Desarrollo de sistemas Multiagentes con Jade

Jose Angel Bañares
UNIVERSITY OF ZARAGOZA

A partir del libro de Bellifemine, Caire yGreenwood
Transparencias de los autores

Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

JADE

- JADE es un middleware que facilita el desarrollo de
  - **Aplicaciones Multi-agente Peer-to-Peer.**
  - **El ciclo de vida de los agentes y la movilidad.**
  - **Servicios de páginas Blancas y Amarillas.**
  - **Transporte y análisis de mensajes Peer-to-peer**
  - **Seguridad**
  - **Planificación de las tareas del agente.**
  - **Herramientas gráficas para la monitorización, logs, depuración.**

JADE

- Desarrollado en Java
  - Ejecutable sobre todas las MVJ desde J2EE a J2ME MIDP 1.0
- Distribuido en Open Source bajo licencia LGPL
  - Descargable en http://jade.tilab.com
- El proyecto JADE es una iniciativa de TILAB
- **JADE cumple las especificones FIPA**

JADE

- ***Algunas funcionalidades de Interés reseñadas***
  - **Integración con JESS (Java Expert System Shell)**
    - **Permite razonar con los mensajes en JESS**
    - **Permite a un programa JESS controlar el envío y recepción de mensajes y/o crear/destruir comportamientos JADE**
  - **JADE herramientas de Internet**
    - **Integrado con servlets, applets, JSP**
  - **Características Avanzadas**
    - **Seguridad distribuida, tolerancia a fallos, soporte para la replica de agentes y servicios, persistencia, …**
    - **JADE y .NET**
    - **JADE, Protègè, XML, RDF and OWL**

      Nota: La documentación incluye un tutorial para casi todos estos aspectos.

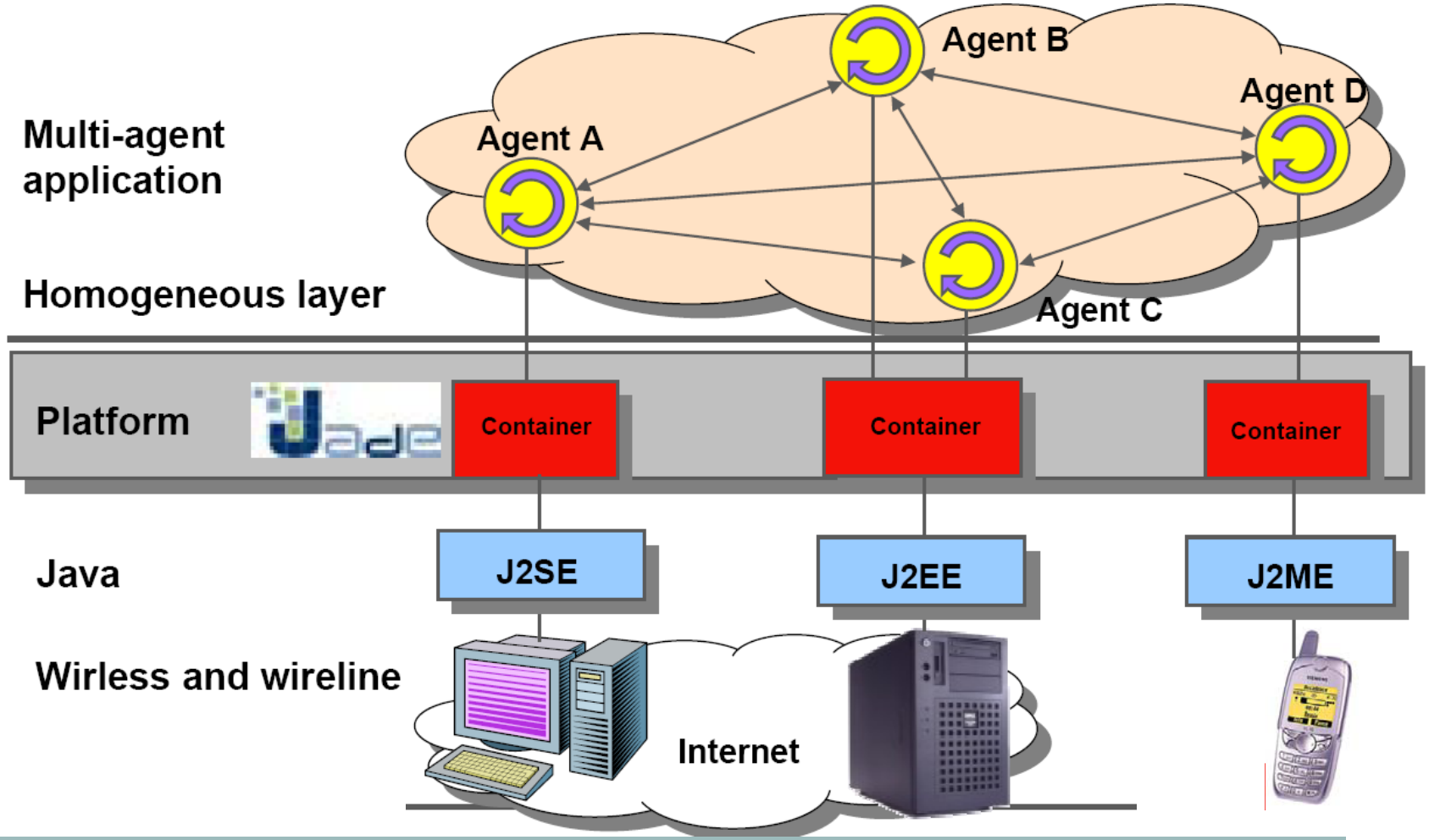
Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Modelo Arquitectural

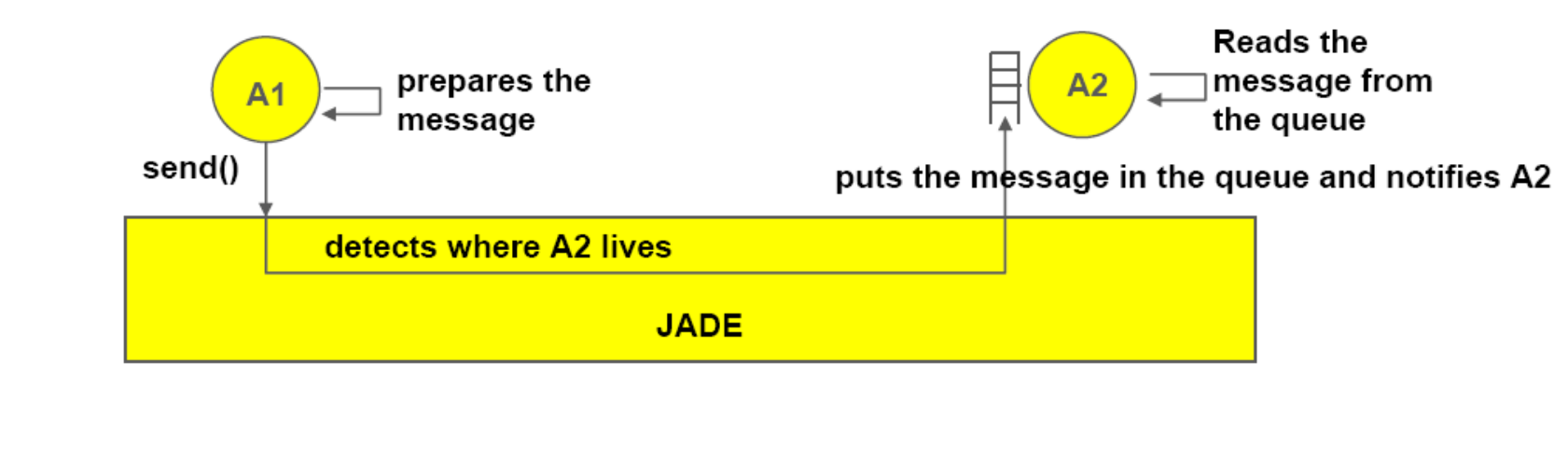
- **Una aplicación basada en JADE consta de "componentes activas" denominadas Agentes.**
- **Cada agente tiene un nombre (identificador) único.**
- **Cada agente es un participante (peer) capaz de comunicar bidireccionalmente con otros agentes.**
- **Cada agente "vive" en un contenedor.**

El modelo Arquitectural



El modelo de comunicación

- **Basado en paso de mensajes asícrono.**
  - **Cada agente tiene una cola de mensajes (mailbox) donde pueden enviarse los mensajes. Cuando un mensaje se coloca en el mailbox, el agente es notificado. Es responsabilidad del agente cuando leer el mensaje y como reaccionar.**



Modelo de referencia FIPA de una plataforma de agentes

- Sistema de gestión de agentes. AMS, **Agent Management System**, supervisa la plataforma. Es el punto de contacto para todos los agentes que necesitan interaccionar para acceder a las **páginas blancas** y **gestionar su ciclo de vida**.
- Páginas Amarillas. DF, **Directory Facility** , utilizada por cualquier agente que busca o registra servicios disponibles.
- Canal de comunicación de agentes. ACC, **Agent Communication Channel**, suministra protocolos de transporte de mensajes: MTS (**Message Transport Protocols**). Es el responsable de enviar y recibir los mensajes sobre una plataforma de agentes, AP (Agent Platform).

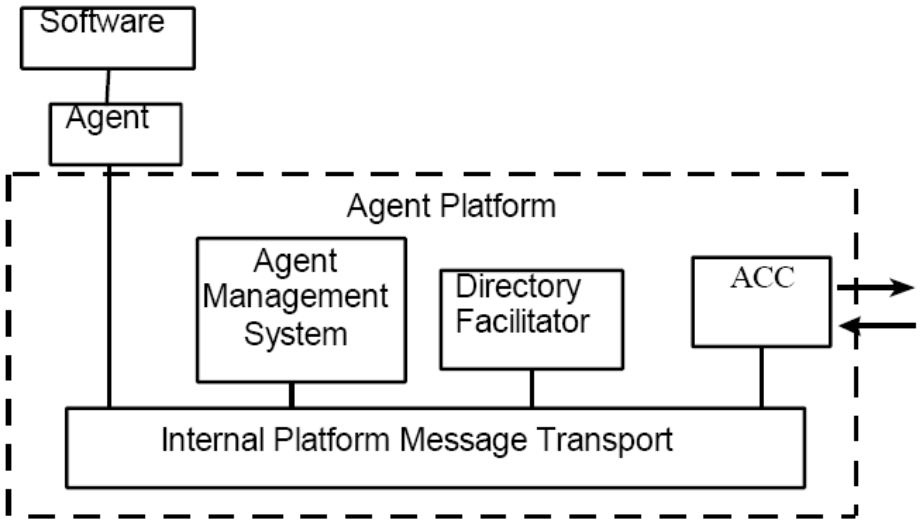


Figure 1 – FIPA reference model of an Agent Platform

Arquitectura Jade

La arquitectura software se basa en la coexistencia de varias MVJ y la comunicación es soportada por **Java RMI** (Remote Method Invocation) **entre diferentes MVJ** y por **eventos dentro de una MV**. Cada MV es un contenedor de agentes completamente ejecutable.

