Introducción a JADE

Luis Oliva Felipe

Ignasi Gómez

Luis Oliva

Sergio Álvarez

{loliva,igomez,ia,salvarez}@cs.upc.edu

SID2022

https://kemlg.upc.edu

Knowledge Engineering and Machine Learning Group

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Setup del entorno

- Descargad jade.jar, commons-codec-1.3.jar y ejemplos
  - https://jade.tilab.com/dl.php?file=JADE-all-4.5.0.zip
  - https://repo1.maven.org/maven2/commons-codec/commons-codec/1.3/commons-codec-1.3.jar
  - https://gitlab.fib.upc.edu/sergio.alvarez-napagao/material-sid
- Aseguraos que tenéis bien configurado el CLASSPATH
  - `JADE-all-4.5.0/JADE-bin-4.5.0/jade/lib/jade.jar`
  - `commons-codec-1.3.jar`
  - Directorio que guarda los `.class` (e.g. `output/` o `classes/`)
  - `export CLASSPATH="$CLASSPATH:<JARS_AND_FOLDERS>"`
- Arrancamos la GUI
  - `java jade.Boot –gui`
  - Si no tenemos el CLASSPATH siempre podemos usar –cp
    - `java –cp lib/jade.jar jade.Boot –gui`

Agente

- Una clase de agente se crea…
  - Extendiendo la clase `jade.core.Agent`
  - Y redefiniendo el método `setup()`
- Cada instancia se identifica con un AID (`jade.core.AID`)
  - Un AID se compone de un nombre único más la dirección de la plataforma
  - Podemos recuperar el AID de un agente mediante el método `getAID()` de la clase `Agent`

```
import jade.core.Agent;

public class HalloWorldAgent extends Agent {

  protected void setup() {
    System.out.println("Hallo World! my name is "+getAID().getName());
  }
}
```

Nombres locales, GUID y direcciones

- Nombre agente: `<nombre-local>@<nombre-plataforma>`
  - `Nombre-local` es único (localmente)
  - El nombre completo ha de ser único globalmente
- Podemos especificar el nombre de la plataforma al arrancarla con la opción *–name*
- Dentro de la plataforma nos referimos al agente usando únicamente su `nombre-local`

```
– AID id = new AID(localname, AID.ISLOCALNAME);
– AID id = new AID(name, AID.ISGUID);
```

Paso de parámetros

- Podemos pasar parámetros a un agente
  - `java jade.Boot .. ´A:myPackage.MyAgent(arg1 arg2)´`
- Los recuperamos usando el método `getArguments()` de la clase `Agent`

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

Expiración del agente

- El agente termina su ejecución cuando se llama a su método `doDelete()`
- Durante su terminación se invoca el método `takeDown()` (e.g., para realizar operaciones de limpieza)

```
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...");
}
```

Ejercicios I

1. Añadid un `Sniffer` y un `Introspector`
2. Añadid dos `DummyAgent`
   - Al añadirlos se abre una ventana para enviar mensajes (es para lo que sirven principalmente)
3. Desde la ventana de `Sniffer`, añadid los dos `dummy agent`
4. Desde la ventana de `Introspector`, haced `Debug On` de alguno de los dos `dummy agent`
5. Desde el que no estáis vigilando con `Introspector`, enviad un mensaje al otro
   - Name: Tenéis que poner el `local-name@platform`
     - E.g.: da0@192.168.1.123:1099/JADE
   - Añadid cualquier cosa al `content` y enviad (icono del sobre)

Ejercicios II

1. Para añadir nuestros agentes tenemos que generar el `.class` y añadirlos al CLASSPATH al arrancar la GUI
2. Añadid el source de `PingAgent`
3. Compiladlo (cread antes la carpeta classes)
   - `javac -cp lib/jade.jar -d classes src/examples/PingAgent/PingAgent.java`
4. Arrancad la GUI
   - Añadid un `PingAgent` y un `DummyAgent`
   - Añadid un `Introspector` y un `Sniffer`
   - Enviad un mensaje a `PingAgent`, ¿qué sucede?
   - Mirad el código de `PingAgent` para entender qué hay que decirle

