Department of Artificial Intelligence
E.T.S. Ingenieros Informáticos
Universidad Politécnica de Madrid

Sistemas Inteligentes

Agentes JADE

Javier Bajo
Catedrático de Universidad
jbajo@fi.upm.es

Index

Index

4. Creación de Agentes
Introducción

- Los agentes JADE se instancian a partir de la clase **jade.core.Agent**
- **A modo de resumen**, lo que vamos a ver en este tema es:

1. Un nuevo agente que se cree debe siempre **heredar** de la clase **jade.core.Agent**, es decir, hereda las propiedades y métodos ya definidos para un agente JADE.

2. Un agente debe implementar al menos el método **setup()** que es el **constructor** del agente.

3. Un agente puede incorporar una llamada a su **finalización**. El agente permanece ejecutándose hasta que se invoca el método **doDelete()** (que a su vez invoca al método **takeDown()**, éste último puede utilizarse para ejecutar operaciones antes de la muerte del agente).

4. Un agente puede **recibir parámetros** al ser creado. Para ello utiliza el método **getArguments()** que devuelve una lista de objetos Object [].

4. Creación de Agentes
Introducción

5. Los agentes JADE tienen **comportamientos** en los que pueden llevar a cabo acciones o percepciones. Existe una clase que permite instanciar comportamientos en un agente:

   jade.core.behaviours.Behaviour

   Un comportamiento tiene 2 métodos principales:

   - **action()**: Se ejecuta cada vez que se llama al comportamiento.

   - **done()**: Comprueba si el comportamiento ha terminado.

6. Un agente puede enviar y recibir **mensajes**.

7. Un agente puede ofrecer y consumir **servicios** que se registran y acceden a través del Directory Facilitator.

Index

4. Creación de Agentes
La Clase Agent

- Para crear un agente sólo es necesario heredar de la clase Agent.
- Habitualmente cada agente registra varios servicios dentro de la plataforma.
- Cada servicio debería de ser implementado por uno o más comportamientos.

```
public class MiAgente extends Agent
{
	public void setup()
	{
		//Crear servicios proporcionados por el agente y
		registrarlos en el DF
		//añadir comportamientos
	}

	//Implementación de Comportamientos
}
```

4. Creación de Agentes
Ejemplo: Agente Básico

AgBasico.java

Creación de un agente que escribe un mensaje en pantalla: Primer Agente JADE

```
package es.upm.ejemplo;
import jade.core.Agent;

public class AgBasico extends Agent
{
		protected void setup()
		{
			System.out.println("Primer Agente JADE");
		}
}
```

Ejecución:

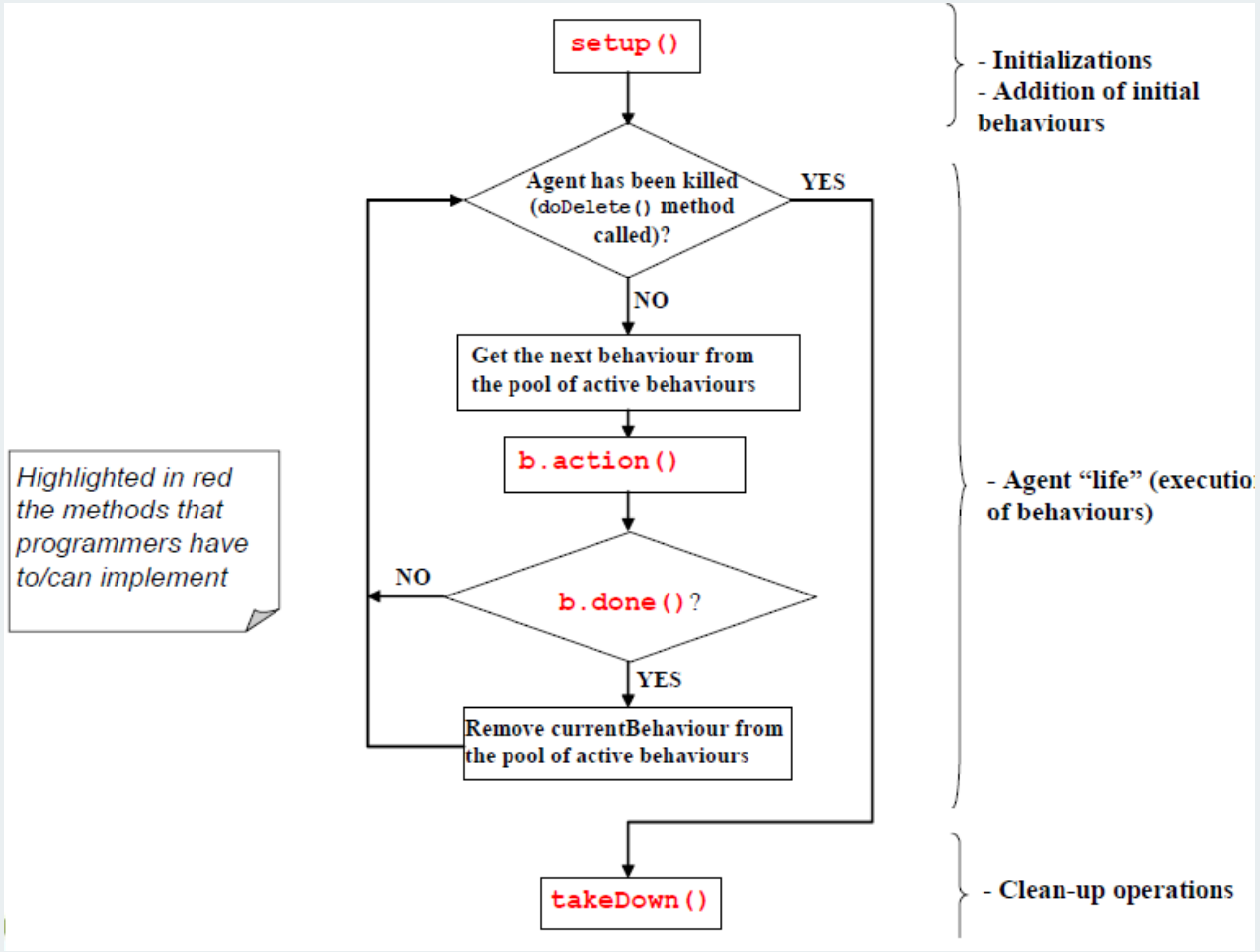
jade.Boot -gui agBasico:es.upm.ejemplo.AgBasico

4. Creación de Agentes
La Clase Agent

- **Ciclo de Vida de un agente**
  - Un agente tiene un ciclo de vida compuesto de tres etapas principales:
    - **Creación**
      - Dentro del método **setup**() se inicializan los distintos parámetros y objetos del agente, así como los comportamientos que ejecutará el agente.
      - Se ejecuta una sola vez.
    - **Vida**
      - Mientras el agente siga vivo, se ejecutan los comportamientos que tenga disponibles y que se encuentren activos.
      - Para ello es necesario gestionar una **lista de comportamientos**, así como gestionar el **estado de los comportamientos**.
      - Cuando un comportamiento está activo y es el primero de la lista de comportamientos, se ejecuta su método **action()** de dicho comportamiento
      - Tras la ejecución el método action() se ejecuta el método **done()** del comportamiento para comprobar si el comportamiento ha finalizado o si sigue en la lista de comportamientos.
      - Se ejecuta mientras existan comportamientos activos.
    - **Muerte**
      - Se elimina el agente ejecutando el método **takedown()**
      - Se ejecuta una sola vez.