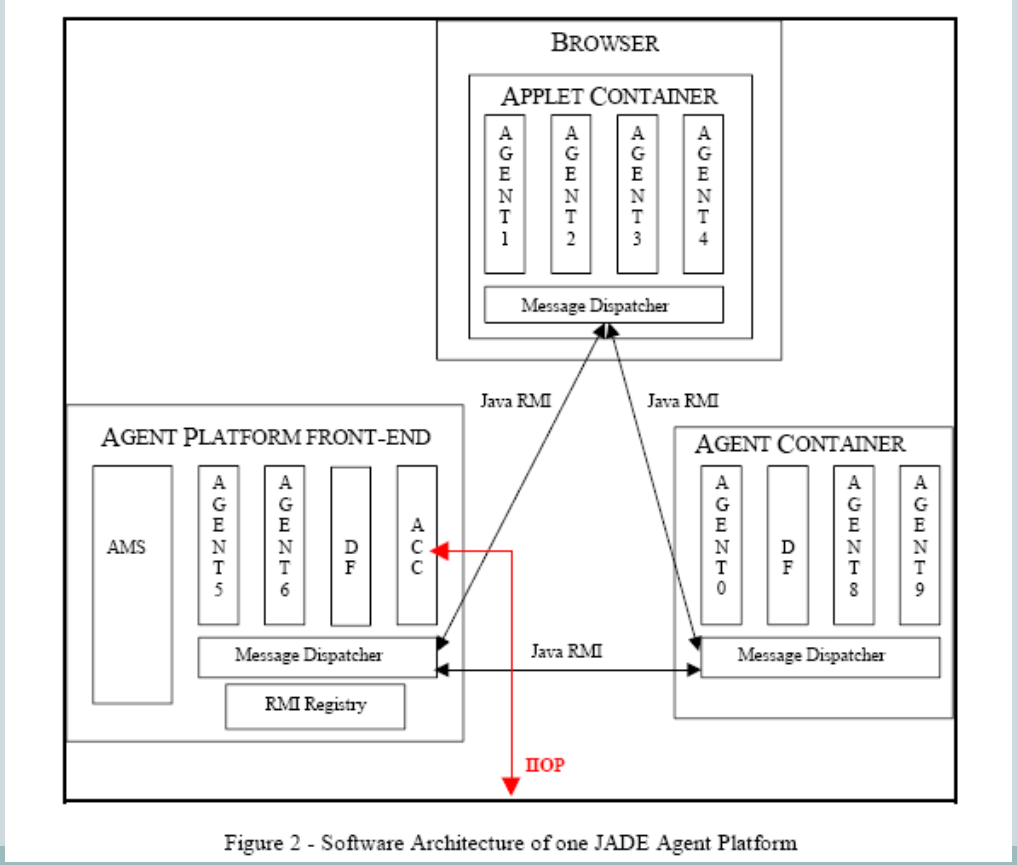


Figure 2 - Software Architecture of one JADE Agent Platform

Outline

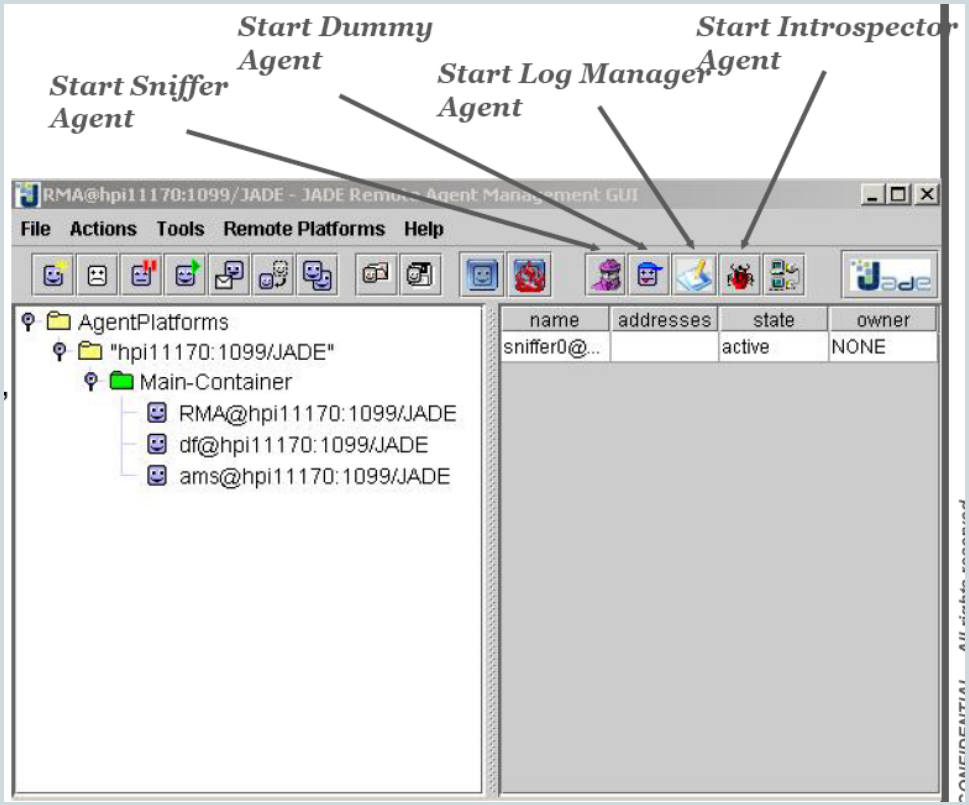
- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Las principales herramientas gráficas Jade

- **Soporte para la gestion, control, monitorización y depurado de la plataforma multi-agente.**
  - RMA (Remote Monitoring Agent)
  - DummyAgent
  - SnifferAgent
  - IntrospectorAgent
  - Log Manager Agent
  - DF (Directory Facilitator) GUI

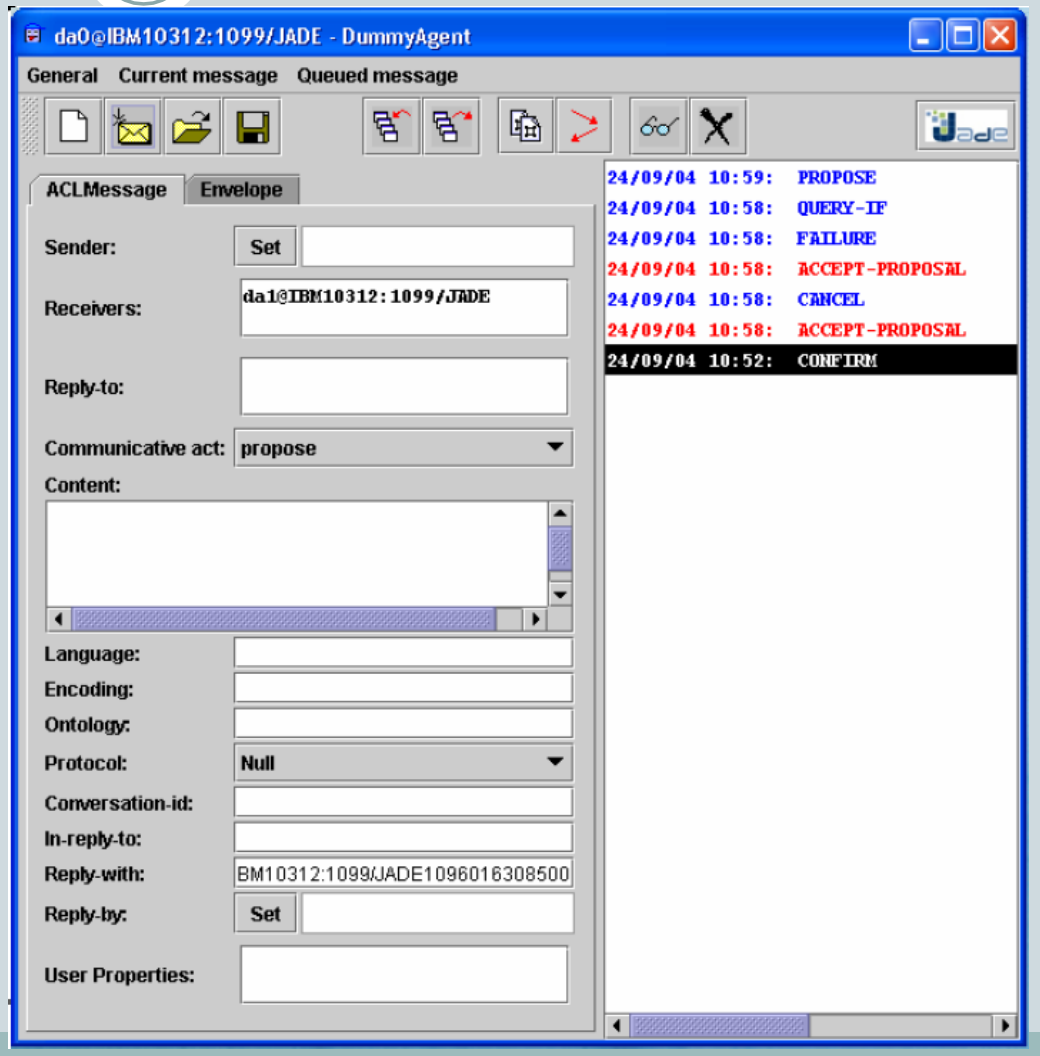
Remote Management Agent

- **Funcionalidades:**
  - Monitorizar y controlar la plataforma y todos sus contenedores remotos.
  - Gestión remota del ciclo de vida de los agentes (creción, suspender, reactivar, matar, migrar, clonar)
  - Componer y enviar mensajes “ah-doc” a un agente.
  - Lanzar otras herramientas gráficas.
  - Monitorizar (lectura de operaciones)



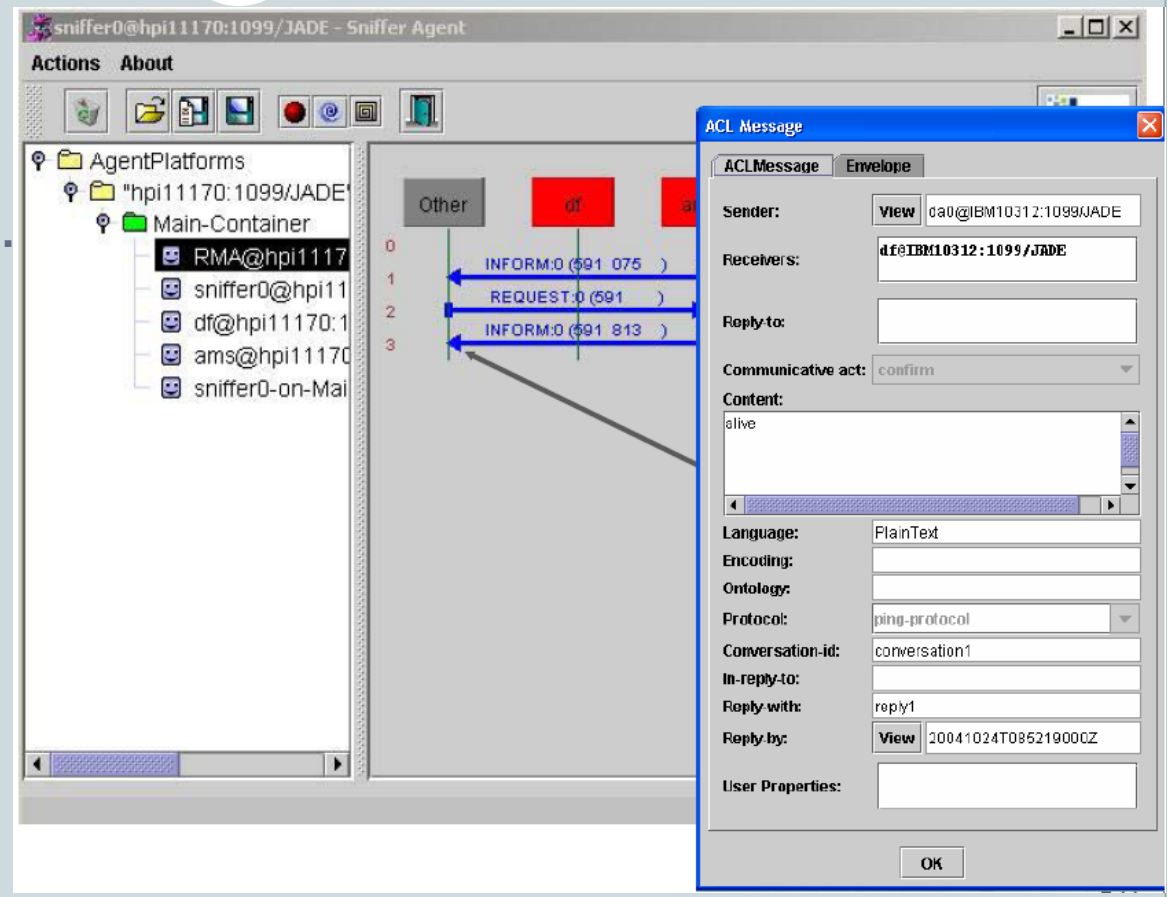
Dummy Agent

- **Funcionalidades**
  - Compone y envía mensajes “ad-hoc”
  - load/save la cola de mensajes de/a un fichero.



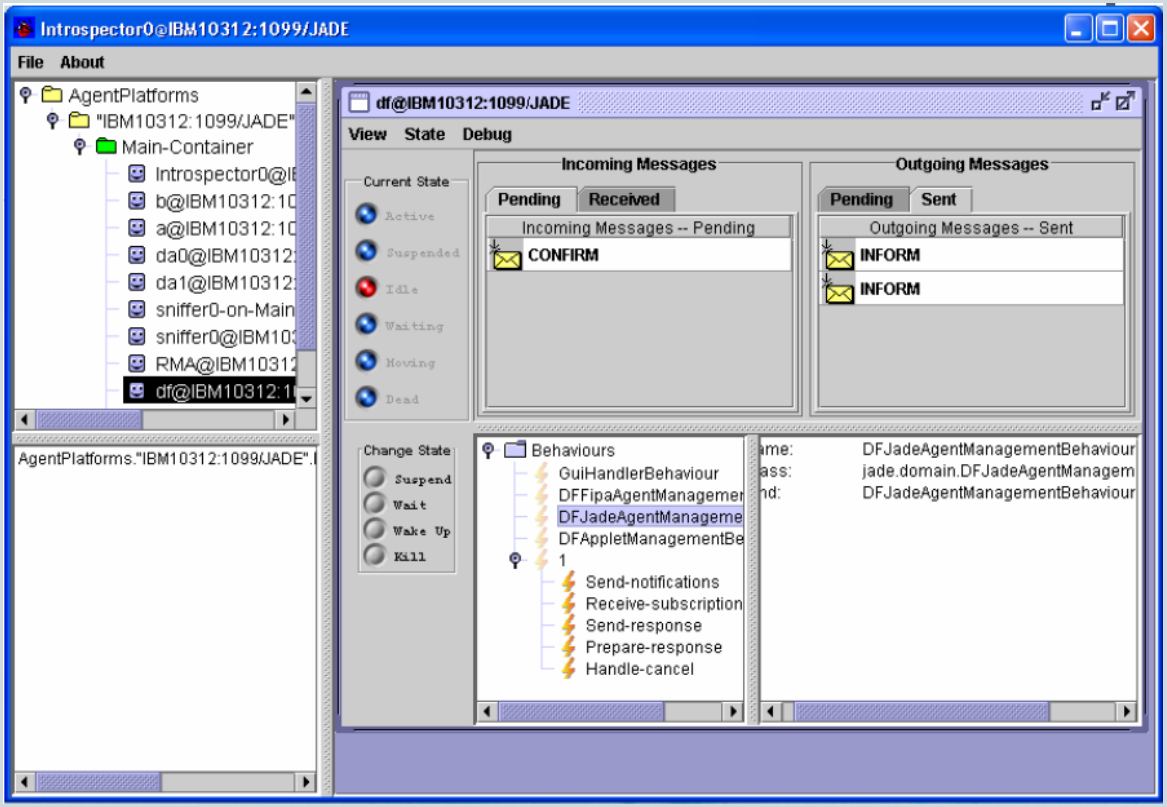
Sniffer Agent

- **Funcionalidades:**
  - Muestra el flujo de interacciones entre los agentes seleccionados.
  - Muestra el contendido de cada mensajes,
  - Save/load el flujo de/a un fichero.



