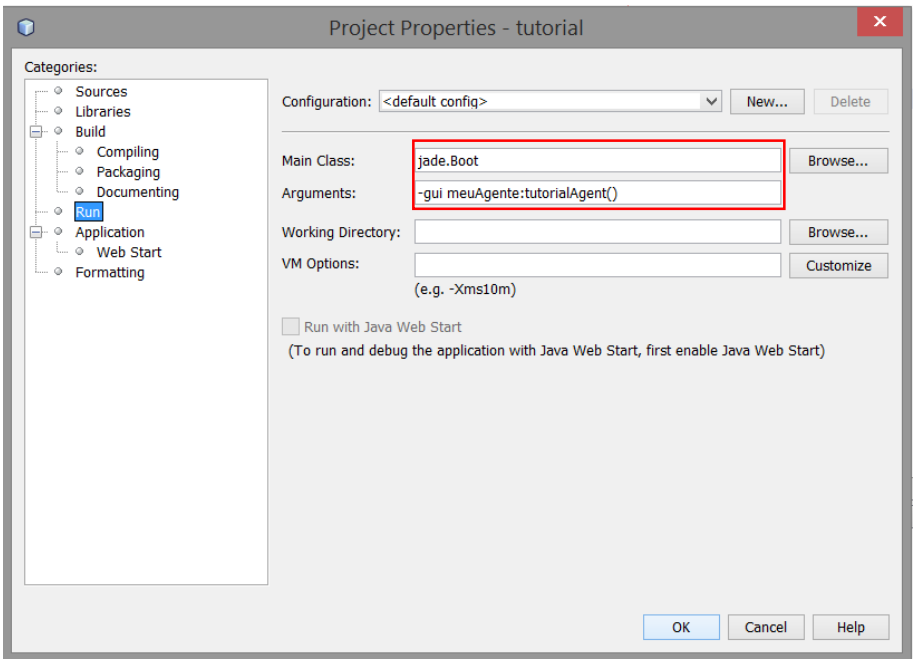

Figura 4 - Propriedades do projeto de forma a lançar o meu agente

Como *main class* é necessário definir **jade.Boot** e os argumentos devem ser **–gui** para lançar a plataforma e a interface gráfica e posteriormente o nome do agente separado por ":" e a *class* que define o tipo de agente que é, no nosso caso **meuAgente:tutorialAgent()**.

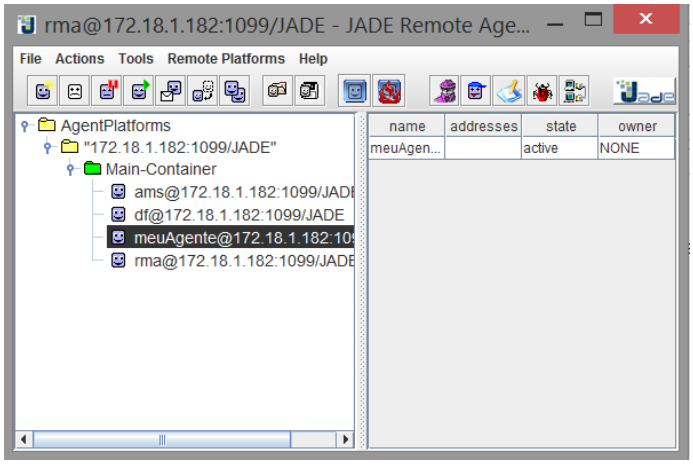

Figura 5 - Interface gráfica do JADE onde é possível visualizar o meuAgente

ADICIONANDO COMPORTAMENTOS AOS AGENTES

Tendo como ponto de partida o agente codificado no ponto anterior vão ser adicionados comportamentos simples, criando assim agentes mais realistas, pois os agentes devem ser codificados baseados em comportamentos, pois essa é a natureza dos mesmos.