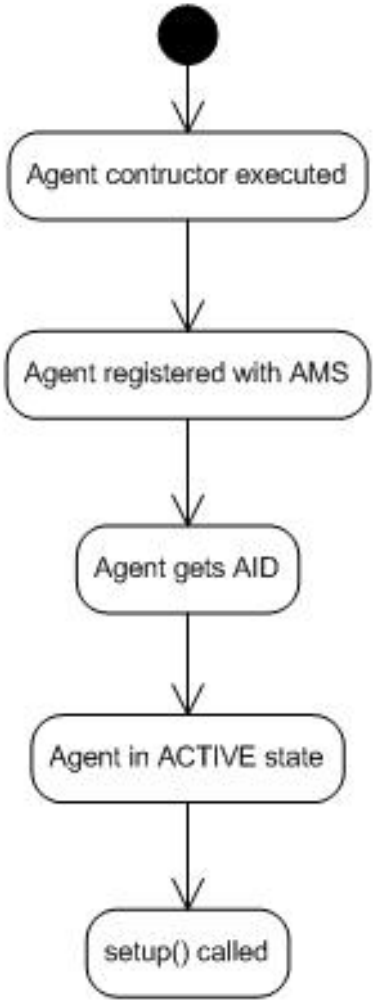
Ejercicios III

- Abrir el código de `HelloWorldAgent`
  - ¿Qué importa?
  - ¿Qué extiende?
  - Obtén su AID y muéstralo
  - Pasadle parámetros y que los muestre (ojo no os dejéis ‘’)
  - Modifica y ejecuta
    - Desde la GUI
    - Desde consola (cargad el agente desde el comando)
    - Mirad los resultados en consola

Construyendo un agente

- `setup()`: método para preparar el agente; se ejecuta una vez
  - Registro en el DF
  - Iniciar todos los *behaviours* requeridos
    - Es obligatorio que haya al menos un *behaviour* en el setup
- `takeDown()`: al finalizar el agente
  - Eliminar del registro del DF
  - Cerrar todos los recursos abiertos/requeridos…
- Modificad el agente `HelloWorldAgent` para que se despida al terminar