Introspector Agent

- **Funcionalidad: Monitorización del estado interno del agente**
  - Mensajes recibidos/ enviados/ pendientes
  - Comportamientos planificados (activo, bloqueado) y subcomportamientos.
- **Depurado**
  - step-by-step
  - slowly
  - break points



Log Manager Agent

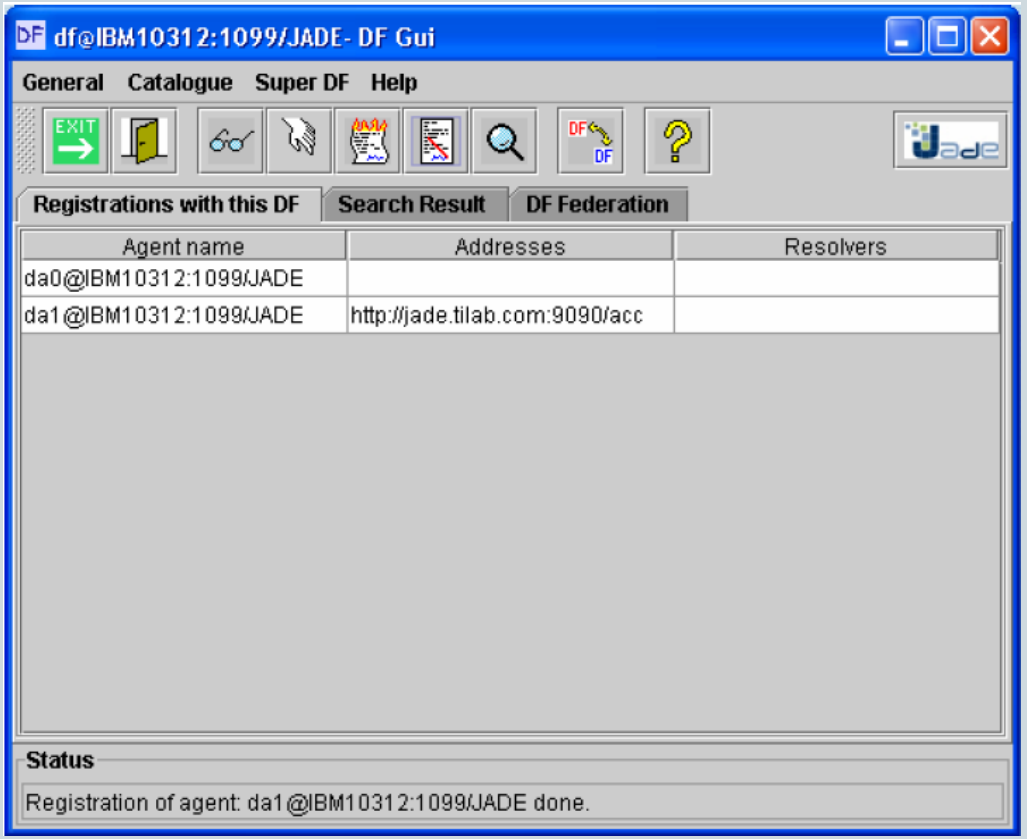
- **GUI para modificar el logging de la plataforma**
- **Se basa en java.util.logging y permite:**
  - **browse Logger objects en su contenedor)**
  - **Modificar el logging level –añadir nuevos logging handlers (e.g. files)**

da0@hpi11170:1099/JADE - LogManagerAgent

Logger Name	Set Level	Handlers	Set log file
jade.content.lang.sl.SL0Ont...	INFO	java.util.logging.ConsoleHandler	
jade.content.lang.sl.SL1Ont...	INFO	java.util.logging.ConsoleHandler	
jade.content.lang.sl.SL2Ont...	INFO	java.util.logging.ConsoleHandler	
jade.content.lang.sl.SLOntol...	INFO	java.util.logging.ConsoleHandler	
jade.content.onto.BasicOntol...	SEVERE	java.util.logging.ConsoleHandler, java.util.logging.FileHandler	myLog.txt
jade.content.onto.Ontology	INFO	java.util.logging.ConsoleHandler	
jade.content.onto.Serializabl...	INFO	java.util.logging.ConsoleHandler	
jade.content.schema.AgentA...	INFO	java.util.logging.ConsoleHandler	
jade.content.schema.Aggreg...	INFO	java.util.logging.ConsoleHandler	
jade.content.schema.Conce...	INFO	java.util.logging.ConsoleHandler	
jade.content.schema.Conte...	INFO	java.util.logging.ConsoleHandler	
jade.content.schema.Conte...	FINER	java.util.logging.ConsoleHandler	
jade.content.schema.IRESc...	INFO	java.util.logging.ConsoleHandler	
jade.content.schema.Object...	INFO	java.util.logging.ConsoleHandler	
jade.content.schema.Predic...	SEVERE	java.util.logging.ConsoleHandler	
jade.content.schema.Primiti...	WARNING	java.util.logging.ConsoleHandler, java.util.logging.FileHandler	log.txt
jade.content.schema.TermS...	INFO	java.util.logging.ConsoleHandler	
jade.content.schema.Variabl...	CONFIG	java.util.logging.ConsoleHandler	
jade.core.AgentContainerImpl	FINE	java.util.logging.ConsoleHandler	
jade.domain.DFGUIManage...	FINER	java.util.logging.ConsoleHandler	
jade.domain.DFMemKB	FINEST	java.util.logging.ConsoleHandler	
jade.domain.FIPAAgentMan...	ALL	java.util.logging.ConsoleHandler	
jade.domain.FIPAAgentMan...		java.util.logging.ConsoleHandler	
jade.domain.JADEAgentMan...	INFO	java.util.logging.ConsoleHandler	
jade.domain.ams	INFO	java.util.logging.ConsoleHandler	
jade.domain.df	INFO	java.util.logging.ConsoleHandler	

DF GUI

- **GUI del servicio de Páginas amarillas, permite:**
  - browse, registro, desregistro, modificación, búsqueda de agentes
  - Federación con otros DF
  - Ejecutar búsquedas federadas.



Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Programación con JADE

- Creación de agentes
- Un agente se crea heredando de la clase jade.core.Agent class y reescribiendo el método **setup**().
  - Cada instancia de agente se idendifica por un AID (jade.core.AID).
  - Un AID se compone de un nombre único más una dirección.
  - Un agente puede recuperar su AID mediante el método getAID() de la clase Agent

```
import jade.core.Agent;

 public class HelloWorldAgent extends Agent
 {
     protected void setup()
     {
      System.out.println("Hello World. ");
      System.out.println("My name local is "+getLocalName());
      System.out.println("“My GUID is "+getAID().getName());

     }
 }
```

Peter

Peter anduril:1099/
JADE

Local name, GUID y Adresses

- Los nombres son <local-name>@<platform-name>
- El nombre completo debe ser globalmente único.
- El nombre de la plataforma por defecto es <main-host>:<main-port>/JADE
- El nombre de la plataforma puede establecerse con la opción –name
- Dentro de una plataforma JADE pueden ser designados **sólo por sus nombres.**
- •Dado el nombre de un agente su AID puede ser creado como
  - AID id = new AID(localname, AID.ISLOCALNAME);
  - AID id = new AID(name, AID.ISGUID);
- Las direcciones incluidas en un AID son las de los MTPs y se utilizan SOLO para comunicar entre agentes de diferentes plataformas FIPA.

Argumentos para un agente

- **Es posible pasar argumentos a un agente al lanzar una plataforma java jade.Boot .... a:myPackage.MyAgent(arg1 arg2)**
- **El agente puede recuperar los argumentos mediante el método de la clase Agente getArguments()**

```
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]);
                        }
        }
}
```

Terminación de un agente

- **Un agente finaliza cuando se invoca su método doDelete().**
  - **Al terminar se invoca el método del agente takeDown() (con objeto de realizar las operaciones de limpieza necesarias).**

```
protected void setup() {
        System.out.println("Hello 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...");
}
```

CLASSPATH & Lanzamiento de Agentes

set JADE_HOME=c:\projects\jade

Set CLASSPATH=%JADE_HOME%\lib\jade.jar;%JADE_HOME%\lib\jadeTools.jar;%JADE_HOME%\lib\http.jar;%JADE_HOME%\lib\iiop.jar;%JADE_HOME%\lib\commons-codec\commons-codec-1.3.jar;%JADE_HOME%\classes;classes

javac –classpath JADE-classes;. HelloWorldAgent.java

Java –classpath JADE-classes **jade.Boot Peter:HelloWorldAgent**

Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Comportamientos en los Agentes

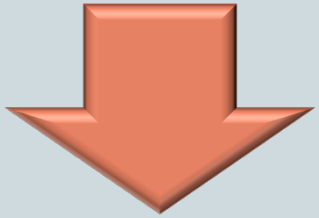
- **Diferentes comportamientos en un agente**
  - Java **multi-thread** o/y
  - **behaviours** JADE que cooperan
    - Un **thread-por-agente** en lugar de un thread-por-tarea/ conversación. **Planificación cooperativa**.
    - Cada **comportamiento debe liberar** el control para permitir el trabajo de otros comportamientos.
    - El **planificador** de JADE ejecuta una **política no expulsiva round-robin** entre todos los comportamientos de la cola de comportamientos.
  - Los comportamientos se pueden componer en una máquina de estados finito

Programación con JADE

- **El trabajo real de los agentes se realiza mendiante comportamientos ("behaviours")**
- **Los Behaviours se crean heredando de la clase jade.core.behaviours.Behaviour**
- **Para que un agente ejecute una tarea es suficiente con crear una instancia de la subclase correspondiente de Behaviour y llamar al método addBehaviour() de la clase Agente.**
  - Cada subclase de Behaviour debe implementar los métodos
    - **public void action(): Lo que hace realmente el behaviour**
    - **boolean done(): Indica si el comportamiento ha finalizado su tarea.**

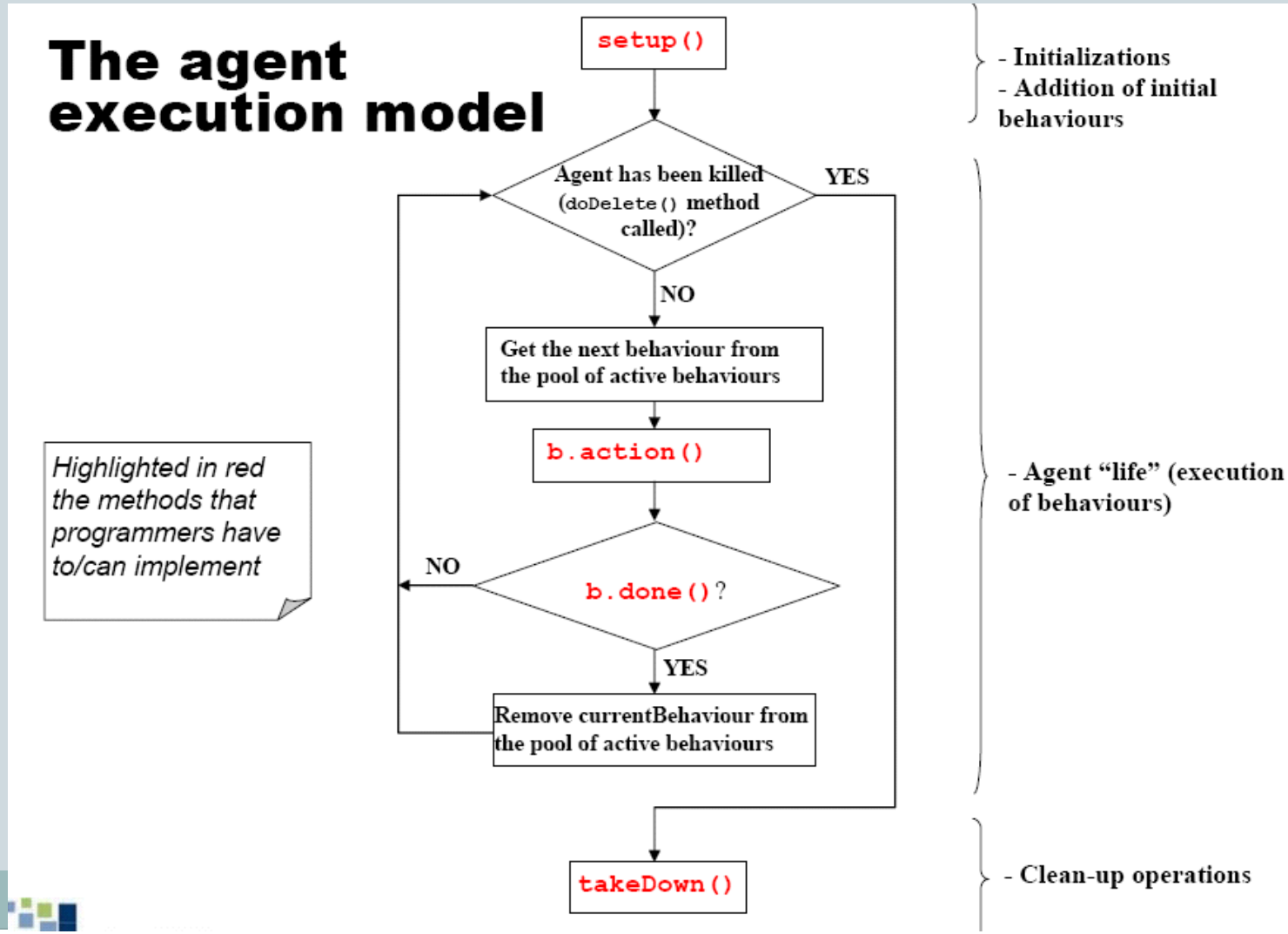
Behaviour Scheduling and Execution

- **Un agente puede ejecutar varios comportamientos en paralelo, sin embargo la planificación de comportamientos no es expulsiva (no es preemptive), sino cooperativa y todo ocurre en un único Java Thread**



El cambio de Behaviours sólo se puede realizar cuando el método action() del behaviour en curso ejecuta returns.