ADICIONANDO UM *SIMPLE BEHAVIOUR*

O *Simple Behaviour* é responsável por executar uma determinada tarefa ciclicamente até que esta termine, como tal neste exemplo é pedido que se faça um *Simple Behaviour* que corra quatro vezes até terminar.

```
import jade.core.Agent;
import jade.core.behaviours.SimpleBehaviour;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        this.addBehaviour(new myBehaviour(this));
    }

    class myBehaviour extends SimpleBehaviour {

        int step = 0;

        public myBehaviour(Agent a) {
            super(a);
        }

        @Override
        public void action() {
            System.out.println(step);
            if(step++ == 3)
                finished = true;
        }
        private boolean finished = false;

        @Override
        public boolean done() {
            System.out.println("Terminei o Simple Behaviour");
            return finished;
        }
    }
}
```

Figura 6 - *Simple Behaviour*

ADICIONANDO UM *ONE SHOT BEHAVIOUR*

O *One Shot Behaviour* quando é adicionado é responsável por fazer uma tarefa uma única vez. Na prática é como se chamasse um *Simple Behaviour* mas em que o método *done()* faz logo *return true*, terminando assim a execução logo na primeira chamada. Implemente no agente utilizado como referência um *behaviour* deste género e verifique o resultado.

```
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        this.addBehaviour(new myBehaviour());
    }

    class myBehaviour extends OneShotBehaviour {
        @Override
        public void action() {
            System.out.println("OneShotBehaviour: Disparou o behaviour");
        }
    }
}
```

Figura 7 - *One Shot Behaviour*

ADICIONANDO UM *WAKER BEHAVIOUR*

O *Waker Behaviour* tem um comportamento idêntico ao anteriormente apresentado com a diferença que quando este é chamado recebe como parâmetro um valor do tipo *long* que indica o tempo (em milissegundos) de espera até que este comportamento seja disparado. Ou seja, se lançarmos um comportamento deste género com 1000 no parâmetro no construtor deste *behaviour* este ficará "adormecido" durante 1000 milissegundos até ser disparado e correr uma vez.

```
import jade.core.Agent;
import jade.core.behaviours.WakerBehaviour;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        this.addBehaviour(new myBehaviour(this, 1000));
    }

    class myBehaviour extends WakerBehaviour {
        public myBehaviour(Agent a, long timeout) {
            super(a, timeout);
            System.out.println("Start - " + System.currentTimeMillis());
        }
        @Override
        protected void onWake() {
            System.out.println("End - " + System.currentTimeMillis());
        }
    }
}
```

Figura 8 - *Waker Behaviour*

ADICIONANDO UM *CYCLIC BEHAVIOUR*

Cada agente é uma *Thread* e como tal se durante a execução de um comportamento este demorar muito tempo a processar informação ou a realizar uma determinada tarefa, os restantes *behaviours* ficam bloqueados e o agente também "congelado". Como tal os ciclos codificados dentro dos comportamentos devem ser implementados recorrendo a *Cyclic Behaviours*, que em cada chamada, correm uma iteração, ciclicamente.

```
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        this.addBehaviour(new myBehaviour());
    }

    class myBehaviour extends CyclicBehaviour {
        @Override
        public void action() {
            System.out.println("CyclicBehaviour: Disparou o behaviour");
        }
    }
}
```

Figura 9 - *Cyclic Behaviour*

ADICIONANDO UM *TICKER BEHAVIOUR*

O *Ticker Behaviour* permite correr uma tarefa periodicamente, ou seja, à semelhança ao que acontece com o *Cyclic Behaviour*, este arranca e corre ciclicamente desde a instanciação até que o agente deixa a plataforma. A diferença deste *behaviour* para o anterior é que existe uma janela de tempo entre execuções. Experimente a utilização deste *behaviour* com o seguinte código.

```
import jade.core.Agent;
import jade.core.behaviours.TickerBehaviour;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        this.addBehaviour(new myBehaviour(this, 2000));
    }

    class myBehaviour extends TickerBehaviour {

        public myBehaviour(Agent a, long period) {
            super(a, period);
        }

        @Override
        protected void onTick() {
            System.out.println("TickerBehaviour: Disparou o behaviour");
        }
    }
}
```

Figura 10 - *Ticker Behaviour*

COMUNICAÇÕES ENTRE AGENTES

A comunicação entre agentes é um dos pontos mais importantes na implementação de plataformas deste género. O facto de serem entidades distribuídas e que se encontram inseridas numa sociedade de agentes em que cooperam e que partilham informação entre si, torna a implementação dos comportamentos de comunicação essenciais.