4. Creación de Agentes
La Clase Agent

- **Ciclo de Vida de un agente**

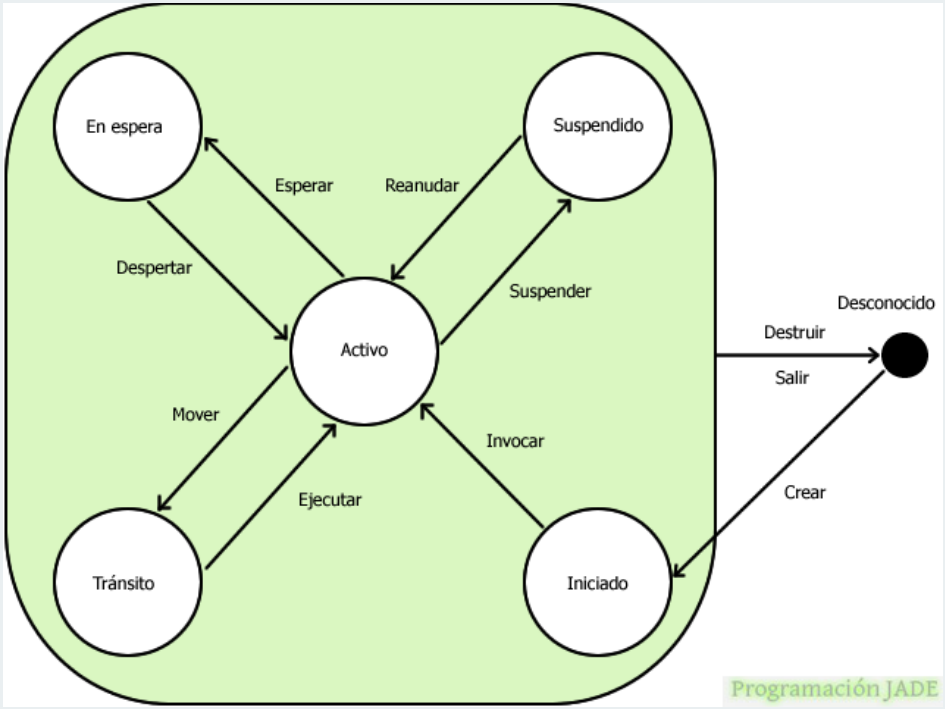


4. Creación de Agentes
La Clase Agent

- **Estados de un agente**. A lo largo de su vida, un agente puede pasar por distintos estados:
  - **Initiated/Iniciado**: el agente se ha creado pero no se ha registrado todavía en el AMS de la plataforma, no tiene nombre y no se puede comunicar con otros agentes.
  - **Active/Activo**: el agente ya se ha registrado y posee nombre. Se puede comunicar con otros agentes.
  - **Suspended/Suspendido**: el agente se encuentra parado porque su hilo de ejecución se encuentra suspendido (esta detenido, no ejecuta ningún comportamiento).
  - **Waiting/Esperando**: se encuentra bloqueado a la espera de un suceso. El agente se encuentra dormido en un monitor java a la espera de que se produzca un suceso que lo despierte, por ejemplo la llegada de un mensaje.
  - **Deleted/Eliminado**: el agente ha terminado por tanto el hilo terminó su ejecución y ya no estará más en el AMS.
  - **Transit/Transito**: el agente se está migrando a una nueva ubicación.

4. Creación de Agentes
La Clase Agent

- Estados de un agente JADE



- El intercambio de estados (o transición de estados) se realiza de modo automático o mediante la ejecución de métodos de la clase Agent que nos permiten controlar el estado del agente (doWait(), doSuspend(), doDelete(), doWake(), ...). Muchas de estas operaciones solamente pueden ser ejecutas por el agente AMS.

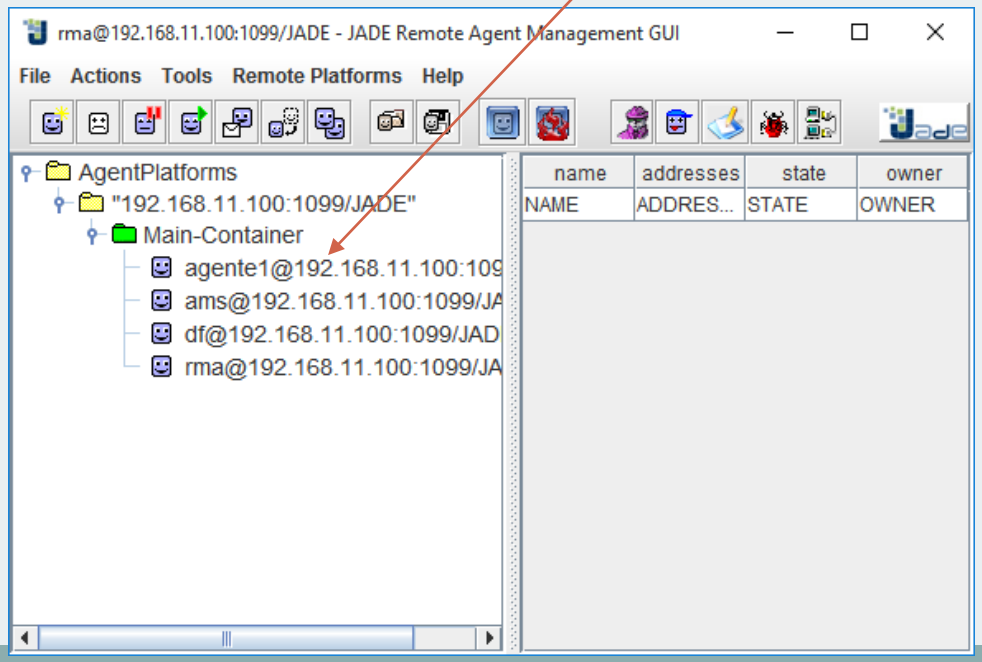
4. Creación de Agentes
La Clase Agent

- Cuando se crea una instancia de la Clase Agent se producen los siguientes pasos:
  1. Se ejecuta el constructor de la clase
  2. Se le asigna un AID (jade.core.AID), que es un identificador único formado por el nombre del agente + su dirección. Es posible conseguir el nombre de un agente con el método getAID() de la clase Agent.
  3. Se le asigna un nombre único global, se concatena el nombre del contenedor al asignado por el programador. El nombre es del tipo: <nombre_local>@<nombre_plataforma>. El nombre de la platforma es <main-host>:<main-port>/JADE por defecto.
  4. Se registra el agente en el AMS
  5. Se ejecuta el método setup()
     - En este método se incorporan todos los comportamientos del agente mediante el método addBehaviour(Behaviour)