Modelo de ejecución



Tipos de Comportamientos

- SimpleBehaviour
  - Comportamiento simple que se puede ejecutar sin interrupciones. Tiene dos subclases
- **One shot" behaviours**: se ejecuta una sola vez
  - **Finalizan inmediatamente y sus método action() se ejecuta sólo una vez.**
  - **Su método done() devuelve true.**
  - **Clase jade.core.behaviours.OneShotBehaviour**
- **"Cyclic" behaviours.**: se ejecuta de forma ciclica
  - **Nunca finalizan, y sus método action() ejecuta la misma operación cada vez que se invoca.**
  - **Su método done() devuelve false.**
  - **Clase jade.core.behaviours.CyclicBehaviour**

Tipos de Comportamientos

- **"Complex" behaviours.**
  - **Contienen estado y ejecutan su métodod action() de diferente forma dependiendo del estado.**
  - **Finalizan cuando se cumple cierta condición.**

Planificación de operaciones periódicas

- **Dos formas de ejecutar comportamientos periódicamente**
- **WakerBehaviour**
  - **Los métodos action() y done() están preimplementados de forma que el método onWake() (que debe ser implementado en las subclases) se ejecuta después de un timeout.**
  - ***Después de la ejecución el behaviour finaliza*.**

```
public class MyAgent extends Agent
{
    protected void setup()
    {
     System.out.println("Adding waker behaviour. ");
     addBehaviour(new WakerBehaviour(this, 10000)){
                    protected void onWake(){
                    //perform operation X
     }
    });
    }}      ;; Realiza operación X después de 10 segundos
```

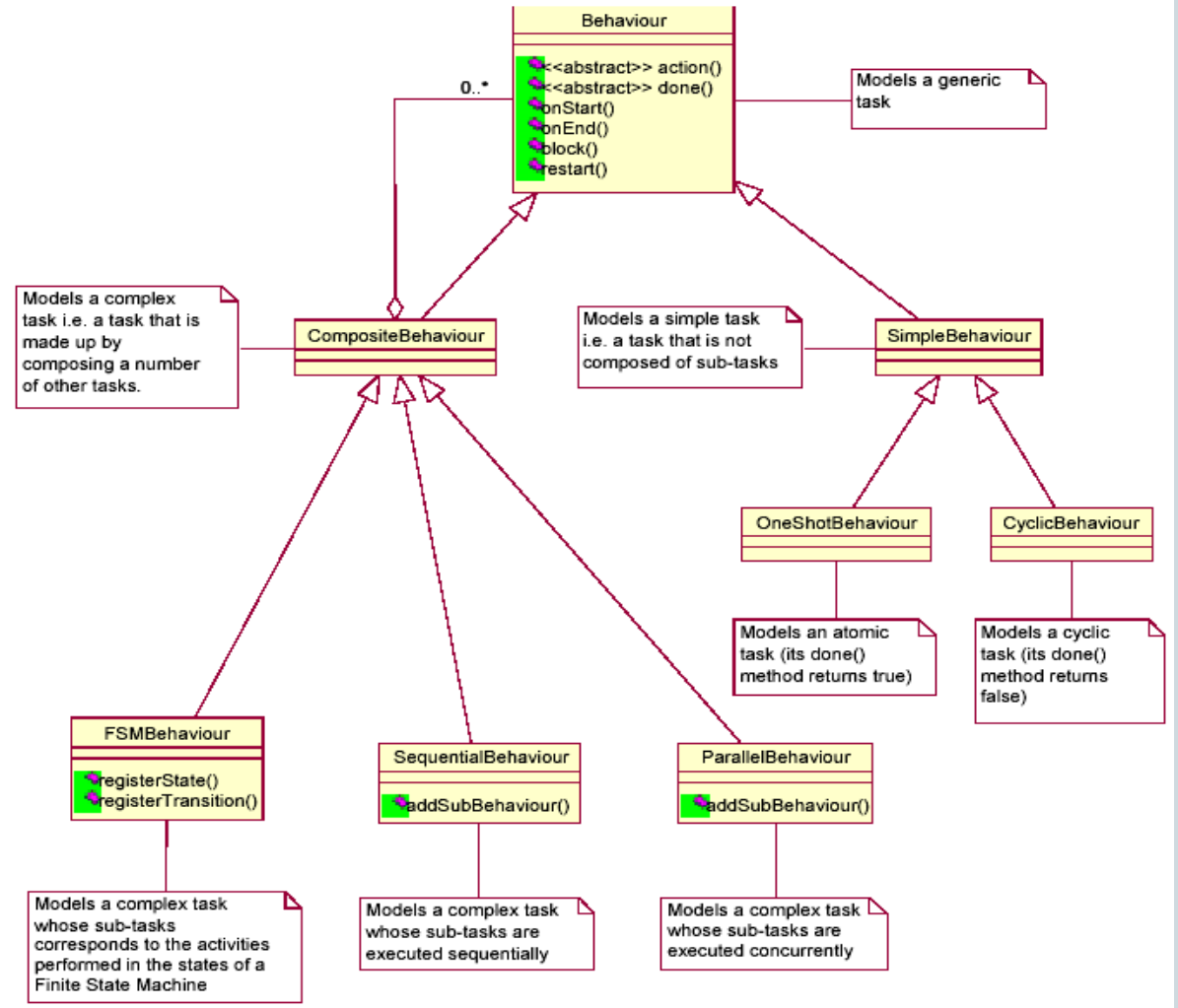
- **TickerBehaviour**
  - Los métodos **action()** y **done()** están preimplementados de forma que el métodos **onTick()** (que debe ser implementado en las subclases) se ejecuta periódicamente.
  - El behaviour *se ejecuta siempre a no ser que se invoque el método* **stop()**.

```
public class MyAgent extends Agent
{
    protected void setup()
    {
     System.out.println("Adding Ticker behaviour. ");
     addBehaviour(new TickerBehaviour(this, 10000)){
                  protected void onWake(){
                  //perform operation Y
     }
    });  }
}    ;; Realiza operación Y cada 10 segundos
```

Mas sobre comportamientos

- **El método onStart() de la clase Behaviour sólo una vez antes de la invocación del método action().**
- **El método onEnd() de la clase Behaviour se invoca sólo una vez después de que el método done() devuelva true.**
  - •Cada comportamiento tiene un puntero al agente que lo ejecuta: La variable protegida MyAgent.
- **El método removeBehaviour() de la clase Agent se puede utilizar para eliminar un behaviour de la cola de comportamientos del agente. El método onEnd() no se llama en éste caso.**
- **Cuando la cola de comportamientos activos está vacia el agente entra en IDLE y su thread pasa a estar dormido.**

Jerarquía de Comportamientos



Ejemplos behaviour

```
SimpleBehaviour hello_behaviour = new SimpleBehaviour(this){
    boolean finished = false;

    public void action(){
        System.out.println("Hello World Behaviour run: Hello World!");
        System.out.println("-----About Me:-----");
        System.out.println("My local name is:"+getLocalName());
        System.out.println("My globally unique name is:"+getName() );
        System.out.println("-----About Here:-----");
        Location l = here();
        System.out.println("I am running in a location called:"+l.getName());
        System.out.println("Which is identified uniquely as:"+l.getID());
        System.out.println("And is contactable at:"+l.getAddress());
        System.out.println("Using the protocol:"+l.getProtocol());
        finished = true;
    }
    public boolean done(){
        return finished;
    }
}
```

Ejemplos behaviour

```
SimpleBehaviour Notas= new SimpleBehaviour(this){
    boolean finished = false;
    int state = 0;
    public void action(){
            switch(state){
              case 0: System.out.println("Do"); break;
              case 1: System.out.println("Re"); break;
              case 2: System.out.println("Mi"); break;
              case 3: System.out.println("Fa"); break;
              case 4: System.out.println("Sol"); break;
              case 5: System.out.println("Do");  finished = true; break;
             }
            state++;
   }
   public boolean done(){
return finished; }
 };
```

Ejemplos. Sequential Behaviours

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

    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("Do");}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("Re");}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println(”Mi");}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("Fa");}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("Sol");}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("La");}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println(”Si");}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println(”Do");}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println(”Re);}});
    s.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println(”Mi");}});
```

Ejemplos: Parallel Behaviours

```
public class ParallelBehaviourAgent extends Agent{
    public void setup(){
    SequentialBehaviour s1 = new SequentialBehaviour(this);
    s1.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("1) This ");}});
    s1.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("1) is");}});
    s1.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("1) the ");}});
    s1.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("1) first");}});
    s1.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("1)
    behaviour");}});

    SequentialBehaviour s2 = new SequentialBehaviour(this);
    s2.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("2) This ");}});
    s2.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("2) is");}});
    s2.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("2) the ");}});
    s2.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("2) second");}});
    s2.addSubBehaviour(new OneShotBehaviour(this){public void action(){System.out.println("2)
    behaviour");}});
    ParallelBehaviour p = new ParallelBehaviour(this,ParallelBehaviour.WHEN_ALL);
    p.addSubBehaviour(s1);
    p.addSubBehaviour(s2);
    addBehaviour(p);
} }
```

Ejemplos

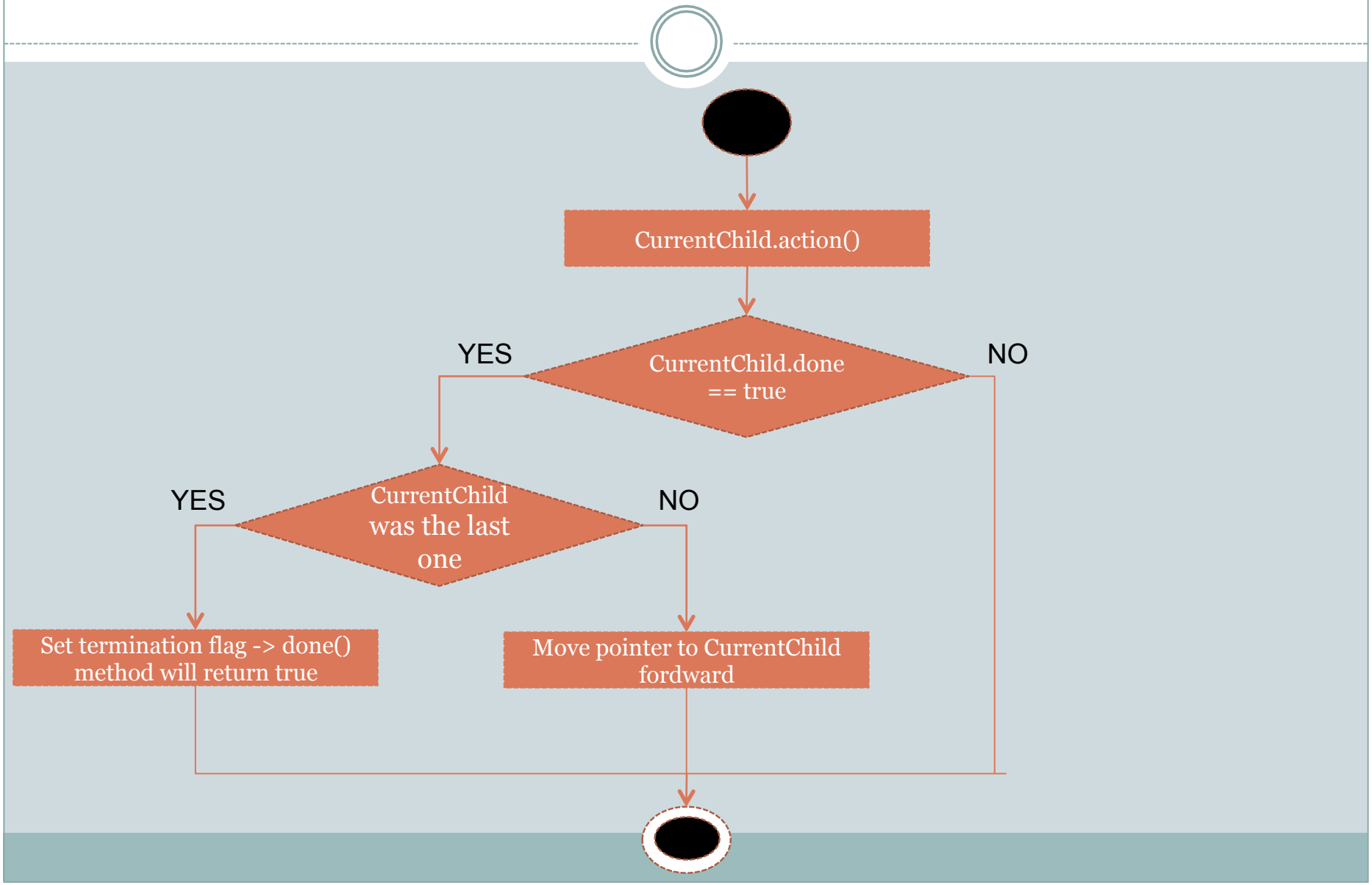
```
1) This
2) This
1) is
2) is
1) the
2) the
1) first
2) second
1) behaviour
2) behaviour
```

Ejemplos

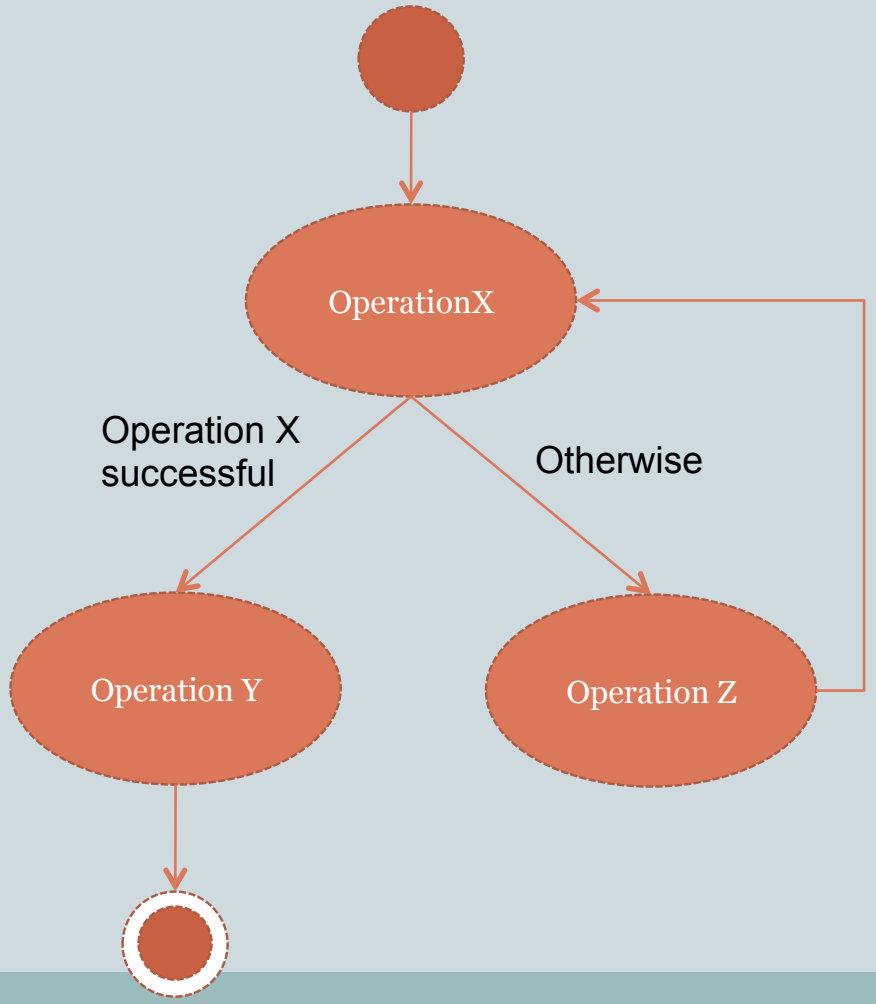
- **FSMBehaviour: Todos los composite behaviours se pueden implementar mediante este comportamiento.**

- El FSMBehaviour se basa en el concepto de Maquina de Estados (Finite State Machine).
- Planifica el comportamiento de acuerdo a los estados de una FMS, cuyos estados corresponden a los comportamientos de los subcomportamientos (hijos).

The FMSBehaviour Class



A simple State Machine



SampleFMS