A plataforma JADE permite criar agentes regidos pelos protocolos de comunicação criados e geridos pela FIPA (*Foundation for Intelligent Physical Agents*). Como tal a implementação das comunicações entre agentes JADE usualmente é feita recorrendo a estes protocolos de comunicação como referência.

PROTOCOLO *FIPA REQUEST*

O protocolo *FIPA Request* deve ser utilizado quando um agente pretende fazer uma comunicação ponto a ponto com outro agente, tendo em vista o pedido de execução de alguma tarefa ou pedido de informação referente a esse agente.

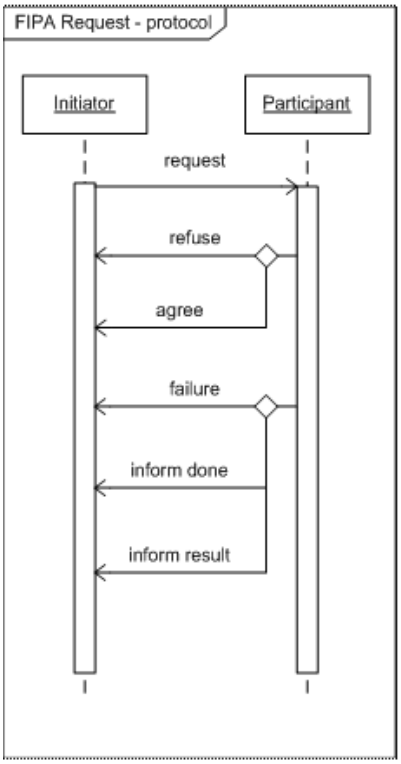

Figura 11 - Protocolo *FIPA Request*

Todas as mensagens trocadas têm associada uma performativa que define em que ponto do protocolo se encontra. Por exemplo, a primeira mensagem enviada pelo *Initiator* tem como performativa ***Requet***.

De forma a implementar um protocolo deste género é necessário que cada um dos agentes intervenientes implemente um comportamento responsável por enviar e receber as mensagens respetivamente. De forma a implementar esta comunicação o agente que inicia a comunicação deve implementar um *behaviour* do tipo *AchieveREInititor*, enquanto o agente que recebe processa o pedido necessita de implementar um comportamento do tipo *AchieveREResponder*.

De forma a testar uma implementação deste protocolo de comunicação deve implementar dois tipos de agente, um que terá o *behaviour* responsável por inicializar o protocolo e outro com o *behaviour* que processa o pedido.

AGENTE QUE INICIA A COMUNICAÇÃO – *ACHIEVEREINITIATOR*

Implemente o seguinte agente com um comportamento responsável por inicializar um pedido.

```
import jade.core.AID;
import jade.core.Agent;
import jade.lang.acl.ACLMessage;
import jade.proto.AchieveREInitiator;

public class initiatorAgent extends Agent{

    @Override
    protected void setup() {
        ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
        msg.addReceiver(new AID("responder", false));
        this.addBehaviour(new initiator(this, msg));
    }

    private class initiator extends AchieveREInitiator{

        public initiator(Agent a, ACLMessage msg) {
            super(a, msg);
        }

        @Override
        protected void handleAgree(ACLMessage agree) {
            System.out.println(myAgent.getLocalName() + ": AGREE message received");
        }

        @Override
        protected void handleInform(ACLMessage inform) {
            System.out.println(myAgent.getLocalName() + ": INFORM message received");
        }
    }
}
```