4. Creación de Agentes
La Clase Agent

- El nombre único o AID de un agente JADE se genera a partir de los siguientes elementos:
  - Nombre del agente
  - @
  - Dirección del host (normalmente una dirección IP)
  - Puerto (normalmente 1099)
  - /JADE

NombreAgente@hostname:puerto/JADE



4. Creación de Agentes
Ejemplo: Agente Básico

AgBasico.java

Creación de un agente que escribe un mensaje en pantalla: Primer Agente JADE y otro mensaje con el AID del agente, entra en estado de espera y posteriormente en estado de suspensión.

```
package es.upm.ejemplo;
import jade.core.Agent;

public class AgBasico extends Agent
{
        protected void setup()
        {
                System.out.println("Primer Agente JADE");
                System.out.println("AID: " + this.getAID());

                System.out.println("Entrando en espera");
                this.doWait(10000);
                System.out.println("Saliendo de espera, entrando en suspendido");
                this.doSuspend();
                System.out.println("Saliendo de suspendido");
        }
}
```

4. Creación de Agentes
Creación de un agente

Paso de argumentos:

- Es posible pasar argumentos de entrada a un agente:

```
jade.Boot ..... agente:paquete.Agente(arg1 arg2)
```

- Es posible recuperar los argumentos a través del método getArguments() de la clase Agent.

```
Object[] args = getArguments();

if (args != null) {
        for (int i=0; i<args.length; ++i){
        }
}
```

4. Creación de Agentes
Ejemplo: Agente Básico con Parámetros

AgBasicoParams.java

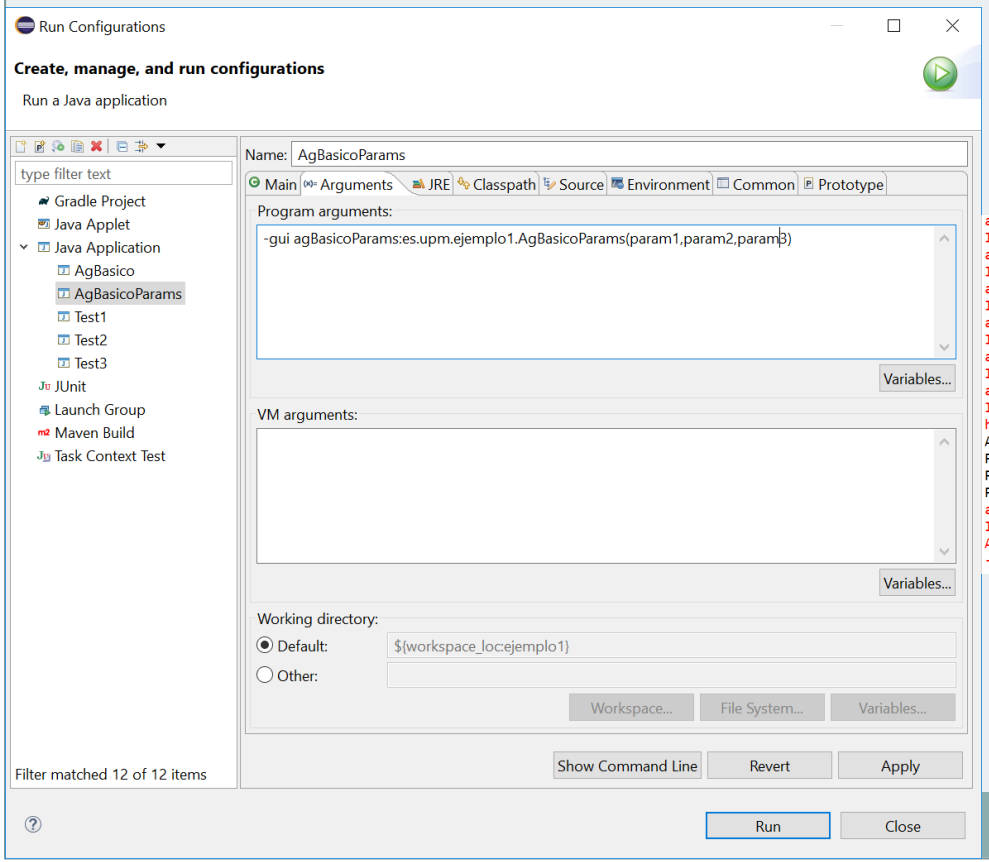
Agente básico al que se le pasan parámetros de entrada y los escribe por pantalla

```
package es.upm.ejemplo;
import jade.core.*;
public class AgBasicoParams extends Agent
{
		protected void setup()
		{
			Object [] listaparametros = getArguments();
			if ((listaparametros == null) || (listaparametros.length < 1))
			{
			  System.out.println("No se han introducido parametros");
			}
			 else
			 {
			  System.out.println("Agente JADE con Parametros: Soy el agente " + getLocalName());
			  int i;
			  for (i=0;i<listaparametros.length;i++)
					System.out.println("Parametro " + i + " es: " + (String) listaparametros[i]);
			 }
		}
}
```

4. Creación de Agentes
Ejemplo: Agente Básico con Parámetros

AgBasicoParams.java

Ejecución:
jade.Boot -gui agBasicoParams:es.upm.ejemplo.AgBasicoParams(1,2,3)




```
abr 25, 2019 10:45:22 PM jade.core.BaseService init
INFORMACIÓN: Service jade.core.messaging.Messaging initialized
abr 25, 2019 10:45:22 PM jade.core.BaseService init
INFORMACIÓN: Service jade.core.resource.ResourceManagement initialized
abr 25, 2019 10:45:22 PM jade.core.BaseService init
INFORMACIÓN: Service jade.core.mobility.AgentMobility initialized
abr 25, 2019 10:45:22 PM jade.core.BaseService init
INFORMACIÓN: Service jade.core.event.Notification initialized
abr 25, 2019 10:45:22 PM jade.mtp.http.HTTPServer <init>
INFORMACIÓN: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
abr 25, 2019 10:45:22 PM jade.core.messaging.MessagingService boot
INFORMACIÓN: MTP addresses:
http://192.168.1.41:7778/acc
Agente JADE con Parametros: Soy el agente agBasicoParams
Parametro 0 es: 1
Parametro 1 es: 2
Parametro 2 es: 3
abr 25, 2019 10:45:23 PM jade.core.AgentContainerImpl joinPlatform
INFORMACIÓN: ----------------------------------------
Agent container Main-Container@192.168.1.41 is ready.
------------------------------------------------
```

4. Creación de Agentes
Creación de un agente

- Creación de un agente
  - Se realizan varias tareas de forma automática:
    - Se llama al método setup().
    - Se crea un identificador (AID).
    - Se registra el agente en el AMS.
    - Se ejecuta el método setup(), que únicamente debe contener código relativo a tareas de inicialización.
  - El método setup() del agente puede utilizarse para:
    - Modificar el registro del AMS.
    - Registrar el agente en el DF.
    - Añadir las tareas que ejecutará el agente.
    - Etc.