Ciclo de vida del agente I



- `addBehaviour()`
- `removeBehaviour()`
- `blockingReceive()`

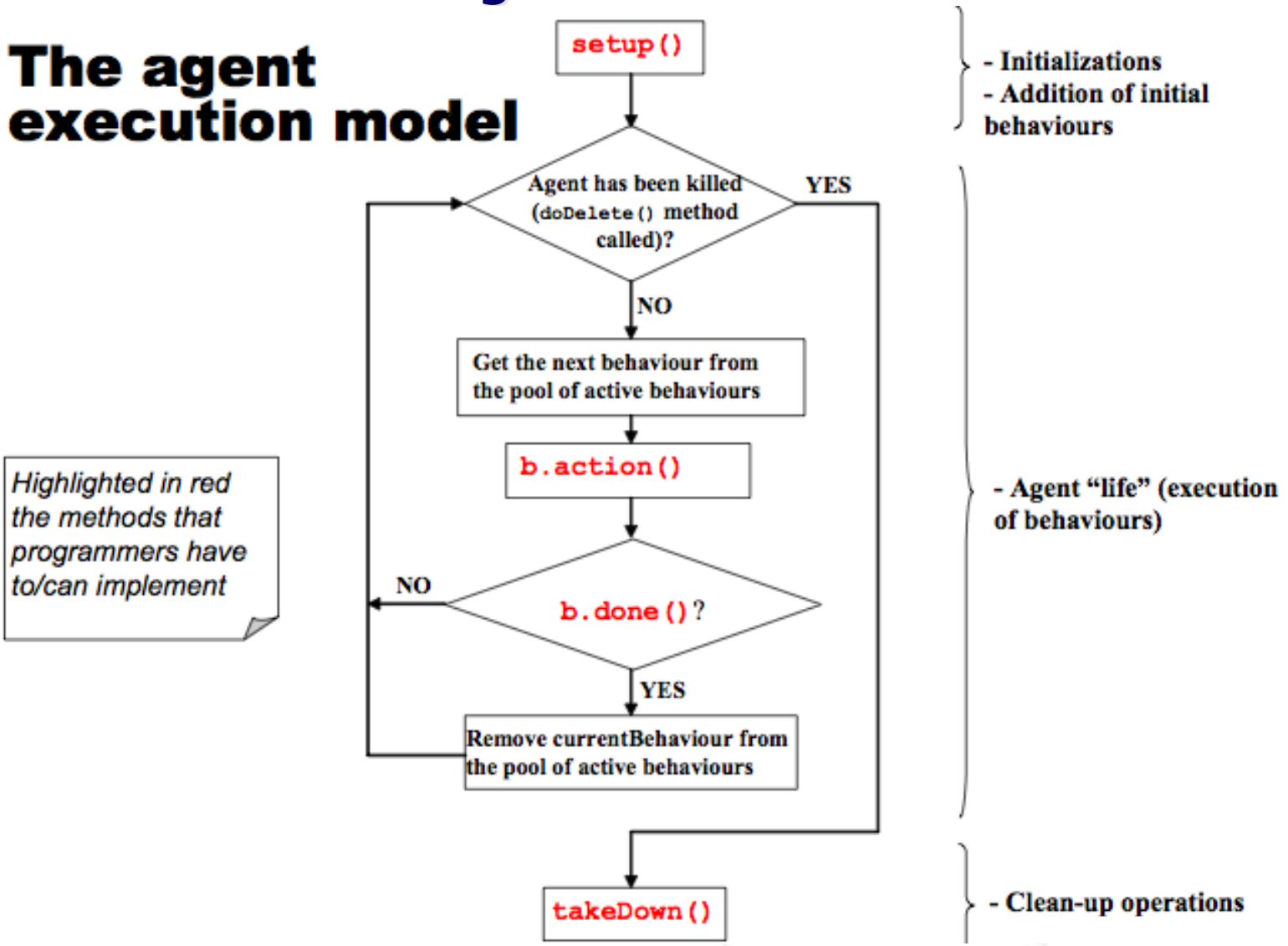
Comportamientos (*Behaviours*)

- Los agentes realizan sus tareas mediante *behaviours*
- Los *behaviours* se crean extendiendo la clase `jade.core.behaviours.Behaviour`
- Para que un agente ejecute una tarea basta con
  - Crear una instancia de la subclase `Behaviour` correspondiente y
  - Llamar al método `addBehaviour()` de la clase `Agent`
- Cada *behaviour* debe implementar:
  - `public void action()` : Qué hace el *behaviour*
  - `public boolean done()` : Cierto si ya ha finalizado

Planificación y ejecución de Behaviours

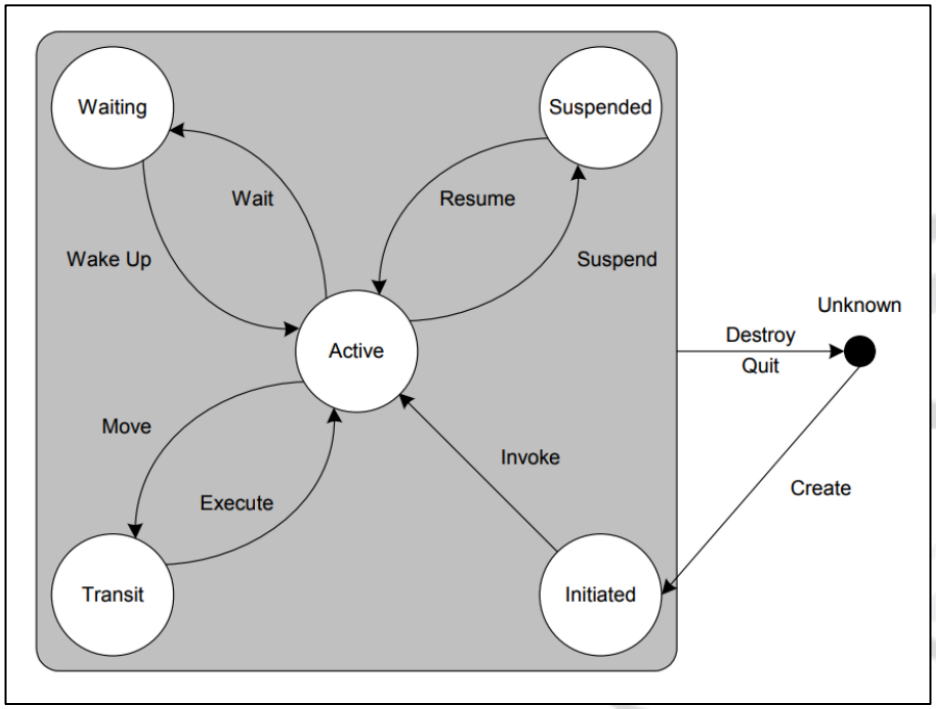
- Un agente puede ejecutar varios comportamientos en paralelo
- La planificación de estas ejecuciones no es **preferente**
  - No es determinista a la hora de seleccionar un behaviour si hay varios disponibles
    - Cola con round-robin con planificación colaborativa
  - Todo ocurre en el mismo thread de Java
  - Los cambios de behaviour a ejecutar ocurren cuando el método `action` del behaviour que se está ejecutando finaliza