Figura 12 – *AchieveREInitiator*

AGENTE QUE PROCESSA O PEDIDO – *ACHIEVERERESPONDER*

Implemente agora o agente com um comportamento responsável por processar o pedido.

```
import jade.core.Agent;
import jade.domain.FIPAAgentManagement.FailureException;
import jade.domain.FIPAAgentManagement.NotUnderstoodException;
import jade.domain.FIPAAgentManagement.RefuseException;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.proto.AchieveREResponder;

public class responderAgent extends Agent{

    @Override
    protected void setup() {
        this.addBehaviour(new responder(this, MessageTemplate.MatchPerformative(ACLMessage.REQUEST)));
    }

    private class responder extends AchieveREResponder{

        public responder(Agent a, MessageTemplate mt) {
            super(a, mt);
        }

        @Override
        protected ACLMessage handleRequest(ACLMessage request) throws NotUnderstoodException, RefuseException {
            System.out.println(myAgent.getLocalName() + ": Processing REQUEST message");
            ACLMessage msg = request.createReply();
            msg.setPerformative(ACLMessage.AGREE);
            return msg;
        }

        @Override
        protected ACLMessage prepareResultNotification(ACLMessage request, ACLMessage response) throws FailureException {
            System.out.println(myAgent.getLocalName() + ": Preparing result of REQUEST");
            block(5000);
            ACLMessage msg = request.createReply();
            msg.setPerformative(ACLMessage.INFORM);
            return msg;
        }
    }
}
```

Figura 13 - *AchieveREResponder*

LANÇAR O TESTE DA COMUNICAÇÃO

Para lançar um agente de cada tipo anteriormente codificados, por favor mude as propriedades do projeto no *Run*. Ficando com o seguinte aspeto.

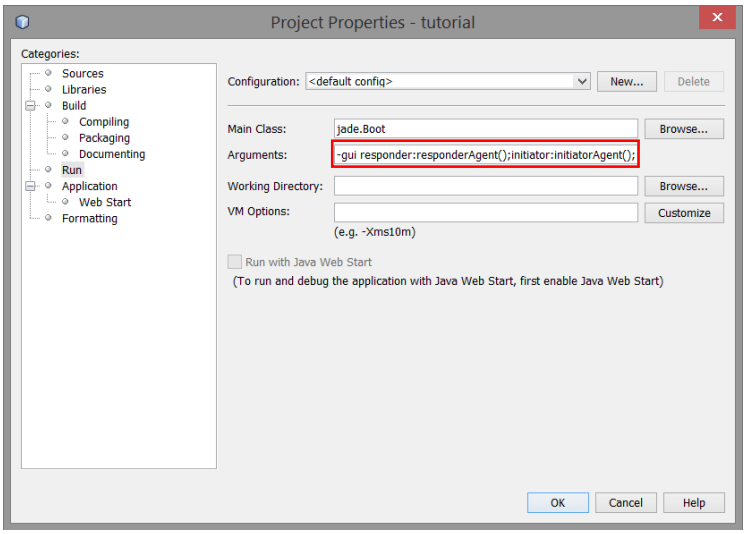

Figura 14 - Propriedades do projeto, de forma a lançar os dois agentes

Nota: Tenha em atenção que o nome do agente que responde ao pedido, no nosso exemplo, tem de ser obrigatoriamente **responder**.

PROTOCOLO *FIPA CONTRACTNET*

O Protocolo *FIPA Contract Net* é utilizado quando se pretende inicializar uma negociação prévia, antes do pedido de execução. Ou seja, quando se pretende negociar com todos os agentes que conseguem executar o que é pretendido e posteriormente decidir aquele ou aqueles que melhor satisfazem as necessidades do agente que inicializou a negociação.