4. Creación de Agentes
Terminación de un agente

- Terminación de un agente
  - Un agente acaba cuando se invoca al método doDelete().
  - Desde el método doDelete() se invoca al método takeDown(), que incorpora operaciones de limpieza.
  - Ejemplo: Agente Hola Mundo

```
protected void setup() {
  System.out.println("Hallo World! my name is "+getAID().getName());
  Object[] args = getArguments();
  if (args != null) {
    System.out.println("My arguments are:");
    for (int i = 0; i < args.length; ++i) {
      System.out.println("- "+args[i]);
    }
  }
  doDelete();
}

protected void takeDown() {
  System.out.println("Bye...");
}
```

4. Creación de Agentes
Ejemplo: Agente Básico con Parámetros

AgBasicoParams.java

Incluimos la destrucción del agente en el agente básico con parámetros

```
package es.upm.ejemplo;
import jade.core.*;
public class AgBasicoParams extends Agent
{
		protected void setup()
		{
			Object [] listaparametros = getArguments();
			if ((listaparametros == null) || (listaparametros.length < 1))
			{
			  System.out.println("No se han introducido parametros");
			 }
			 else
			{
			   System.out.println("Agente JADE con Parametros: Soy el agente " + getLocalName());
			   int i;
			   for (i=0;i<listaparametros.length;i++)
					System.out.println("Parametro " + i + " es: " + (String) listaparametros[i]);
			}
			doDelete();
		}
		protected void takeDown()
		{
			System.out.println("Bye.....");
		}
}
```

4. Creación de Agentes
Ejemplo: Agente Básico

AgBasico.java

Podemos crear agentes desde código, por ejemplo un nuevo agente AgBasicoParams creado desde AgBasico.

```
package es.upm.ejemplo;
import jade.core.Agent;
import jade.wrapper.AgentContainer;
import jade.wrapper.AgentController;

public class AgBasico extends Agent
{
		protected void setup()
		{
				System.out.println("Primer Agente JADE");
				System.out.println("AID: " + this.getAID());

				System.out.println("Entrando en espera");
				this.doWait(10000);
				System.out.println("Saliendo de espera, entrando en suspendido");
				this.doSuspend();
				System.out.println("Saliendo de suspendido");

				AgentContainer container=(AgentContainer) getContainerController();
				Object[] params=new Object[1];
				params[0]=“nuevo_parametro";

				try{
						AgentController agnt=container.createNewAgent("nuevoAgente", "es.upm.ejemplo.AgBasicoParams", params);
						agnt.start();
				}
				catch(Exception e){e.printStackTrace();}
		}
}
```

Index

4. Creación de Agentes
Definición de comportamientos

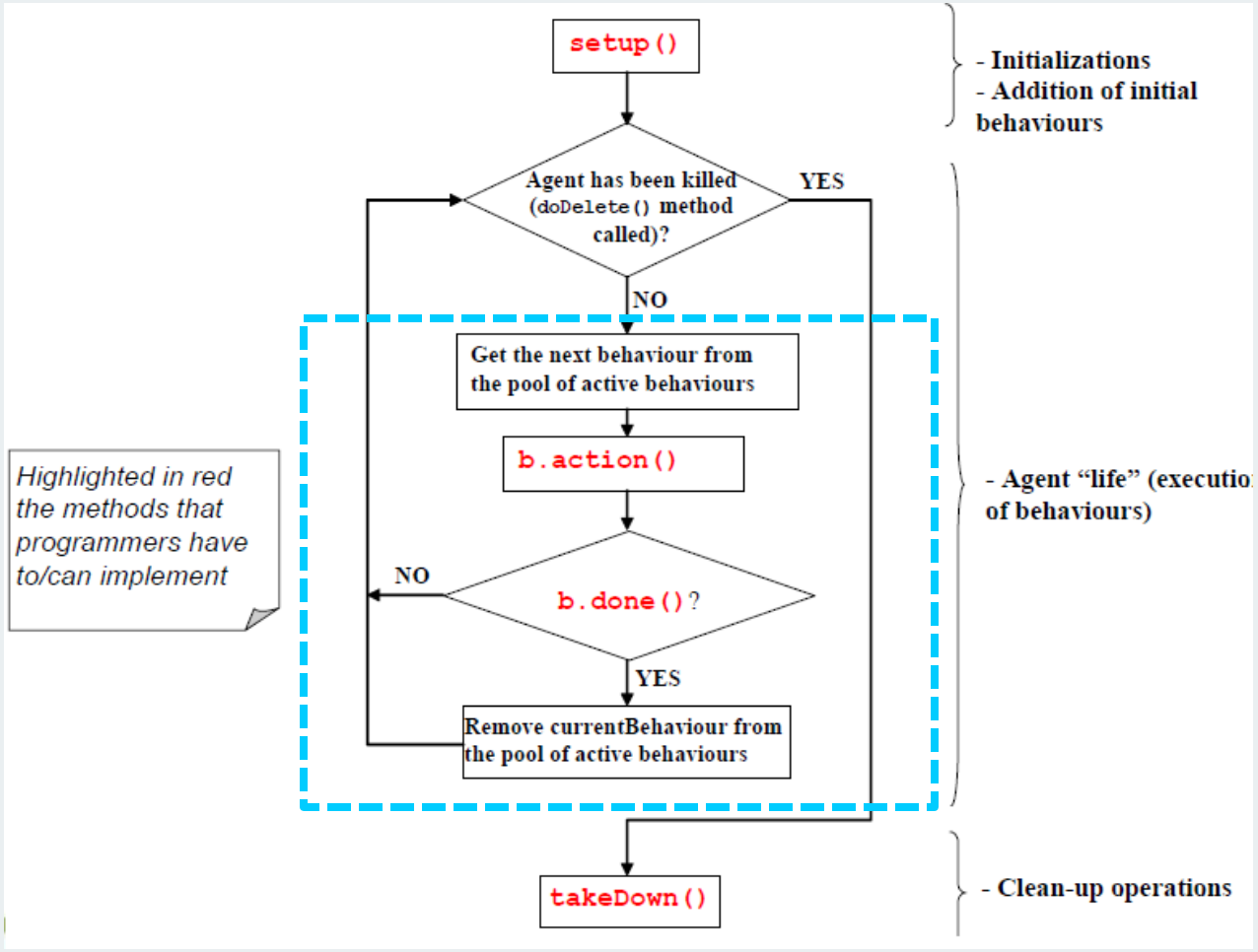
- Los agentes JADE ejecutan comportamientos.
- Los comportamientos se ejecutan en un único Java Thread.
- El comportamiento define las acciones que se desencadenan cuando se produce un determinado evento.
- El comportamiento del agente se define en el método setup() y se añade a la lista de comportamientos mediante el método addBehaviour().

```
60          addBehaviour(new CyclicBehaviour(this){
61              private static final long serialVersionUID = 1L;
62
63              public void action()
64              {
65                  ACLMessage msg=blockingReceive(MessageTemplate.and(
    MessageTemplate.MatchPerformative(ACLMessage.REQUEST), MessageTemplate.
    MatchOntology("ontologia")));
66                  try
67                  {
68                      System.out.println("Mensaje: "+ (String)msg.
    getContentObject());
69                  }
70                  catch (UnreadableException e)
71                  {
72                      // TODO Auto-generated catch block
73                      e.printStackTrace();
74                  }
75              }
76          });
```