Ciclo de vida del agente II



Estados del agente

State	Description
Initiated	The agent object is built, but hasn't registered itself with AMS, has neither name nor an addressand can't communicate with other agents.
Active	The agent object is registered with AMS, has regular name and address, can access all the various JADE features.
Suspended	The agent object is currently stopped. Its internal thread is suspended and no agent behaviour is being executed.
Waiting	The agent is blocked, waiting for somethink. Its internal thread is sleeping on a JAVA monitor and will wake up when some conditions are met (for example when message arrives).
Deleted	The agent is definitely dead. The intternal thread has terminated its execution and the agent is no more registered with AMS.
Transit	The mobile agent enters this state while it is migrating to the new location. The system continues to buffer messages that will then be sent to its new location.



Tipos de behaviours (I)

- One shot (`jade.core.behaviours.OneShotBehaviour`)
  - `action()`: se ejecuta una vez (se completa inmediatamente)
  - `done()`: simplemente devuelve `true`
- Cyclic (`jade.core.behaviours.CyclicBehaviour`)
  - `action()`: ejecuta la misma operación cada vez que se invoca (nunca se completa)
  - `done()`: simplemente devuelve `false`

Tipos de behaviours (II)

- JADE provee dos clases para implementar comportamientos que ejecutan operaciones en ciertos puntos de tiempo dados
- `WakerBehaviour`
  - Los métodos `action()` y `done()` ya están implementados para que el método `onWake()` (a implementar en las subclases) se ejecute tras un tiempo dado (*timeout*)
  - Tras su ejecución, el *behaviour* queda completado
- `TickerBehaviour`
  - Los métodos `action()` y `done()` ya están implementados para que el método `onTick()` (a implementar en las subclases) se ejecute periódicamente según un periodo dado
  - El behaviour se ejecuta para siempre a menos que se ejecute su método `stop()`

Tipos de behaviours (III)

- Los anteriores heredan de `SimpleBehaviour`
- Tenemos también `CompositeBehaviour`
  - `SequentialBehaviour`: Ejecuta todos los sub-behaviours secuencialmente
  - `ParallelBehaviour`: Ejecuta todos los sub-behaviours asíncronamente
  - `FSMBehaviour`: Ejecuta todos los sub-behaviours definiendo una máquina de estados
  - Utilizaremos el método `addSubBehaviour(Behaviour b)`

Más sobre behaviours

- `onStart()`
  - Se ejecuta una vez antes de ejecutar el método `action()`
  - Indicado para operaciones que tengan que ocurrir al inicio del behaviour
- `onEnd()`
  - Se ejecuta una vez después de que el método `done()` devuelva `true`
  - Indicado para operaciones que tengan que ocurrir al final del behaviour
- Cada behaviour tiene un apuntador al agente que lo ejecuta: el atributo protegido `myAgent`
- `removeBehaviour()`
  - Elimina un behaviour del pool de behaviours de un agente
  - No se llama al método `onEnd()`

Más sobre behaviours (II)