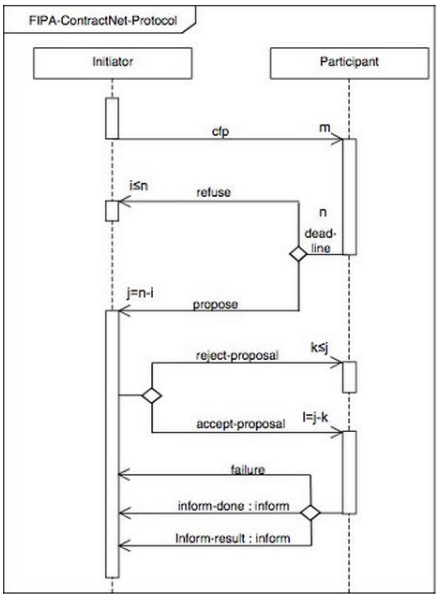

Figura 15 - Protocolo *FIPA Contract Net*

À semelhança do que foi feito no protocolo anterior cada *initiator* ou *responder* tem de ser codificado sob a forma de comportamento.

AGENTE QUE INICIA A NEGOCIAÇÃO – CONTRACTNETINITIATOR

```
import jade.core.AID;
import jade.core.Agent;
import jade.lang.acl.ACLMessage;
import jade.proto.ContractNetInitiator;
import java.util.Vector;

public class initiatorAgent extends Agent{

    @Override
    protected void setup() {
        ACLMessage msg = new ACLMessage(ACLMessage.CFP);
        msg.addReceiver(new AID("responder", false));
        this.addBehaviour(new initiator(this, msg));
    }

    private class initiator extends ContractNetInitiator{

        public initiator(Agent a, ACLMessage msg) {
            super(a, msg);
        }

        @Override
        protected void handleInform(ACLMessage inform) {
            System.out.println(myAgent.getLocalName() + ": INFORM message received");
        }

        @Override
        protected void handleAllResponses(Vector responses, Vector acceptances) {
            System.out.println(myAgent.getLocalName() + ": All PROPOSALS received");
            ACLMessage auxMsg = (ACLMessage)responses.get(0);
            ACLMessage reply = auxMsg.createReply();
            reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
            acceptances.add(reply);
        }
    }
}
```

Figura 16 - *Contract Net Initiator*

AGENTES QUE RESPONDEM AO PEDIDO DE PROPOSTAS – CONTRACTNETRESPONDER

```
import jade.core.Agent;
import jade.domain.FIPAAgentManagement.FailureException;
import jade.domain.FIPAAgentManagement.NotUnderstoodException;
import jade.domain.FIPAAgentManagement.RefuseException;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.proto.ContractNetResponder;

public class responderAgent extends Agent{

    @Override
    protected void setup() {
        this.addBehaviour(new responder(this, MessageTemplate.MatchPerformative(ACLMessage.CFP)));
    }

    private class responder extends ContractNetResponder{

        public responder(Agent a, MessageTemplate mt) {
            super(a, mt);
        }

        @Override
        protected ACLMessage handleCfp(ACLMessage cfp) throws RefuseException, FailureException, NotUnderstoodException {
            System.out.println(myAgent.getLocalName() + ": Processing CFP message");
            ACLMessage msg = cfp.createReply();
            msg.setPerformative(ACLMessage.PROPOSE);
            msg.setContent("My Proposal value");
            return msg;
        }

        @Override
        protected ACLMessage handleAcceptProposal(ACLMessage cfp, ACLMessage propose, ACLMessage accept) throws FailureException {
            System.out.println(myAgent.getLocalName() + ": Preparing result of CFP");
            block(5000);
            ACLMessage msg = cfp.createReply();
            msg.setPerformative(ACLMessage.INFORM);
            return msg;
        }
    }
}
```

Figura 17 - *Contract Net Responder*

LANÇAR O TESTE DA COMUNICAÇÃO

Para testar este protocolo de comunicação com apenas dois agentes, um *initiator* e um *responder*, deverá manter as propriedades do projeto já definidas nos testes do protocolo anterior.