```
FMSBehaviour sampleFMS = new FMSBehaviour(anAgent);
sampleFMS.registerFirstState(new OneShotBehaviour (anAgent) {
   public void action() {
   //perform operation X
   }
   public int onEnd() {
          return (operation X successful ? 1 : 0);
   }
}, "X");
sampleFMS.registerLastState(new OneShotBehaviour (anAgent) {
   public void action() {
   //perform operation Y
   }
}, "Y");

sampleFMS.registerState(new OneShotBehaviour (anAgent) {
   public void action() {
   //perform operation Z
   }
}, "Z");
sampleFMS.registerTransition("X","Y",1);
sampleFMS.registerTransition("X","Z",0);
sampleFMS.registerDefaultTransition("Z","X", new String[]{"X","Z"});
```

Register sub-behaviours as FMS states

This paramenter indicates a set of FSM states that must be reset

Resumen

- Agente = Thread.
- Cola de comportamientos.
- Comportamiento Re-Entrante.
  - Scheduling (Planificador) Round-Robin non- preemtive : Ejecucion del metodo action en bloque (Multitarea cooperativa).

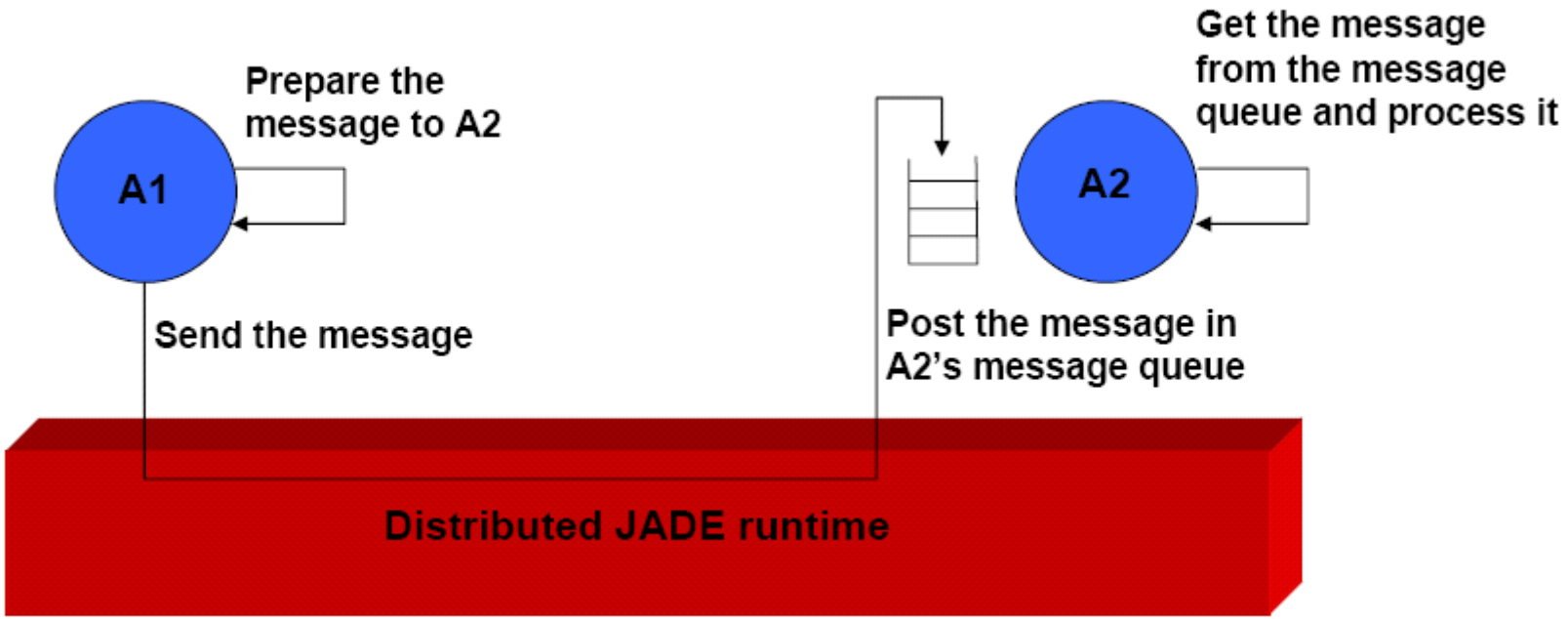
IMPORTANTE: Evitar action() infinitos o de larga duracion

Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

El modelo de comunicación

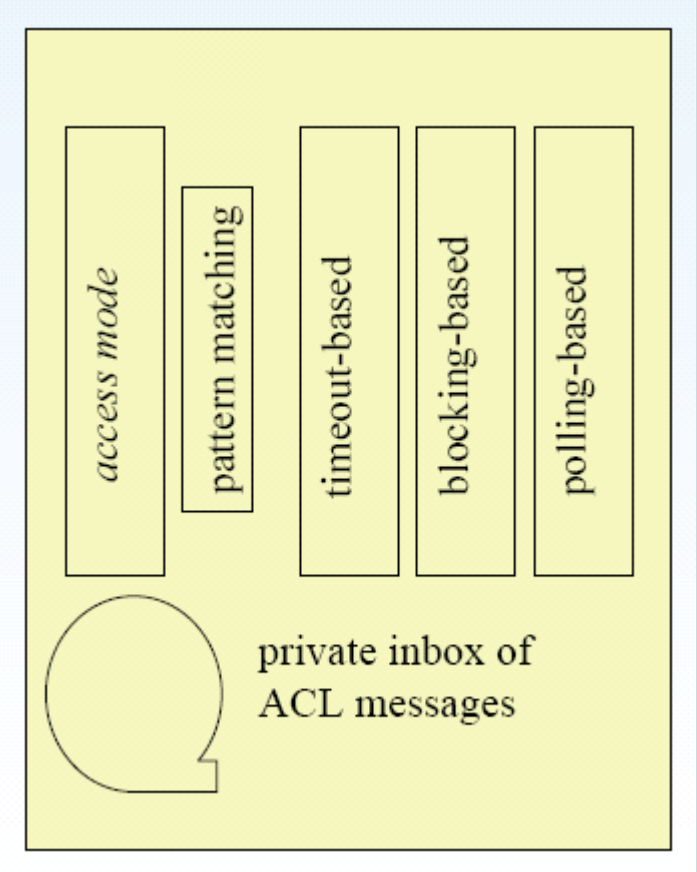
- **Basado en paso de mensajes asíncrono.**
- **Formato definido por el lenguaje ACL de (FIPA)**



El modelo de comunicación

Los agentes son autónomos

- Controlan su traza de ejecución
- Deciden que mensajes y cuando los leen.
  - El mecanismo de transporte completa las colas de mensajes de cada agente
  - No hay callback automáticos



La clase ACLMessage

- **Los mensajes intercambiados por los agentes son instancias de la clase jade.lang.acl.ACLMessage**
  - **Ofrece metodos de acceso y escritura a todos los campos definidos por el lenguaje ACL:**
  - **get/setPerformative();**
  - **get/setSender();**
  - **add/getAllReceiver();**
  - **get/setLanguage();**
  - **get/setOntology();**
  - **get/setContent();**
  - **....**

Envío y recepción de mensajes

- **El envío de mensajes es tan sencillo como crear un objecto ACLMessage y llamar al método send() del Agent class**

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-Forecast-Ontology");
msg.setContent("Today it's raining");
send(msg);
```

Leer mensajes de una cola privada se realiza a través del método receive() de la clase Agent.

```
ACLMessage msg = receive();
if (msg != null) {
// Process the message
}
```

Bloqueo de un comportamiento a la espera de un mensaje

- **Un comportamiento que procesa mensajes de entrada no sabe cuando llegaran mensajes. Esto llevaria a comprobar continuamente la cola de mensajes myAgent.receive().**
  - Esto consumiría mucha CPU.
  - El método block() de la clase Behaviour class saca el comportamiento del agente y lo coloca en estado bloqueado. No es una llamada bloqueante.
  - Cada vez que se recibe un mensaje todos los comportamientos bloqueados se colocan en la la lista de agentes que tienen la oportunidad de leer y procesar mensajes.

```
public void action() {
ACLMessage msg = myAgent.receive();
        if (msg != null) {
        // Process the message
        }
        else {
                block();
        }
}
```

This is the strongly recommended pattern to receive messages within a behaviour

Lectura selectiva de la cola de mensajes

- **El metodo receive() devuelve el primer mensaje de la cola y lo saca de está.**
  - Si hay dos o más comportamientos reciviendo mensajes, un comportamiento le puede quitar a otro el mensaje.
  - Para evitar esto es posible leer mensajes con ciertas características (por ejemplo, el mensaje enviado al agente "Peter") especificando un parámetro jade.lang.acl.MessageTemplate en el método receive().

```
MessageTemplate tpl = MessageTemplate.MatchOntology("Test-Ontology");
public void action() {
ACLMessage msg = myAgent.receive(tpl);
        if (msg != null) {
                // Process the message
        }
        else {
                block();
        }
}
```

Recibir mensajes en el modo Blocking

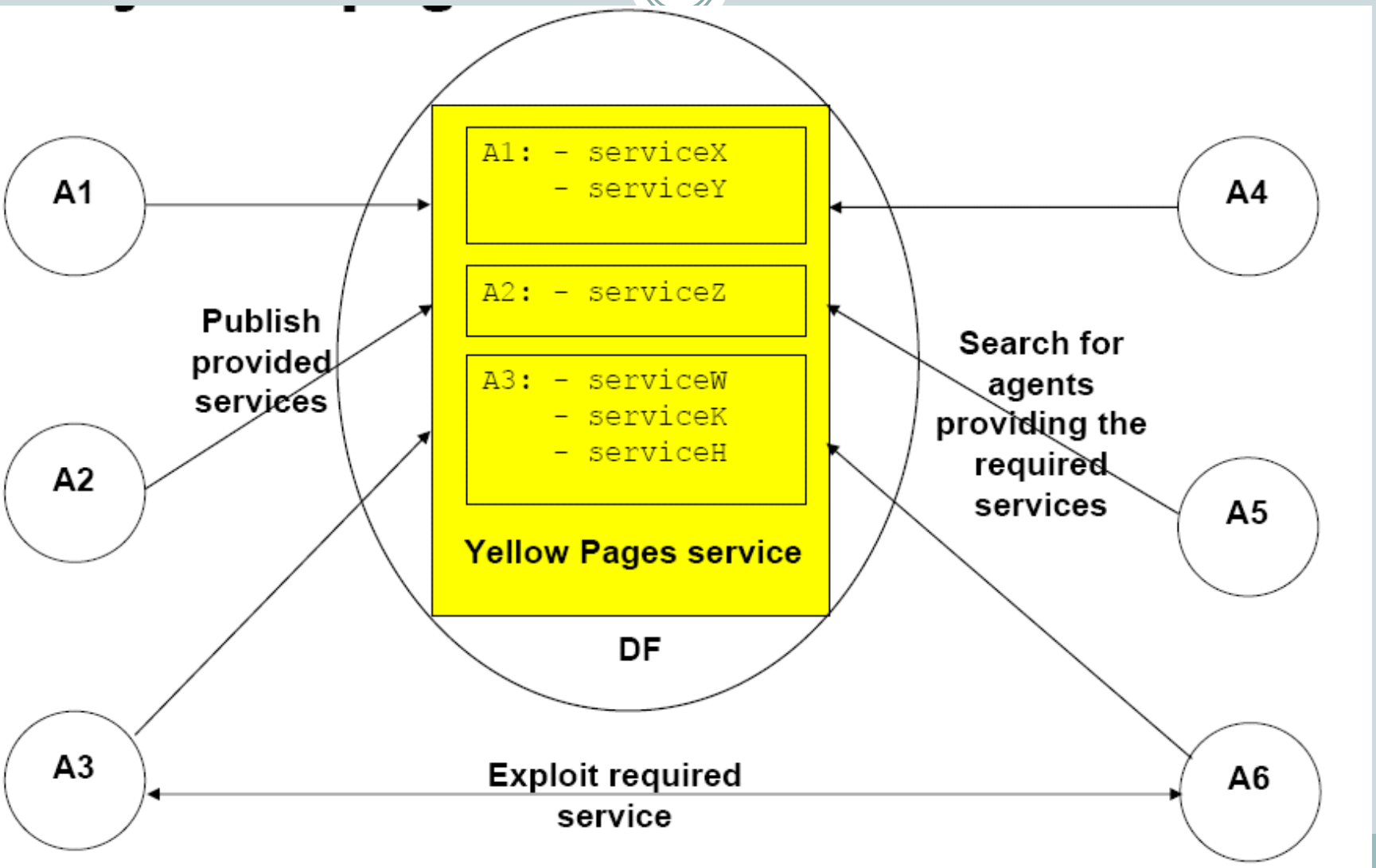
- **La clase agente puede tambien utilizar el método blockingReceive() que devuelve un valor sólo cuando hay un mensaje en la cola.**
  - Hay versiones sobrecargadas que aceptan un MessageTemplate y o un and or a timeout (si el tiempo expira devuelve null).
  - Es peligroso utilizar las llamadas bloqueantes blockingReceive() en un comportamiento. Ningún otro behaviour se ejecutará hasta que finalize blockingReceive() ¡.