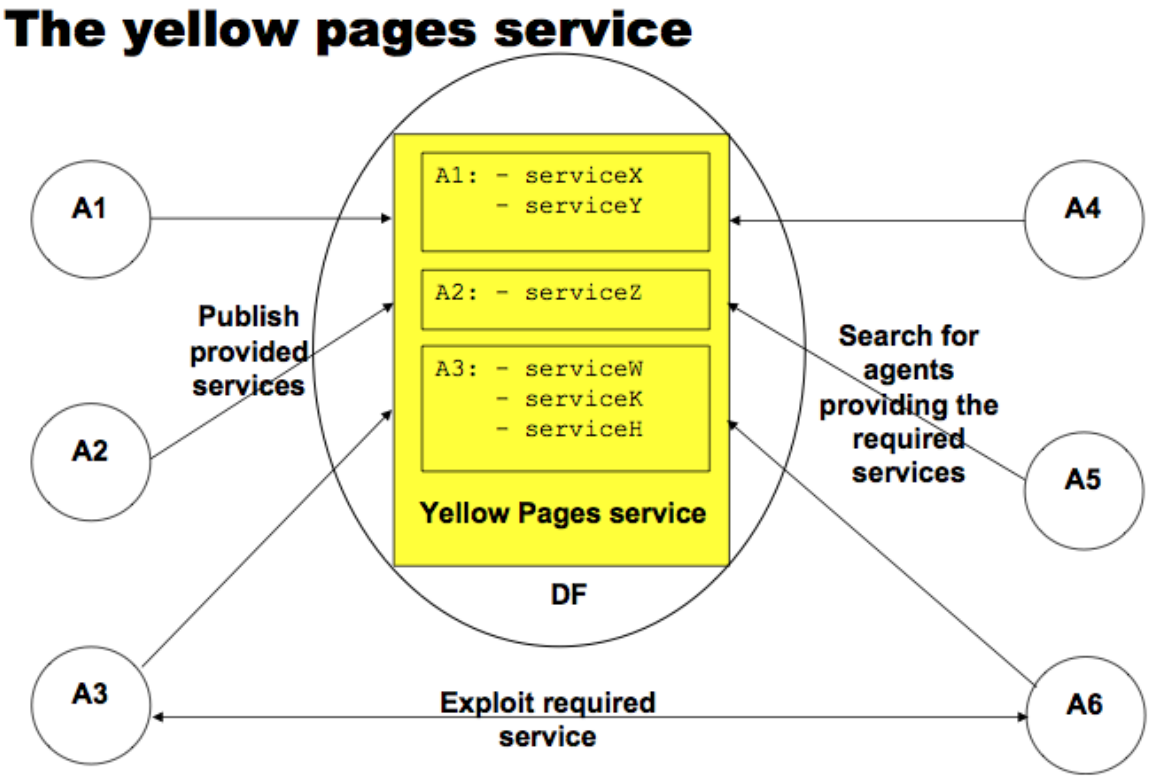
- Cuando el *pool* de behaviours de un agente está vacío, el agente pasa a un estado inactivo (IDLE state) y su thread entra en modo *sleep*

Ejercicios IV

- Implementad un agente para los siguientes behaviours
  - `OneShot`
  - `Waker`
  - `Cyclic`
  - `Ticker`
- Implementad las funciones
  - `action`
  - `onStart`
  - `onEnd`
  - `done`
  - `setup`
  - `doDelete`
  - `takeDown`
- Comprobad cómo se comporta cada uno y sus diferencias

Directory Facilitator (DF)

- Recordad que el DF es un agente también
- Clase `jade.domain.DFService`
  - `register()`
  - `modify()`
  - `deregister()`
  - `search()`



Registro

- Normalmente en el `setup` nos registraremos en el DF
  - Usaremos un `DFAgentDescription` para indicar
    - el ID del agente (`getAID()`),
    - los servicios que ofrece
  - Usaremos un ServiceDescription para cada servicio, indicando
    - el nombre del servicio (`string`)
    - el tipo de servicio (`string`)
    - una ontología (opcional)
    - un lenguaje del contenido (e.g., `FIPA_SL`, también opcional)