UTILIZAÇÃO DO DF – *DIRECTORY FACILITATOR*

O DF no JADE funciona como o serviço de páginas amarelas, permitindo que todos os agentes se registem com o seu identificador (AID) e com os serviços que podem oferecer a outros agentes. Desta forma qualquer agente durante a execução, consegue procurar pelos agentes que oferecem e prestam os serviços que este necessita.

INSCRIÇÃO NO DF

A inscrição no DF deve ser o primeiro passo a ser executado por um agente, para que este possa ser procurado pelos outros agentes presentes na plataforma. Desta forma a inscrição deve ser feita no método *setup()* de forma a ser realizada logo na instanciação do agente. No exemplo, a baixo, o agente tutorial regista-se com o seu identificador e oferecendo o serviço "tutorial".

```
import jade.core.Agent;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(this.getAID());
        ServiceDescription sd  = new ServiceDescription();
        sd.setType("tutorial");
        sd.setName( getLocalName() );
        dfd.addServices(sd);
        try {
            DFService.register(this, dfd );
        } catch (FIPAException ex) {
            Logger.getLogger(tutorialAgent.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Figura 18 - Inscrição no DF

PROCURA NO DF

A procura no DF permite que qualquer agente em qualquer altura durante a execução procure por todos os agentes que oferecem um determinado serviço. Esta característica é importante devido à natureza deste tipo de plataformas, em que as entidades interagem frequentemente pedindo informação e realização de tarefas que podem e devem ser descritas como serviços oferecidos no DF. De forma a testar esta funcionalidade, codifique um *One Shot Behaviour* que é adicionado no *setup()* após a inscrição no DF para procurar e imprimir os agentes inscritos no DF que oferecem o serviço "tutorial".

```
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(this.getAID());
        ServiceDescription sd = new ServiceDescription();
        sd.setType("tutorial");
        sd.setName(getLocalName());
        dfd.addServices(sd);
        try {
            DFService.register(this, dfd);
        } catch (FIPAException ex) {
            Logger.getLogger(tutorialAgent.class.getName()).log(Level.SEVERE, null, ex);
        }
        this.addBehaviour(new searchInDF());
    }

    private class searchInDF extends OneShotBehaviour {

        @Override
        public void action() {
            DFAgentDescription dfd = new DFAgentDescription();
            ServiceDescription sd = new ServiceDescription();
            sd.setType("tutorial");
            dfd.addServices(sd);
            DFAgentDescription[] result = null;
            try {
                result = DFService.search(myAgent, dfd);
            } catch (FIPAException ex) {
                Logger.getLogger(tutorialAgent.class.getName()).log(Level.SEVERE, null, ex);
            }
            System.out.println(result[0].getName());
        }
    }
}
```

Figura 19 - Procura no DF

COMPORTAMENTOS COMPLEXOS

Para além dos comportamentos simples apresentados anteriormente, o JADE permite criar comportamentos complexos, combinando os comportamentos simples de forma sequencial, paralela ou sob a forma de máquina de estados.

Nesta secção serão apresentados exemplos de como criar estes comportamentos complexos.

SEQUENTIAL BEHAVIOUR

O *Sequential Behaviour* permite criar um comportamento constituído por sub comportamentos que são executados de forma sequencial. Este comportamento complexo é responsável por coordenar a execução dos sub comportamentos que o constituem.

```
import jade.core.Agent;
import jade.core.behaviours.SequentialBehaviour;
import jade.core.behaviours.SimpleBehaviour;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        SequentialBehaviour sb = new SequentialBehaviour();
        sb.addSubBehaviour(new simpleBeh(this, "SB1"));
        sb.addSubBehaviour(new simpleBeh(this, "SB2"));
        sb.addSubBehaviour(new simpleBeh(this, "SB3"));
        this.addBehaviour(sb);
    }

    private class simpleBeh extends SimpleBehaviour {

        private boolean finished = false;
        int step = 0;
        String printOut;

        public simpleBeh(Agent a, String prtOut) {
            super(a);
            this.printOut = prtOut;
        }