-Utilizar receive() + Behaviour.block() para recibir mensajes dentro de comportamiento.
- Utilizar blockingReceive() para recibir mensajes al iniciar el agente o finalizarlo (métodos del agente setup() y takeDown ()).

Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Advanced/Yellow Pages



Interacción con el agente DF

- **El Directory Facility (DF) es un agente y como tal comunica utilizando ACL**
- **El lenguaje y la ontología utilizado por DF es la especificada por FIPA**
  - **Es posible buscar/registrar en un agente DF de una plataforma remota.**
- **La clase jade.domain.DFServiceclass ofrece métodos de la clase que facilitan intercciones con DF**
  - **register(); modify(); deregister(); search();**
- **JADE DF también soporta suscripción**

DF Description Format

- Cuando se registra un agente en el DF debe aportar una descripción (implementada por la clase **jade.domain.FIPAAgentManagement.DFAgentDescription**) que consta de:
- El agent AID
- Una colección de descripciones de servicios (implementadas por la clase **ServiceDescription**) que incluyen:
  - El tipo de servicio (e.g. “Weather forecast”);
  - El nombre(e.g. “Meteo-1”);
  - Los lenguajes, las ontologías, y los protocolos de interacción que se deben conocer para explotar el servicio
  - Una colección de propiedades de servicios especificadas en forma de pares clave-valor
- Cuando un agente busca/suscribe a un DF debe suministrar un **DFAgentDescription** que se utilizará como template.

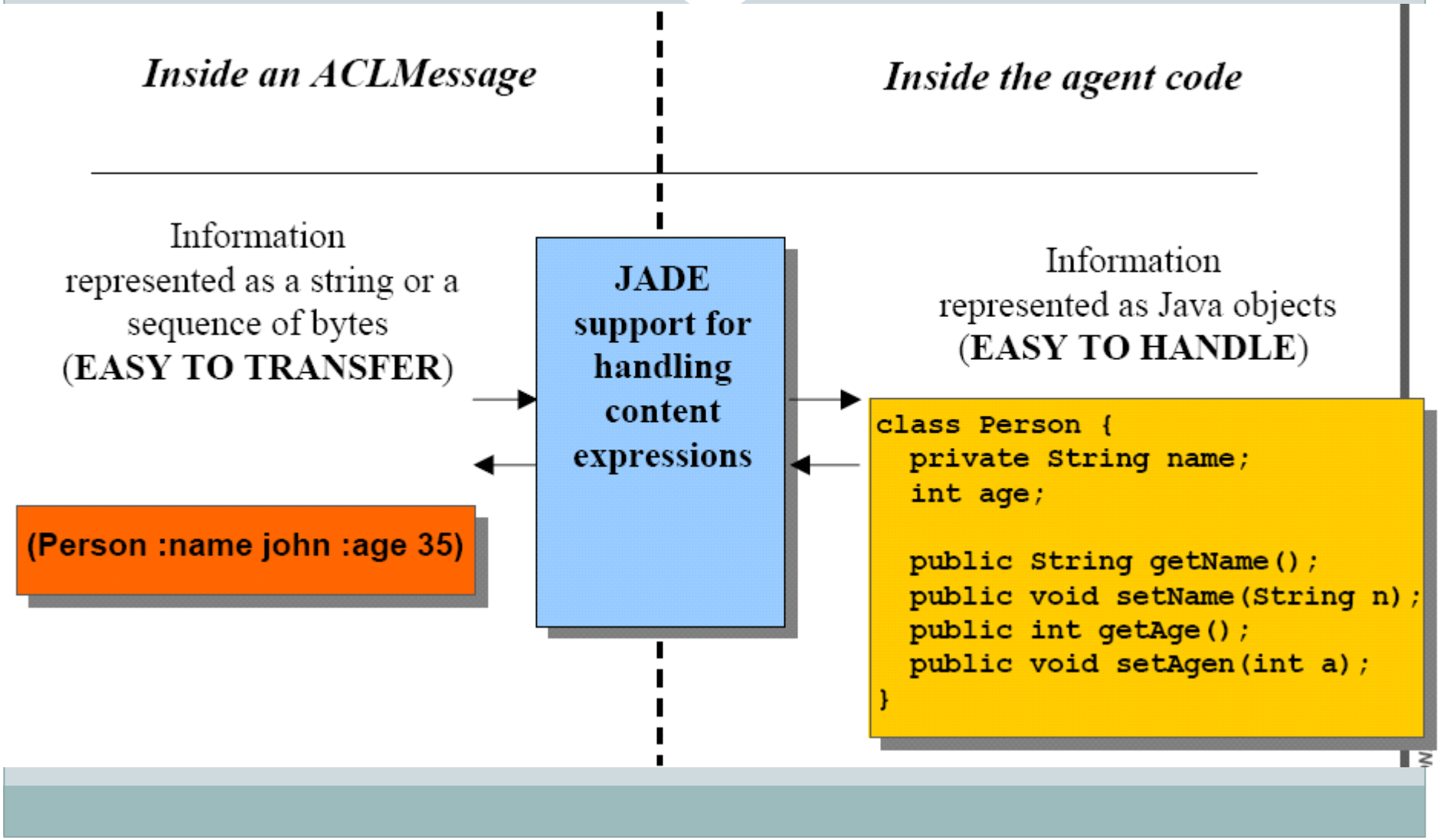
Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Outline

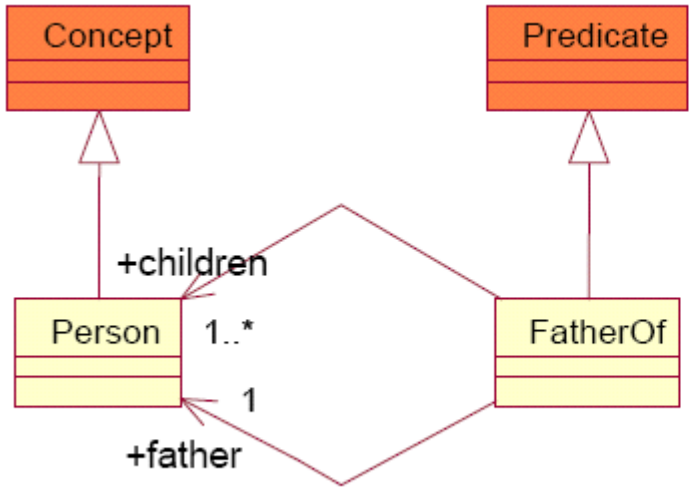
- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Soporte Jade para manejo del contenido de los mensajes



Como trabaja…

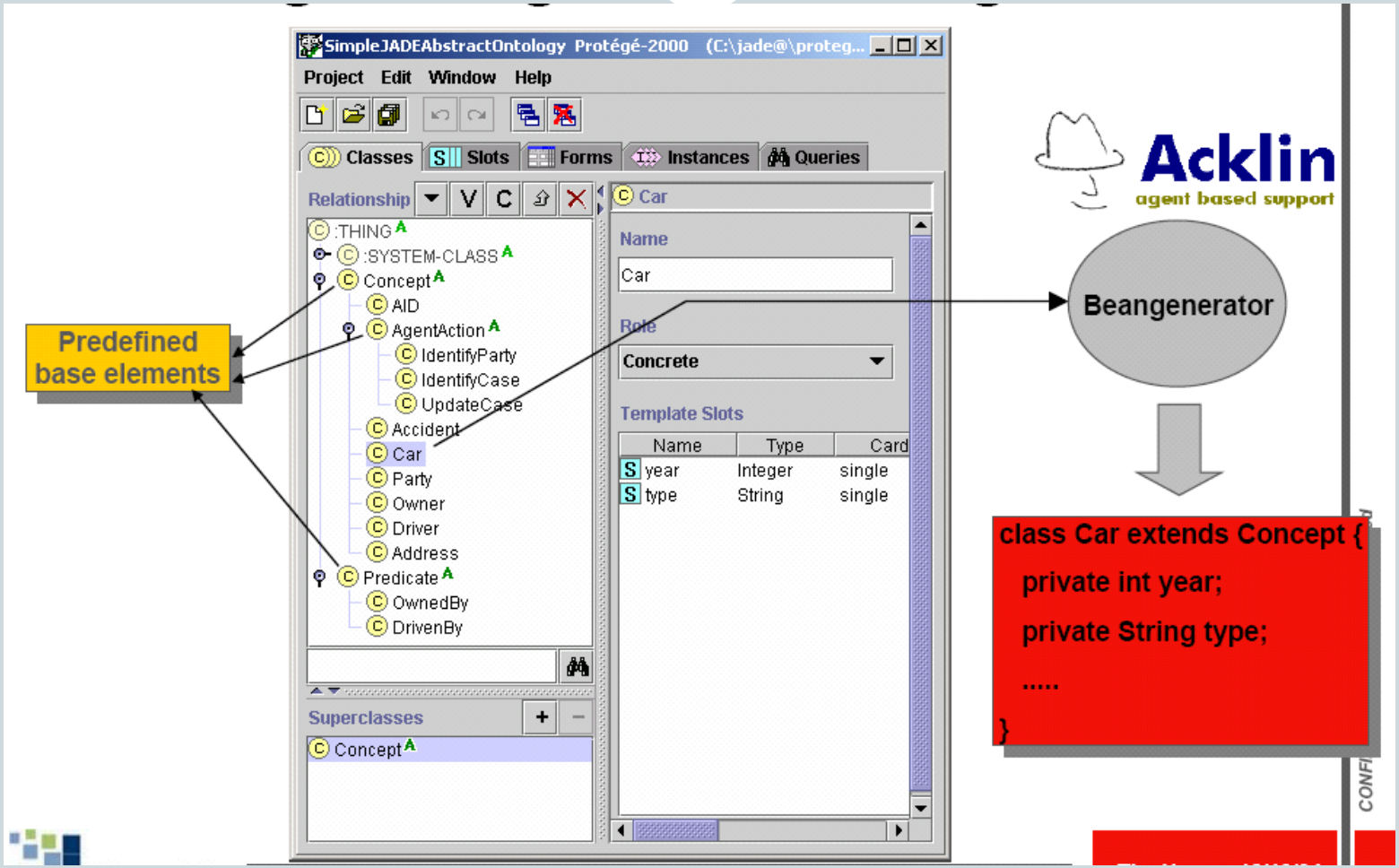
- Creación de una ontología (específica de dominio)
  - Definición de los esquemas de los elementos de la ontología.
  - Definición de las clases Java correspondientes.
    - Manejo de las expresiones contenido como objetos Java.
    - Utilización del ContentManager para llenar y procesar el contenido de los mensajes.




```
Person john = new Person("John", 35);
Person bill = new Person("Bill", 67);
FatherOf f = new FatherOf();
f.setFather(bill);
f.addChildren(john);
```

```
(father-of (person :name Bill :age 67) (set (person :name John :age 35) ) )
```