Registro

```
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setName(serviceName);
sd.setType("weather-forecast");
// Agents that want to use this service need to
// "know" the weather-forecast-ontology
sd.addOntologies("weather-forecast-ontology");
// Agents that want to use this service need to
// "speak" the FIPA-SL language
sd.addLanguages(FIPANames.ContentLanguage.FIPA_SL);
sd.addProperties(new Property("country", "Italy"));
dfd.addServices(sd);
```

Búsqueda

- Usaremos un `DFAgentDescription` y un `ServiceDescription` como template del agente/servicio que buscamos
- Usaremos un `SearchConstraints` para añadir restricciones a la búsqueda (como la cantidad de resultados)
- Enviaremos una petición de búsqueda al `DFService`
- De los resultados obtendremos el AID del proveedor

Búsqueda

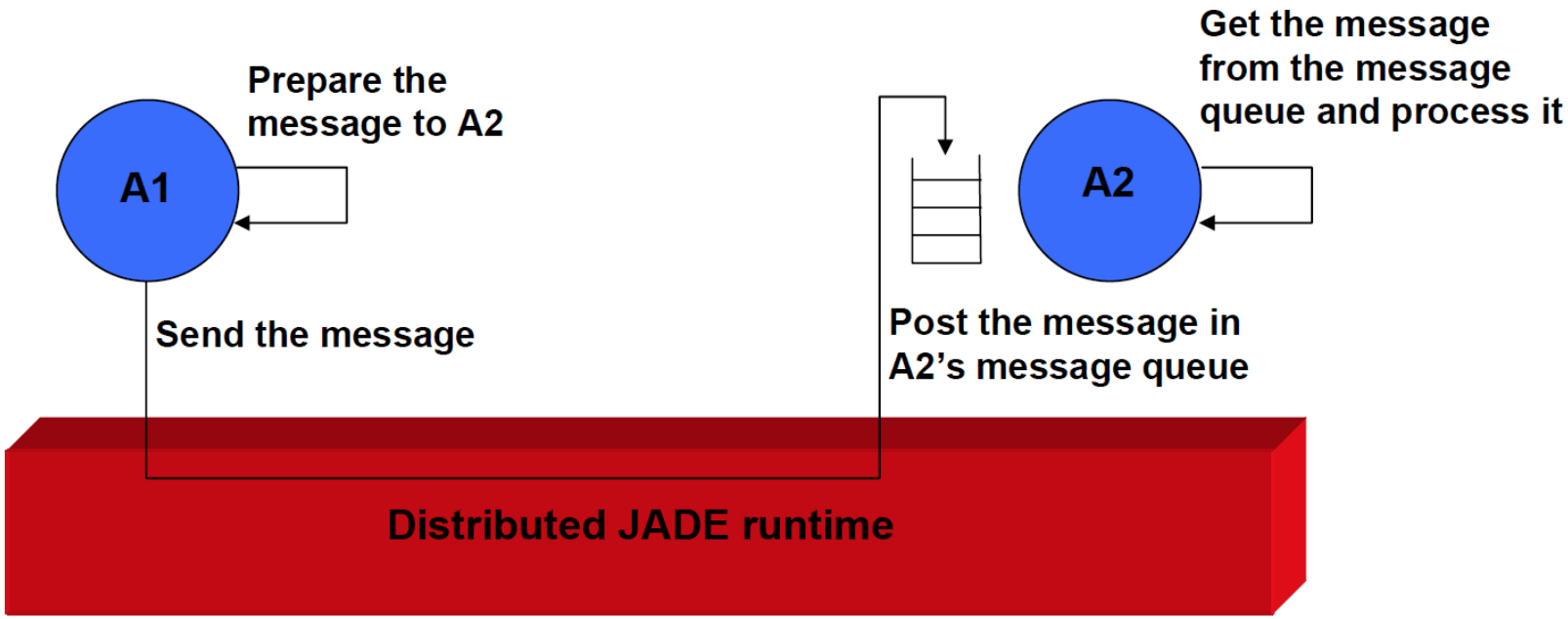
```
// Build the description used as template for the search
DFAgentDescription template = new DFAgentDescription();
ServiceDescription templateSd = new ServiceDescription();
templateSd.setType("weather-forecast");
template.addServices(templateSd);

SearchConstraints sc = new SearchConstraints();
// We want to receive 10 results at most
sc.setMaxResults(new Long(10));

DFAgentDescription[] results =
                    DFService.search(this, template, sc);
if (results.length > 0) {
  DFAgentDescription dfd = results[0];
  AID provider = dfd.getName();
}
```