        @Override
        public void action() {
            System.out.println("SimpleBehaviour: SubBehaviour: " + printOut + " - step: " + ++step);
            if(step==3)
                finished = true;
        }

        @Override
        public boolean done() {
            return finished;
        }
    }
}
```

Figura 20 - *Sequential Behaviour*

PARALLEL BEHAVIOUR

Quando for pretendido que alguns comportamentos corram em paralelo é possível constituir um *Parallel Behaviour*. Num comportamento complexo deste tipo os sub comportamentos correm em paralelo e o *Parallel Behaviour* pode ser terminado de duas maneiras possíveis, quando todos os sub comportamentos terminarem ou quando um deles chegar ao fim. Esta caraterística é definida quando se instancia um novo *Parallel Behaviour*.

```
import jade.core.Agent;
import jade.core.behaviours.ParallelBehaviour;
import jade.core.behaviours.SimpleBehaviour;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        ParallelBehaviour pb = new ParallelBehaviour(ParallelBehaviour.WHEN_ALL);
        pb.addSubBehaviour(new simpleBeh(this, "SB1"));
        pb.addSubBehaviour(new simpleBeh(this, "SB2"));
        pb.addSubBehaviour(new simpleBeh(this, "SB3"));
        this.addBehaviour(pb);
    }

    private class simpleBeh extends SimpleBehaviour {

        private boolean finished = false;
        int step = 0;
        String printOut;

        public simpleBeh(Agent a, String prtOut) {
            super(a);
            this.printOut = prtOut;
        }

        @Override
        public void action() {
            System.out.println("SimpleBehaviour: SubBehaviour: " + printOut + " - step: " + ++step);
            if(step==3)
                finished = true;
        }

        @Override
        public boolean done() {
            return finished;
        }
    }
}
```

Figura 21 - *Parallel Behaviour*

O JADE também permite construir uma máquina de estados como comportamento complexo. Cada um dos estados da máquina de estados finita é associado a um comportamento e todas as transições possíveis para cada estado são definidas aquando da instanciação e criação da máquina de estados. De forma a poder escolher qual o próximo estado, de entre as possíveis transições, o método *onEnd()* deve retornar o inteiro que identifica a transição desejada.

```
import jade.core.Agent;
import jade.core.behaviours.FSMBehaviour;
import jade.core.behaviours.SimpleBehaviour;

public class tutorialAgent extends Agent {

    @Override
    protected void setup() {
        FSMBehaviour fsmb = new FSMBehaviour();
        fsmb.registerFirstState(new simpleBeh(this, "A"), "St_A");
        fsmb.registerState(new simpleBeh(this, "B"), "St_B");
        fsmb.registerState(new simpleBeh(this, "C"), "St_C");
        fsmb.registerLastState(new simpleBeh(this, "D"), "St_D");
        fsmb.registerTransition("St_A", "St_B", 0);
        fsmb.registerTransition("St_B", "St_C", 1);
        fsmb.registerTransition("St_C", "St_D", 2);
        addBehaviour(fsmb);
    }

    private class simpleBeh extends SimpleBehaviour {

        private boolean finished = false;
        int step = 0;
        String currentState;

        public simpleBeh(Agent a, String crtSta) {
            super(a);
            this.currentState = crtSta;
        }

        @Override
        public void action() {
            System.out.println("SimpleBehaviour: SubBehaviour: " + currentState + " - step: " + ++step);
            if(step==3)
                finished = true;
        }

        @Override
        public boolean done() {
            return finished;
        }

        @Override
        public int onEnd() {
            switch(currentState){
                case "A": return 0;
                case "B": return 1;
                case "C": return 2;
            } return -1;
        }

    }
}
```

Figura 22 - *FSM Behaviour*

AUTORES:

André Rocha – andre.rocha@uninova.pt

José Barata – jab@uninova.pt