Ejemplo de definición del comportamiento en el que sólo se imprimen por pantalla los mensajes que le van llegando al agente, **no se incluye el método done**

4. Creación de Agentes
La Clase Agent

- **Ciclo de Vida de un agente**

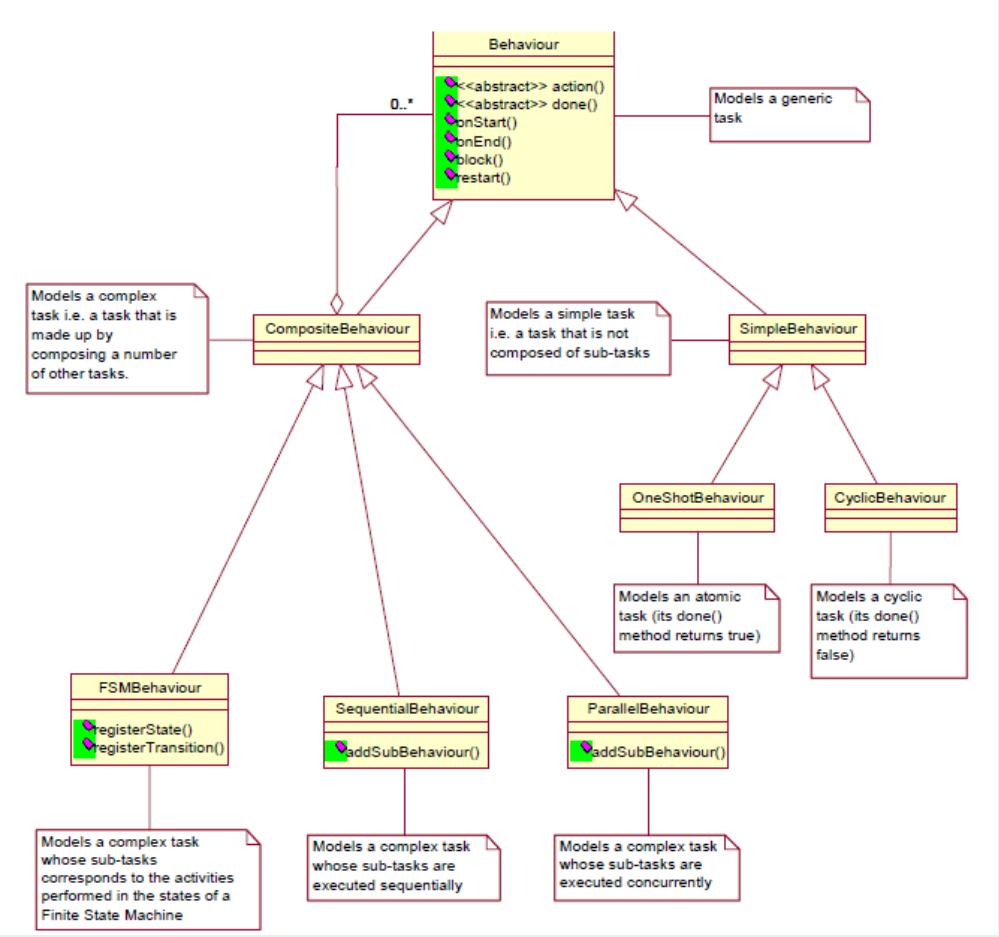


4. Creación de Agentes
Definición de comportamientos

- Los comportamientos se definen a partir de la clase abstracta Behaviour.
- La clase Behaviour contiene los métodos abstractos
  - action(): se ejecuta cuando la acción tiene lugar.
  - done(): se ejecuta al finalizar el comportamiento.
- Además posee los metodos onStart() y onEnd() que el usuario puede redefinir. onStart() se ejecuta una única vez antes de la primera ejecución del método Action(). onEnd() se ejecuta un única vez después de que el método done() devuelve true.
- Otros métodos son:
  - block() para bloquear el comportamiento hasta que se produzca un evento.
  - restart() para reiniciar el comportamiento del agente bloqueado.
- Cuando un agente no ejecuta comportamientos está bloqueado. Se puede desbloquear de una de las siguientes maneras:
  - Se recibe un mensaje
  - Se produce el timeout asociado a block
  - Se llama a restart

4. Creación de Agentes
Definición de comportamientos

- Diagrama de clases



4. Creación de Agentes
Definición de comportamientos

- Implementaciones de Behaviour
  - SimpleBehaviour: se ejecuta una vez el comportamiento. Se ejecuta sin interrupciones. Atómico. Hay dos tipos:
    - OneShotBehaviour: se ejecuta una vez el comportamiento. El método *done()* devuelve *true*. Atómico.
    - CyclicBehaviour: se ejecuta iterativamente el comportamiento. Su método *action()* ejecuta la misma operación cada vez que se invoca. El método *done()* devuelve *false*, nunca finaliza. Atómico.
  - CompositeBehaviour: permite incorporar varios comportamientos. Tienen un estado y ejecutan el método *action()* de diferentes formas dependiendo del estado. Finaliza cuando se cumple cierta condición.
    - SequentialBehaviour: ejecuta secuencialmente los comportamientos.
    - ParallelBehaviour: ejecuta paralelamente los comportamientos.
    - FSMBehaviour: se ejecutan según una máquina de estados finitos que define el usuario.

4. Creación de Agentes
Definición de comportamientos

- Implementaciones de Behaviour
  - WakerBehaviour: se ejecuta solo una vez el comportamiento tras un timeout. El método *action()* ya está definido. El usuario modifica el método *onWake()*.
  - TickerBehaviour: se ejecuta periódicamente. El método *action()* ya está definido. El usuario modifica el método *onTick()*.
  - SenderBehaviour: Acción de envío de un mensaje *ACL*.
  - ReceiverBehaviour: Acción de recepción de un mensaje *ACL*.

4. Creación de Agentes
Definición de comportamientos

- Si se quiere que el comportamiento se lance en un hilo separado hay que usar la clase *ThreadedBehaviourFactory*.
- También se pueden hacer hilos de java normales para reproducir este comportamiento.

Para eliminar, interrumpir el comportamiento o cuando el agente termina hay que actuar directamente sobre el hilo: *thread (interrupt)*.

```
import jade.core.*;
import jade.core.behaviours.*;

public class ThreadedAgent extends Agent
{
        private ThreadedBehaviourFactory tbf = new ThreadedBehaviourFactory;

        protected void setup()
        {
                Behaviour b = new OneShotBehaviour(this)
                {
                        Public void action()
                        {
                                //Establecer acciones a realizar
                        }
                };

                //Ejecución del comportamiento en un hilo dedicado
                //El comportamiento se encapsula dentro de un hilo
                addBehaviour(tbf.wrap(b));
        }
}
```

4. Creación de Agentes
Definición de comportamientos

Establecer un comportamiento:

- Los comportamientos de los agentes se registran en el método *setup()* de la clase *Agent*.
- Los comportamientos son variados y heredan de la clase abstracta *Behaviour*.
- Hay que implementar el método abstracto *action()* que es el que se ejecuta durante la ejecución del comportamiento.