Ejercicio V

- Cread un agente con comportamiento `OneShotBehaviour` que busque un agente que ofrezca un servicio `PingAgent`
- Cuando lo encuentre, que os diga su AID por consola
- Probad este agente con la GUI junto a un `PingAgent`

Modelo de comunicación

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



La clase ACLMessage

- Los mensajes intercambiados por los agentes son instancias de la clase `jade.lang.acl.ACLMessage`
- Define métodos de acceso (getters/setters)
  - `get/setPerformative()`
  - `get/setSender()`
  - `add/getReceiver()`
  - `get/setLanguage()`
  - `get/setOntology`
  - `get/setContent()`

Envío y recepción de mensajes I

- Enviar un mensaje consiste básicamente en crear un objeto `ACLMessage` y llamar al método `send()` de `Agent`

```
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);
```

- Para procesar mensajes de la cola de mensajes privada, el `Agent` ejecuta el método `receive()`

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

Envío y recepción de mensajes II

- El uso continuo de `receive()` es ineficiente
- Mejor utilizar `block()`
- Permite bloquear el *behaviour* del pool del agente sin bloquear a este
- Cada vez que llegue un mensaje se desbloquean los *behaviours* bloqueados para que puedan leerlo
- Método `blockingReceive()` disponible (pero peligroso)
  - Bloquea todo el agente y no deja que se ejecute ningún otro *behaviour* hasta que llegue el mensaje

```
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

Message templates

- Es más seguro hacer uso de *message templates*

```
MessageTemplate tpl = MessageTemplate.MatchOntology("Test-Ontology");

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

- Podemos hacer Match con los diferentes elementos de un mensaje:
  - `Conversation-id`, `Ontology`, `Language`, `Content`, `Performative`
  - Podemos hacer composición mediante `and`, `or` y `not`

Paso de mensajes

performative	passing info	requesting info	negotiation	performing actions	error handling
accept-proposal			x		
agree				x	
cancel		x		x	
cfp			x		
confirm	x				
disconfirm	x				
failure					x
inform	x				
inform-if	x				
inform-ref	x				
not-understood					x
propose			x		
query-if		x			
query-ref		x			
refuse				x	
reject-proposal			x		
request				x	
request-when				x	
request-whenever				x	
subscribe		x			