Creación de ontologías con Protege



Jade Content Reference model

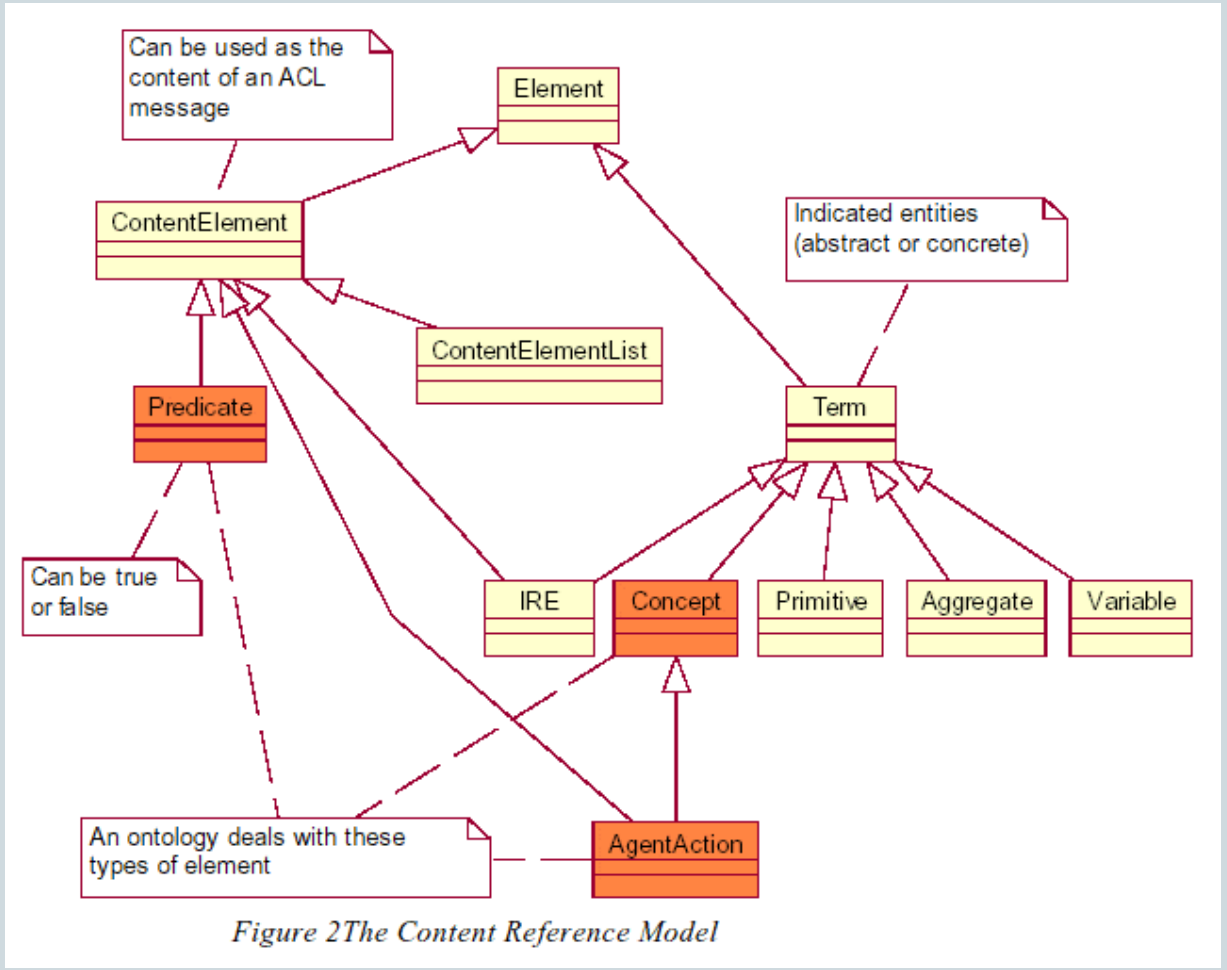


Figure 2The Content Reference Model

Documentación sobre ontologías

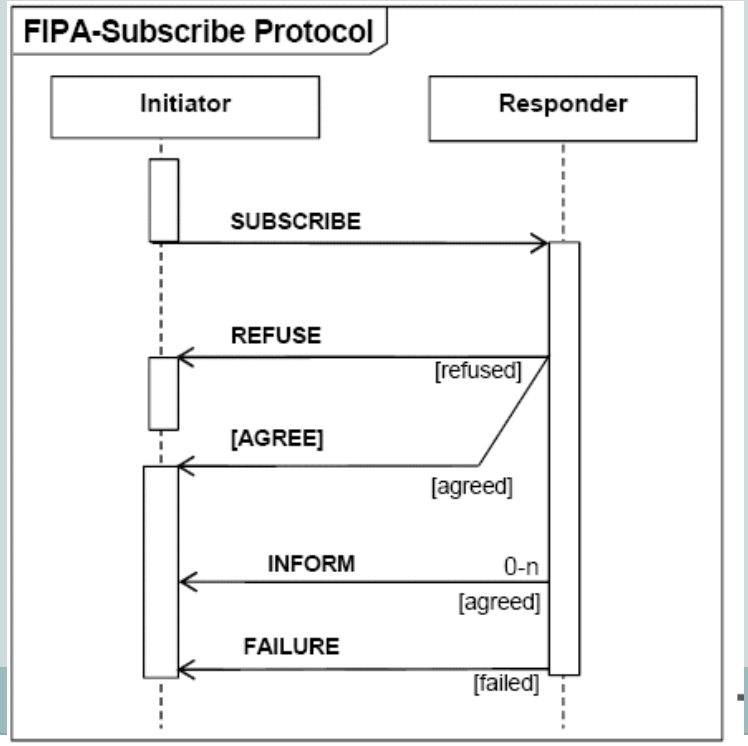
- **Un tutorial completo en la Web de Jase:**
  - **http://jade.tilab.com/doc/CLOntoSupport.pdf**
  - **API (javadoc): jade.content package and subpackages**
  - **Ejemplos: examples.content package en la distribución de JADE**

Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Protocolos de Interacción

- **Dado un conjunto de performativas (actos del habla) predefinidos (INFORM, REQUEST, PROPOSE ….) se pueden especificar secuencias de mensajes intercambiados por los agentes en base a dichas performaticas**
  - **• Protocolos de Interacción FIPA**



Protocolos de Interacción

- **jade.protopackage contiene comportamientos para los roles initiator y responder de los protocolos más habituales:**
  - FIPA-request (AchieveREInitiator/Responder)
  - FIPA-Contract-Net (ContractNetInitiator/Responder)
  - FIPA-Subscribe (SubscriptionInitiator/Responder)
- **Estas clase tratan**
  - Que el flujo de mensajes sigua el protocolo
  - Los timeouts (si los hay)
- **Suministran métodos callback que deberían redefinirse para realizar las acciones necesarias cuando por ejemplo se recibe un mensaje o espira un timeout.**

Protocolos de Interacción

- **Véase apítulo 3.5 de la guía de programación para más detalles**
- **API (javadoc): jade.proto package**
- **Ejemplos : examples.protocols package en la distribución.**

Programación Protocolo sencillo en JADE

- Creación Agente que recive mensajes y responde

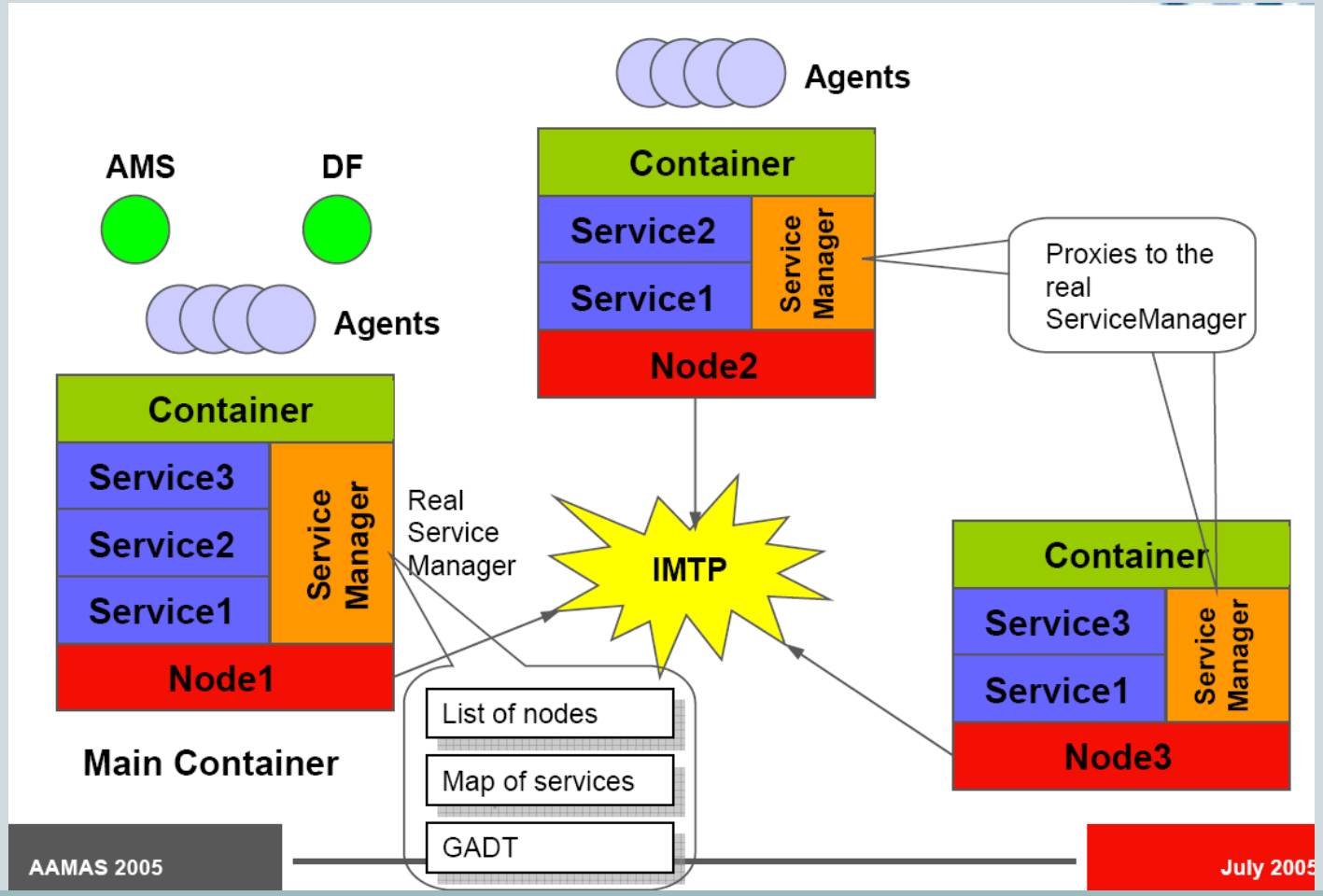
```
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
public class HelloWorldAgent extends Agent {

        public void setup() {
                System.out.println("Hello. My name is "+getLocalName());
                addBehaviour(new CyclicBehaviour() {
                        public void action() {
                                ACLMessage msgRx = receive();
                                if (msgRx != null) {
                                        System.out.println(msgRx);
                                        ACLMessage msgTx = msgRx.createReply();
                                        msgTx.setContent("Hello!");
                                        send(msgTx);
                                } else {block();}
                        }
                });
        }
}
```

Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

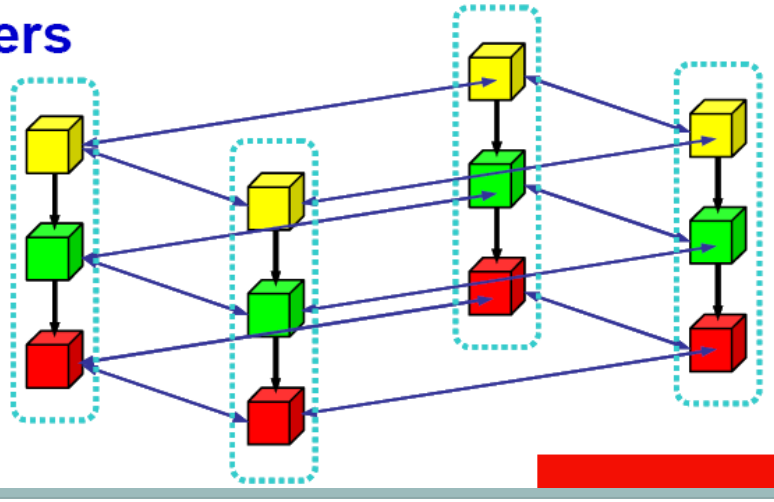
Internal Architecture



Aspect Oriented Programming Inspiratio

The Distributed coordinated filters

- **Inspiration from Aspect Oriented Programming**
  - Separation of concerns + Aspect Weaving
  - Composition Filter approach: Each object is provided with
    - An incoming filter chain whose filters are invoked whenever the object receives a method call
    - An outgoing filter chain whose filters are invoked when the object is about to call another object's method
- **Distributed coordinated filters**
  - Aspect => Service
  - Object => Node
  - Each Service is "sliced" over the nodes



Outline

- JADE
  - Introducción
  - Modelo Arquitectural
  - Principales herramientas gráficas
  - Programación con Jade
    - Características Básicas
      - **Creación de Agentes**
      - **Tareas de los agentes** (Comportamientos)
      - **Comunicación entre agentes**
      - **Servicio de Páginas Amarillas**
    - Características Avanzadas
      - **Gestión de expresiones de contenido**
      - **Protocolos de interacción**
      - **Trabajo con AMS**
      - **Movilidad, seguridad....**
  - Arquitectura Interna
  - Ejemplo

Book Trading Example

```
/**
 * skeleton of the Book-BuyerAgent class.
 **/
package bookTrading.buyer;
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import jade.domain.*;
import jade.domain.FIPAAgentManagement.*;
import java.util.Vector;
import java.util.Date;


public class BookBuyerAgent extends Agent {
  // The list of known seller agents
  private Vector sellerAgents = new Vector();

  // The GUI to interact with the user
  private BookBuyerGui myGui;
```

Setup

```
/**
 * Agent initializations
 **/
  protected void setup() {
    // Printout a welcome message
    System.out.println("Buyer-agent "+getAID().getName()+" is ready.");

    // Get names of seller agents as arguments
    Object[] args = getArguments();
    if (args != null && args.length > 0) {
      for (int i = 0; i < args.length; ++i) {
        AID seller = new AID((String) args[i], AID.ISLOCALNAME);
        sellerAgents.addElement(seller);
      }
    }
     // Show the GUI to interact with the user
    myGui = new BookBuyerGuiImpl();
    myGui.setAgent(this);
    myGui.show();
```