4. Creación de Agentes
Definición de comportamientos

Ejemplo. Agente con comportamiento simple:

- Agente que contiene un comportamiento *SimpleBehaviour* en el que cuenta hasta 10 y va imprimiendo por pantalla el número correspondiente a la cuenta.

- Podemos comprobar lo que ocurre cuando cambiamos *return* a *False* dentro del método *done()*.

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Simple

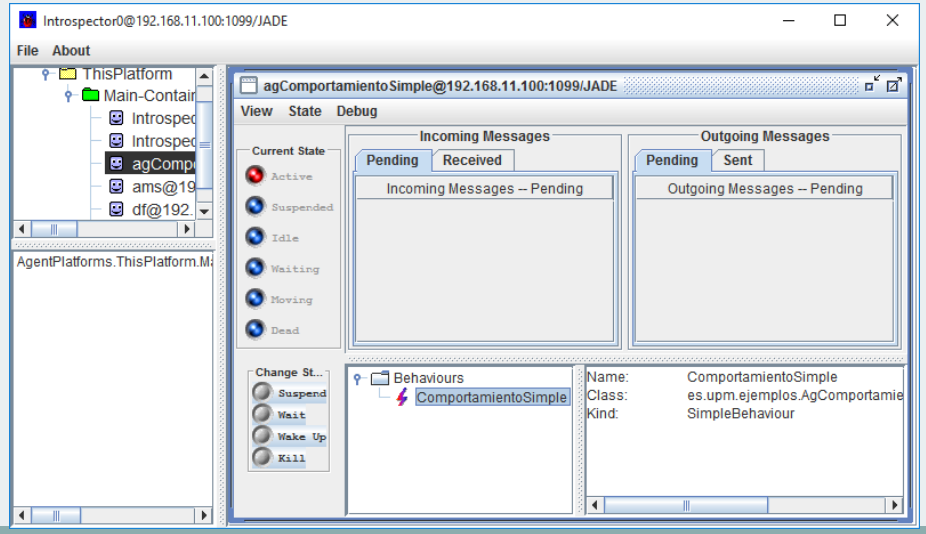
```
package es.upm.ejemplo;

import jade.core.Agent;
import jade.core.behaviours.*;

public class AgComportamientoSimple extends Agent
{
		class ComportamientoSimple extends SimpleBehaviour
		{
			public void action()
			{
				for(int i=0;i<10;i++)
					System.out.println("Ejecuto tarea " + i);
			}
			public boolean done()
			{
				return true;
				//return false;
			}
		}
		protected void setup()
		 {
			System.out.println("Primer Agente JADE con Comportamiento Simple");
			ComportamientoSimple cs= new ComportamientoSimple();
			addBehaviour(cs);
		}
}
```

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Simple

Resultado:

```
Agent container Main-Container@192.168.11.100 is ready.
---------------------------------------------
Primer Agente JADE con Comportamiento Simple
Ejecuto tarea 0
Ejecuto tarea 1
Ejecuto tarea 2
Ejecuto tarea 3
Ejecuto tarea 4
Ejecuto tarea 5
Ejecuto tarea 6
Ejecuto tarea 7
Ejecuto tarea 8
Ejecuto tarea 9
```

Si cambiamos *return* por *False*, lo convertimos en un comportamiento cíclico:



4. Creación de Agentes
Definición de comportamientos

Ejemplo. Agente con 2 comportamientos simples, uno de ellos definido como una clase externa:

- Agente que contiene dos comportamientos *SimpleBehaviour* en el que cuenta hasta 10 y va imprimiendo por pantalla el número correspondiente a la cuenta.
- Uno de los comportamientos se define como una clase externa.
- Podemos comprobar lo que ocurre cuando cambiamos *return* a *False* dentro del método *done()*.

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Simple

```
package es.upm.ejemplo;

import jade.core.behaviours.SimpleBehaviour;

public class ComportamientoSimple2 extends SimpleBehaviour {

		@Override
		public void action() {
				// TODO Auto-generated method stub
				for(int i=0;i<10;i++)
				System.out.println("Soy cs2, Ejecuto tarea " + i);
		}

		@Override
		public boolean done() {
				// TODO Auto-generated method stub
				return true;
				//return false;
}

}
```

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Simple

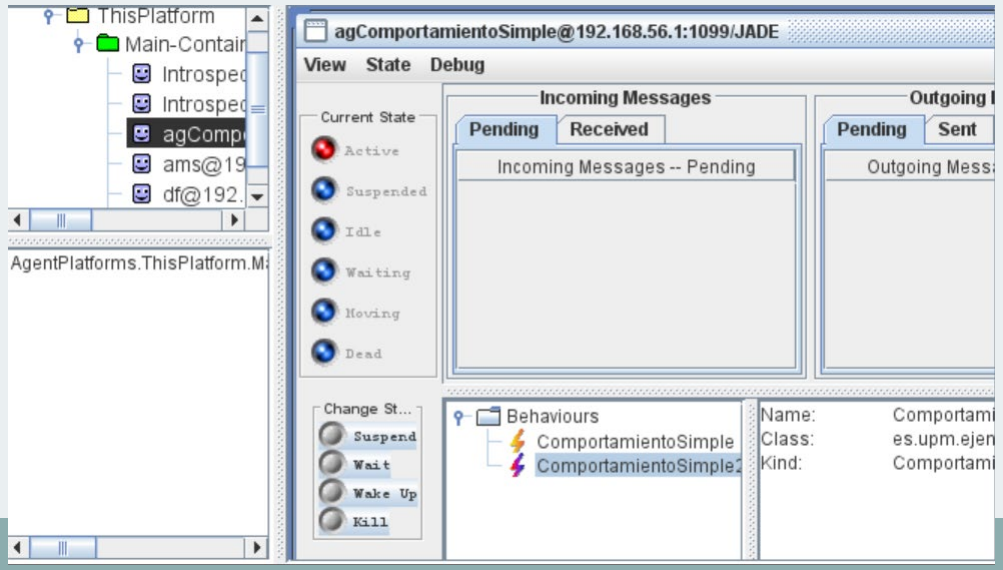
```
package es.upm.ejemplo;

import jade.core.Agent;
import jade.core.behaviours.*;

public class AgComportamientoSimple extends Agent
{
		class ComportamientoSimple extends SimpleBehaviour
		{
			public void action()
			{
				for(int i=0;i<10;i++)
					System.out.println("Ejecuto tarea " + i);
			}
			public boolean done()
			{
				return true;
				//return false;
			}
		}
		protected void setup()
		 {
			System.out.println("Primer Agente JADE con Comportamiento Simple");
			ComportamientoSimple cs= new ComportamientoSimple();
			addBehaviour(cs);
			ComportamientoSimple2 cs2 = new ComportamientoSimple2();
			addBehaviour(cs2);
		}
}
```

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Simple

Resultado:

```
Ejecuto tarea 7
Ejecuto tarea 8
Ejecuto tarea 9
Soy cs2, Ejecuto tarea 0
Soy cs2, Ejecuto tarea 1
Soy cs2, Ejecuto tarea 2
Soy cs2, Ejecuto tarea 3
Soy cs2, Ejecuto tarea 4
Soy cs2, Ejecuto tarea 5
Soy cs2, Ejecuto tarea 6
Soy cs2, Ejecuto tarea 7
```

Si cambiamos *return* por *False*, lo convertimos en dos comportamientos cíclicos:



4. Creación de Agentes
Definición de comportamientos

Ejemplo. Agente con comportamiento cíclico:

- Agente que contiene un comportamiento *CyclicBehaviour* en el que se cuenta de uno en uno y va imprimiendo por pantalla el número correspondiente a la cuenta.

- El **comportamiento se puede crear dentro de la clase Agente o bien en un fichero individual**.

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Cíclico

```
package es.upm.ejemplo;

import jade.core.Agent;
import jade.core.behaviours.*;

public class AgComportamientoCiclico extends Agent
{
		class ComportamientoCiclico extends CyclicBehaviour
		{
			int limite=0;
			public void action()
			{
				limite++;
				System.out.println("Ejecuto tarea " + limite);
			}
		}
		protected void setup()
		{
			System.out.println("Primer Agente JADE con Comportamiento Ciclico");
			ComportamientoCiclico cs1= new ComportamientoCiclico();
			addBehaviour(cs1);
			System.out.println("Despues de añadir el comportamiento Ciclico");
		}
}
```

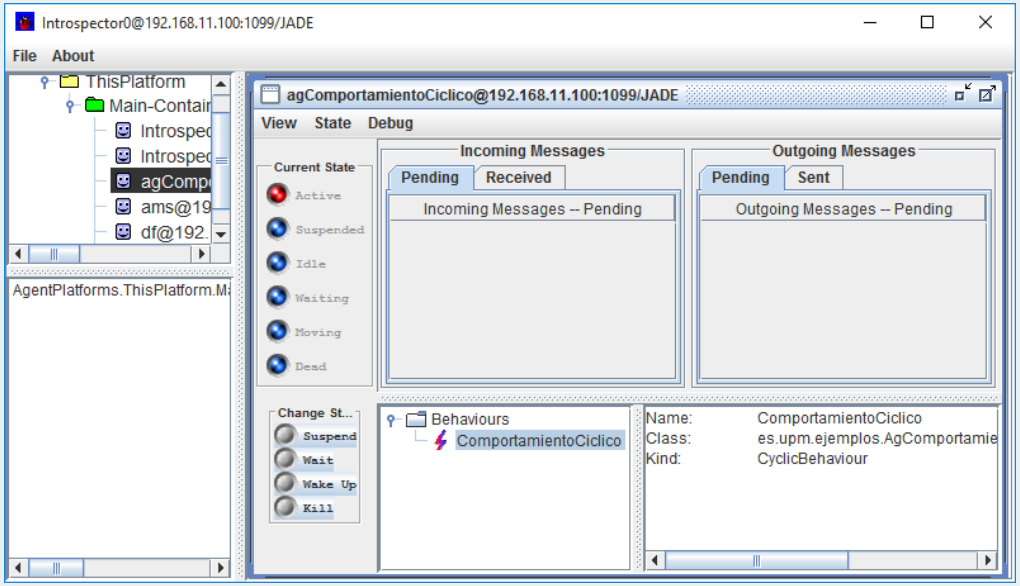
4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Cíclico

Resultado:

Problems @ Javadoc Declaration Console

Agente_Ejemplo [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (

```
Ejecuto tarea 34666
Ejecuto tarea 34667
Ejecuto tarea 34668
Ejecuto tarea 34669
Ejecuto tarea 34670
Ejecuto tarea 34671
Ejecuto tarea 34672
Ejecuto tarea 34673
Ejecuto tarea 34674
Ejecuto tarea 34675
Ejecuto tarea 34676
Ejecuto tarea 34677
```



4. Creación de Agentes
Definición de comportamientos

```
public void setup()
{
    cyclicBehaviour=new CyclicBehaviour(this){
        private static final long serialVersionUID = 1L;

        //método abstracto que hay que implementar se ejecuta ciclicamente
        public void action()
        {
            block();
        }
    };

    addBehaviour(cyclicBehaviour);
}
```

- En este caso se ha creado una clase anónima en java y se ha definido el método abstracto *action()*. De igual modo se podría haber creado una clase que herede de *CyclicBehaviour* y haber definido ahí el método *action()*.

- Dentro del método *action()* se suele incluir todo aquello relacionado con la recepción de mensajes. En ese caso el procedimiento habitual es el de sustituir block por la llamada para recibir mensajes (se verá posteriormente).

4. Creación de Agentes
Definición de comportamientos

Ejemplo. Agente con 2 comportamientos cíclicos:

- Agente que contiene 2 comportamientos *CyclicBehaviour* en los que se cuenta de uno en uno y va imprimiendo por pantalla el número correspondiente a la cuenta.
- El comportamiento 1 cuenta infinitamente.
- El comportamiento 2 cuenta hasta 5000 y cuando llega a 5000 elimina el comportamiento 1.

4. Creación de Agentes
Definición de comportamientos
Agente con 2 Comportamientos Cíclicos

```
package es.upm.ejemplo;

import jade.core.Agent;
import jade.core.behaviours.*;

public class AgComportamientoCiclico2 extends Agent
{
		ComportamientoCiclico2 cs2;
		protected void setup()
		{
			System.out.println("Primer Agente JADE con 2 Comportamientos");
			ComportamientoCiclico1 cs1= new ComportamientoCiclico1();
			cs2= new ComportamientoCiclico2();
			addBehaviour(cs1);
			addBehaviour(cs2);
			System.out.println("Despues de añadir los comportamientos");
		}


//CONTINUA EN LA SIGUIENTE TRANSPARENCIA
```

4. Creación de Agentes
Definición de comportamientos
Agente con 2 Comportamientos Cíclicos

```
        class ComportamientoCiclico1 extends CyclicBehaviour {
                int limite=0;
                public void action() {
                        limite++;
                        System.out.println("Ejecuto tarea C1Lim" + limite);
                        if (limite>5000)
                                removeBehaviour(cs2);
                }
        }

        class ComportamientoCiclico2 extends CyclicBehaviour {
                 int limite=0;
                public void action() {
                        limite++;
                        System.out.println("Ejecuto tarea C2Lim" + limite);
                }
        }
}
```

4. Creación de Agentes
Definición de comportamientos
Agente con 2 Comportamientos Cíclicos

Resultado:

```
Ejecuto tarea C2Lim932
Ejecuto tarea C1Lim933
Ejecuto tarea C2Lim933
Ejecuto tarea C1Lim934
Ejecuto tarea C2Lim934
Ejecuto tarea C1Lim935
Ejecuto tarea C2Lim935
Ejecuto tarea C1Lim936
```

4. Creación de Agentes
Definición de comportamientos

Ejemplo. Agente con 3 comportamientos cíclicos:

- Podemos añadir un tercer comportamiento cs3 que inicialmente esté bloqueado
- Cuando el comportamiento cs1 llegue a su límite, además de eliminar el comportamiento cs2, desbloqueará el comportameniento cs3.