Setup

```
// Update the list of seller agents every minute
    addBehaviour(new TickerBehaviour(this, 60000) {
      protected void onTick() {
        // Update the list of seller agents
        DFAgentDescription template = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("Book-selling");
        template.addServices(sd);
        try {
          DFAgentDescription[] result = DFService.search(myAgent, template);
          sellerAgents.clear();
          for (int i = 0; i < result.length; ++i) {
            sellerAgents.addElement(result[i].getName());
          }
        }
        catch (FIPAException fe) {
          fe.printStackTrace();
        }
      }
    } );  // End Ticker Behaviour
  }  //End Setup
```

takeDown

```
/**
* Agent clean-up
**/
  protected void takeDown() {
    // Dispose the GUI if it is there
    if (myGui != null) {
      myGui.dispose();
    }

    // Printout a dismissal message
    System.out.println("Buyer-agent "+getAID().getName()+"terminated.");
  }
```

Purchase, adds the Ticked Behaviour **PurchaseManager**

```
 /**
   * This method is called by the GUI when the user inserts a new
   * book to buy
   * @param title The title of the book to buy
   * @param maxPrice The maximum acceptable price to buy the book
   * @param deadline The deadline by which to buy the book
  **/

public void purchase(String title, int maxPrice, Date deadline) {
    // the following line is in the book at page 62
    addBehaviour(new PurchaseManager(this, title, maxPrice, deadline));
  }

  /**
   * This method is called by the GUI. At the moment it is not implemented.
   **/
  public void setCreditCard(String creditCarNumber) {
  }
```

PurchaseManager periodically adds the Behaviour **BookNegotiator**

```
private class PurchaseManager extends TickerBehaviour {
   private String title;
   private int maxPrice, startPrice;
   private long deadline, initTime, deltaT;

   private PurchaseManager(Agent a, String t, int mp, Date d) {
     super(a, 60000); // tick every minute
     title = t;
     maxPrice = mp;
     deadline = d.getTime();
     initTime = System.currentTimeMillis();
     deltaT = deadline - initTime;
   }


...
```

PurchaseManager periodically adds the Behaviour **BookNegotiator**

```
 private class PurchaseManager extends TickerBehaviour {

…
    public void onTick() {
      long currentTime = System.currentTimeMillis();
      if (currentTime > deadline) {
        // Deadline expired
        myGui.notifyUser("Cannot buy book "+title);
        stop();
      }
      else {
        // Compute the currently acceptable price and start a negotiation
        long elapsedTime = currentTime - initTime;
        int acceptablePrice = (int)Math.round(1.0 * maxPrice * (1.0 *
  elapsedTime / deltaT));
        // System.out.println("elapsedTime"+elapsedTime+"deltaT"+deltaT
  +"acceptablePrice"+acceptablePrice+"maxPrice="+maxPrice);
        myAgent.addBehaviour(new BookNegotiator(title, acceptablePrice, this));
      }
    }
  }
```

BookNegotiator: *Intermingled version*

```
/**
 * Inner class BookNegotiator.
 * This is the behaviour used by Book-buyer agents to actually
 * negotiate with seller agents the purchase of a book.
**/
private class BookNegotiator extends Behaviour {
  private String title;
  private int maxPrice;
  private PurchaseManager manager;
  private AID bestSeller; // The seller agent who provides the best offer
  private int bestPrice; // The best offered price
  private int repliesCnt = 0; // The counter of replies from seller agents
  private MessageTemplate mt; // The template to receive replies
  private int step = 0;

  public BookNegotiator(String t, int p, PurchaseManager m) {
    super(null);
    title = t;
    maxPrice = p;
    manager = m;
  }
```

BookNegotiator: *Intermingled version*

```
public void action() {
    switch (step) {
      case 0:
        // Send the cfp to all sellers
        ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
        for (int i = 0; i < sellerAgents.size(); ++i) {
          cfp.addReceiver((AID)sellerAgents.elementAt(i));
        }
        cfp.setContent(title);
        cfp.setConversationId("book-trade");
        cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
        myAgent.send(cfp);
        myGui.notifyUser("Sent Call for Proposal");

        // Prepare the template to get proposals
        mt = MessageTemplate.and(
        MessageTemplate.MatchConversationId("book-trade"),
        MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
        step = 1;
        break;
```

BookNegotiator: *Intermingled version*

```
case 1: // Receive all proposals/refusals from seller agents
        ACLMessage reply = myAgent.receive(mt);
        if (reply != null) { // Reply received
          if (reply.getPerformative() == ACLMessage.PROPOSE) {
            // This is an offer
            int price = Integer.parseInt(reply.getContent());
            myGui.notifyUser("Received Proposal at "+price+
                             " when maximum acceptable price was "+maxPrice);
            if (bestSeller == null || price < bestPrice) {
              bestPrice = price; // This is the best offer at present
              bestSeller = reply.getSender();
            }
          }
          repliesCnt++;
          if (repliesCnt >= sellerAgents.size()) {
            // We received all replies
            step = 2;
          }
        }
        else { block(); }   //waiting messages
        break;
```

BookNegotiator: *Intermingled version*

```
case 2:
        if (bestSeller != null && bestPrice <= maxPrice) {
          // Send the purchase order to the seller that provided the best offer
          ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
          order.addReceiver(bestSeller);
          order.setContent(title);
          order.setConversationId("book-trade");
          order.setReplyWith("order"+System.currentTimeMillis());
          myAgent.send(order);
          myGui.notifyUser("sent Accept Proposal");
          // Prepare the template to get the purchase order reply
          mt = MessageTemplate.and(
            MessageTemplate.MatchConversationId("book-trade"),
            MessageTemplate.MatchInReplyTo(order.getReplyWith()));
          step = 3;
        }
        else {
          // If we received no acceptable proposals, terminate
          step = 4;
        }
        break;
```

BookNegotiator: *Intermingled version*

```
    case 3: // Receive the purchase order reply
          reply = myAgent.receive(mt);
          if (reply != null) {
            // Purchase order reply received
            if (reply.getPerformative() == ACLMessage.INFORM) {
              // Purchase successful. We can terminate
              myGui.notifyUser("Book "+title+
                               " successfully purchased. Price = " + bestPrice);
              manager.stop();
            }
            step = 4;
          }
          else { block(); }
          break;
      } // end of switch
    }
    public boolean done() {
      return step == 4;
    }
  } // End of inner class BookNegotiator
}
```

Book negotiator: Using protocol classes

Protocol	Initiator Class	Responder Class
FIPA-Request	`AchieveREInitiator`	`AchieveREResponder`
FIPA-Query		
FIPA-Propose	`ProposeInitiator`	`ProposeResponder`
Iterated version of	`IteratedAchieveREInitiator`	`SSIteratedAchieveREResponder`
FIPA-Request		
FIPA-Query		
Contract-Net	`ContractNetInitiator`	`ContractNetResponder` `SSContractNetResponder`
FIPA-Suscribe	`SubscriptionInitiator`	`SubscriptionResponder`

Protocol Classes

- Las clases protocolo ofrecen un número de métodos callback que los programadores deben redefinir de acuerdo a la lógica de la aplicación:
  - Los métodos callbak del que inicia y el que responde son invocados con la recepción de un mensaje del protocolo y tienen la forma

```
Protected handle<message-performative>(ACLMessage

    receivedMessage)
```

Protocols handlers

- ContractNetResponder sample:

```
Protected ACLMessage handleCfp (ACLMessage cfp) {

   ACLMessage reply = cfp.createReply();
   // Evaluate the call
   if (call OK)  {
   // prepare a proposal
   reply.setPerformative(ACLMessage.PROPOSE);
   }
   else {
   reply.setPerformative(ACLMessage.REFUSE);
   }
   return reply;
}
```

Protocols handlers

- ContractNetInitiator sample:

```
// Methods triggered by the reception of protocol messages
Protected ACLMessage hadnlePropose (ACLMessage propose, Vector acceptances) {
   ACLMEssage reply = cfp.createReply();
   // Evaluate the call
   if (call OK)  {
   // prepare a proposal
   reply.setPerformative(ACLMessage.PROPOSE);
   }
   else {
   reply.setPerformative(ACLMessage.REFUSE);
   }
   acceptances.add(reply);
}
// Handlers when the replies of all responders have been collected
Protected ACLMessage hadnleAllPropose (Vector acceptances) {
…
}
```

Protocols handlers

- ContractNetInitiator sample:

```
Protected ACLMessage hadnlePropose (ACLMessage propose, Vector acceptances) {

   ACLMEssage reply = cfp.createReply();
   // Evaluate the call
   if (call OK)  {
   // prepare a proposal
   reply.setPerformative(ACLMessage.PROPOSE);
   }
   else {
   reply.setPerformative(ACLMessage.REFUSE);
   }
   acceptances.add(reply);
}
```

Protocols handlers

- ContractNetInitiator sample:

```
Protected ACLMessage hadnlePropose (ACLMessage propose, Vector acceptances) {

   ACLMEssage reply = cfp.createReply();
   // Evaluate the call
   if (call OK)  {
   // prepare a proposal
   reply.setPerformative(ACLMessage.PROPOSE);
   }
   else {
   reply.setPerformative(ACLMessage.REFUSE);
   }
   acceptances.add(reply);
}
```

BookNegotiator: *with contract net behaviour*

```
public ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
// variable needed to the ContractNetInitiator constructor
/*** This is the behaviour reimplemented by using the ContractNetInitiator **/
  public class BookNegotiator extends ContractNetInitiator {
    private String title;
    private int maxPrice;
    private PurchaseManager manager;

    public BookNegotiator(String t, int p, PurchaseManager m) {
      super(BookBuyerAgent.this, cfp);
      title = t;
      maxPrice = p;
      manager = m;
      Book book = new Book();
      book.setTitle(title);
      Sell sellAction = new Sell();
      sellAction.setItem(book);
      Action act = new Action(BookBuyerAgent.this.getAID(), sellAction);
      try { cfp.setLanguage(codec.getName());
            cfp.setOntology(ontology.getName());
            BookBuyerAgent.this.getContentManager().fillContent(cfp, act);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
```

BookNegotiator: *with contract net behaviour*

```
protected Vector prepareCfps(ACLMessage cfp) {
      cfp.clearAllReceiver();
      for (int i = 0; i < sellerAgents.size(); ++i) {
        cfp.addReceiver((AID) sellerAgents.get(i));
      }
      Vector v = new Vector();
      v.add(cfp);
      if (sellerAgents.size() > 0)
         myGui.notifyUser("Sent Call for Proposal to "
                          +sellerAgents.size()+" sellers.");
      return v;
    }
```

BookNegotiator: *with contract net behaviour*

```
protected void handleAllResponses(Vector responses, Vector acceptances) {
      ACLMessage bestOffer = null;
      int bestPrice = -1;
      for (int i = 0; i < responses.size(); i++) {
        ACLMessage rsp = (ACLMessage) responses.get(i);
        if (rsp.getPerformative() == ACLMessage.PROPOSE) {
           int price = Integer.parseInt (rps.getContent ());
           if (bestOffer == null || price < bestPrice) {
              bestOffer = rsp;
              bestPrice = price;
            }
        }
      }
       if (bestOffer != null) {
         ACLMessage accept = bestOffer.createReply();
         accept.setContent(title);
         acceptances.add(accept);
       }
 }
// Pero el codigo en la Web es mas complejo!! Responde a todos aceptando o
   rechazando
```

BookNegotiator: *with contract net behaviour*

```
  protected void handleInform(ACLMessage inform) {
      // Book successfully purchased
      int price = Integer.parseInt(inform.getContent());
      myGui.notifyUser("Book "+title+" successfully purchased. Price =" + price);
      manager.stop();
    }

  } // End of inner class BookNegotiator
}
```


THE
END

Sharing Data between sub-behaviours (datastore)

```
Public clas MySequentialBehaviour extends SequentialBehaviour {
   private ACLMessage receivedMsg;

   public MySequentialBehaviour (Agent a) {
   super(a);
   //..
   addSubBehaviour(new SimpleBehaviour(a) {
     private boolean finished = false:
     public void action() {
          receivedMesg = myAgent.receive();
          if (receiveMsg != null) {
              finished = true;
          }
          else { block(); }
      public boolean done() {
          return finished;
      }
    });
   addSubBehaviour (new OneShotBehaviour(a) {
      public action() {
          //process receivedMsg
      }
   });
 }
}
```

Sharing Behaviours and dataStores

```
Public clas MessageReceiver extends SimpleBehaviour {
   public static final String RECV_MSG = "received-message";
   private boolean finished = false;
   public action() {
      ACLMessage msg  = myAgent.receive();
      if (msg!=null) {
       getDataStore().put(RECV-MSG, msg);
       finisjed = true;
      }
      else { block(); }
    }
     public boolean done() { return finished; }
}
// SEQUENTIAL BEHAVIOUR THAT TAKE ADVANTAGE OF THE MessageReceiverClass
Sequential Behaviour sb = new SequentialBehaviour (anAgent);

Behaviour b = new MessageReceiver(anAgent);
b.setDataStore(sb.getDataStore()) // The dataStore of b, is the dataSotre of sb.
Sb.addSubBehaviour(b)

B = new OneShotBehaviour(anAgent) {
   public void action () {
      ACLMessage receivedMsg = getDataStore().get(MessageReceiver.RECV_MSG);
      //Process receivedMsg
}};
```

Each behabiour has its own dataStore, which extends HashMap.

Nested Behaviours

- Use Case:
  - A Broker Agent acts as a responder in a FIPA-Request protocol, but actually delegates the execution of the requested action to an executor selected from a pool of executor agents by means of a Contract-Net protocol
    - We may implement FIPA-REQUEST responder role using an `AchieveREResponder` => We should be able to carry out the Contract-Net Protocol with the executor agents within the `prepareResultNotification()` method

This prevents us from using the ContractNetIterator class since it is not possible to execute a behaviour from within a method.

Nested Behaviours: Solution

- Jade Protocol classes are implemented as subclasses of `FMSBehaviour` and each callback is invoked in a dedicated state of the FSM.

FSM of AchieveREResponder

States dedicated to invoing callback methods are in gray:

For each callback method `mmm()` *there is a* `registerMmm()` *method that allows overriding fo the state that invokes the callback method* `mmm()` *with an application-specific behaviour*