4. Creación de Agentes
Definición de comportamientos
Agente con 3 Comportamientos Cíclicos

```
package es.upm.ejemplo;

import jade.core.Agent;
import jade.core.behaviours.*;

public class AgComportamientoCiclico2 extends Agent {

		ComportamientoCiclico2 cs2;
		ComportamientoCiclico3 cs3;

		protected void setup()
		{
				System.out.println("Primer Agente JADE con 2 Comportamientos");
				ComportamientoCiclico1 cs1= new ComportamientoCiclico1();
				cs2= new ComportamientoCiclico2();
				cs3= new ComportamientoCiclico3();
				addBehaviour(cs1);
				addBehaviour(cs2);
				addBehaviour(cs3);
				cs3.block();
				System.out.println("Despues de añadir los comportamientos");
		}
```

//CONTINUA EN LA SIGUIENTE TRANSPARENCIA

4. Creación de Agentes
Definición de comportamientos
Agente con 3 Comportamientos Cíclicos

```
class ComportamientoCiclico1 extends CyclicBehaviour {
        int limite=0;
        public void action() {
                limite++;
                System.out.println("Ejecuto tarea C1Lim" + limite);
                if (limite>500000) {
                        removeBehaviour(cs2);
                        cs3.restart();
                }
        }
}

class ComportamientoCiclico2 extends CyclicBehaviour {
         int limite=0;
        public void action() {
                limite++;
                System.out.println("Ejecuto tarea C2Lim" + limite);
        }
}

class ComportamientoCiclico3 extends CyclicBehaviour {
         int limite=0;
        public void action() {
                limite++;
                System.out.println("Ejecuto tarea C3Lim" + limite);
        }
}
}
```

4. Creación de Agentes
Definición de comportamientos

Ejemplo. Agente con comportamiento secuencial:

- Agente con 1 comportamiento *SequentialBehaviour*. Dicho comportamiento está formado por 3 sub-comportamientos *OneShotBehaviour* en los que se imprime por pantalla el texto "Subcomportamiento X", siendo X el identificador del sub-comportamiento.

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Secuencial

```
package es.upm.ejemplo;

import jade.core.Agent;
import jade.core.behaviours.*;

public class AgComportamientoSecuencial extends Agent
{
		protected void setup()
		{
			SequentialBehaviour sequentialBehaviour = new SequentialBehaviour(this);

			sequentialBehaviour.addSubBehaviour(new OneShotBehaviour(this){
				public void action(){
					System.out.println(“Subcomportamiento 1");
				}
			});
			 sequentialBehaviour.addSubBehaviour(new OneShotBehaviour(this){
				public void action(){
					System.out.println("Subcomportamiento 2");
				}
			});
			 sequentialBehaviour.addSubBehaviour(new OneShotBehaviour(this){
				public void action(){
					System.out.println("Subcomportamiento 3");
				}
			});

			addBehaviour(sequentialBehaviour);
		}
}
```

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamientos Secuencial

Resultado:

```
INFORMACIÓN: Service jade.core.event.Notification initialized
oct 28, 2015 3:36:35 PM jade.mtp.http.HTTPServer <init>
INFORMACIÓN: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
oct 28, 2015 3:36:35 PM jade.core.messaging.MessagingService boot
INFORMACIÓN: MTP addresses:
http://192.168.11.100:7778/acc
oct 28, 2015 3:36:36 PM jade.core.AgentContainerImpl joinPlatform
INFORMACIÓN: ----------------------------------------
Agent container Main-Container@192.168.11.100 is ready.
------------------------------------------------
Subcomportamiento 1
Subcomportamiento 2
Subcomportamiento 3
```

4. Creación de Agentes
Definición de comportamientos

Ejemplo. Agente con comportamiento paralelo:

- Agente que 1 comportamiento *ParallelBehaviour*. Dicho comportamiento está formado por 2 comportamientos *SequentialBehaviour*. Cada uno de los 2 comportamientos secuenciales está formado por 3 sub-comportamientos *OneShotBehaviour* en los que se imprime por pantalla el texto "Subcomportamiento X_Y", siendo X el identificador del comportamiento secuencial e Y el identificador del sub-comportamiento *OneShotBehaviour*.

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Paralelo

```
package es.upm.ejemplo;

import jade.core.Agent;
import jade.core.behaviours.*;

public class AgComportamientoParalelo extends Agent
{
		protected void setup()
		{
			SequentialBehaviour sequentialBehaviour1 = new SequentialBehaviour(this);

		 sequentialBehaviour1.addSubBehaviour(new OneShotBehaviour(this){
			public void action(){
				System.out.println(“Subcomportamiento 1_1");
			}
		});
		 sequentialBehaviour1.addSubBehaviour(new OneShotBehaviour(this){
			public void action(){
				System.out.println("Subcomportamiento 1_2");
			}
		});
		 sequentialBehaviour1.addSubBehaviour(new OneShotBehaviour(this){
			public void action(){
				System.out.println("Subcomportamiento 1_3");
			}
		});
//CONTINUA EN LA SIGUIENTE TRANSPARENCIA
```

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamiento Paralelo

```
                SequentialBehaviour sequentialBehaviour2 = new SequentialBehaviour(this);

        sequentialBehaviour2.addSubBehaviour(new OneShotBehaviour(this){
                public void action(){
                        System.out.println("Subcomportamiento 2_1");
                }
        });
        sequentialBehaviour2.addSubBehaviour(new OneShotBehaviour(this){
                public void action(){
                        System.out.println("Subcomportamiento 2_2");
                }
        });
        sequentialBehaviour2.addSubBehaviour(new OneShotBehaviour(this){
                public void action(){
                        System.out.println("Subcomportamiento 2_3");
                }
        });

        ParallelBehaviour parallelBehaviour = new ParallelBehaviour(this,ParallelBehaviour.WHEN_ALL);
        parallelBehaviour.addSubBehaviour(sequentialBehaviour1);
        parallelBehaviour.addSubBehaviour(sequentialBehaviour2);

        addBehaviour(parallelBehaviour);

        }
}
```

4. Creación de Agentes
Definición de comportamientos
Agente con Comportamientos Paralelo

Resultado:

```
INFORMACIÓN: ----------------------------------------
Agent container Main-Container@192.168.11.100 is ready.
--------------------------------------------
Subcomportamiento 1_1
Subcomportamiento 2_1
Subcomportamiento 1_2
Subcomportamiento 2_2
Subcomportamiento 1_3
Subcomportamiento 2_3
```

Como se puede observar el paralelismo es simulado ya que se ejecuta un único thread para el agente. Si quisiésemos tener paralelismo real tendríamos que lanzar cada comportamiento en un hilo independiente.