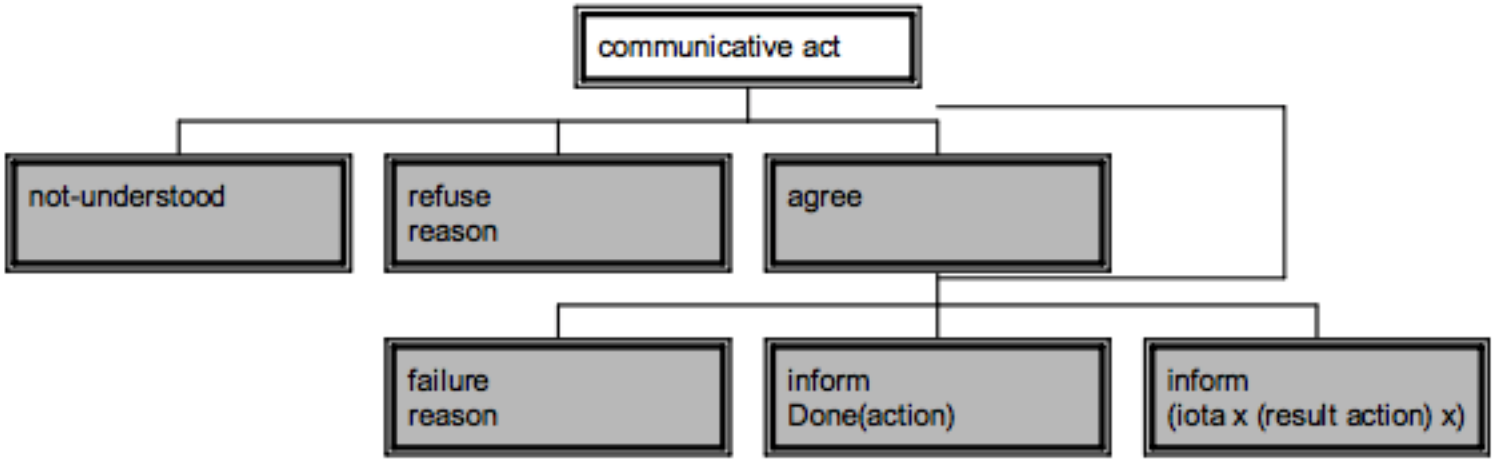
Ejercicio VI

- Al agente que hicisteis en el ejercicio V, haced que envíe un mensaje al `PingAgent`
- Recibid la respuesta

Ejercicios VII

- Cread un agente `Lover`
  - Lo que hace es buscar a otro agente (`SignificantOne`, por ejemplo) y si está, le envía un mensaje. Si no, se queja de que está solo (por consola).
  - Blocking behaviour
    - ¿Se bloquea el agente?
    - Recibir un mensaje usando un template
    - Enviar un mensaje usando un template
    - Uso de performativas
  - Ejecutad el agente y comprobad su comportamiento
    - GUI
    - Consola (Sniffer)
  - Usad el DF para registrar al `Lover` y al `SignificantOne`

Protocolos de interacción



Protocolos de interacción

- Secuencias predefinidas de intercambio de mensajes

- En todos ellos se distingue
  - Un *Initiator*, que permite gestionar varios *Responders* a la vez; es un comportamiento que
    - Termina una vez se alcanza un estado final
    - Se puede resetear para reutilizar el objeto (método `reset()`)

  - Uno o más *Responder*, que son implementados por behaviours cíclicos y que vuelven a reintroducirse una vez alcanzan un estado final

Soporte a los protocolos de interacción

- El paquete `jade.proto` contiene behaviours para los roles que inician y responden a los protocolos más comunes
  - FIPA-request (`AchieveREInitiator/Responder`)
  - FIPA-Contract-Net (`ContractNetInitiator/Responder`)
  - FIPA-Subscribe (`SubscriptionInitiator/Responder`)
- Todas estas clases automáticamente se encargan de
  - Comprobar el flujo de mensajes para validar que se corresponde al protocolo
  - De los timeouts (si hay alguno)
- Proveen de métodos de *callback* que se deben redefinir para tomar las acciones necesarias en los diferentes estados del protocolo; por ejemplo, cuando un mensaje es recibido o expira un timeout

AchieveREInitiator/Responder (I)

- A lo largo de las últimas versiones JADE ha ido aglutinando los protocolos bajo una misma interfaz
  - `AchieveREInitiator/Responder` o `SimpleAchieveRE /Responder` (más rápido pero con menos opciones)

- En general con esta clase podremos iniciar los protocolos más típicos (e.g., `Request, Query`)

- Al crear una instancia le pasaremos a la constructora el mensaje (`ACLMessage`) que queremos enviar
  - **Iniciar el mensaje con el valor correcto de protocolo**

AchieveREInitiator (II)

- Podemos iniciarlo con un mensaje incompleto y terminarlo re-escribiendo la función `prepareRequests`
- En general podemos extender la clase y sobreescribir las funciones `handle…` que gestionan los diferentes estados del protocolo (`refuse, agree, response`)
- Si queremos enviar a varios responders, añadimos a todos en el mensaje
  - En este caso suele ser útil modificar la función `handleAllResponses, handleAllResultNotifications`
- En general una vez enviado el mensaje primero recibiremos la primera respuesta (`agree, not-understood, refuse`) llamando a `handleResponse`
- Y luego, si estaba de acuerdo, el resultado (`failure, inform`)

AchieveREResponder (III)

- En la constructora le pasaremos el template de mensaje sobre el que querremos responder
  - Normalmente indicaremos el protocolo y la performativa
  - Podemos usar también `createMessageTemplate`
- Para esta clase podremos reescribir las funciones `prepare`... para gestionar los diferentes estados (en especial, `prepareResponse` y `prepareResultNotification`)
  - Considerad utilizar la función `createReply` que se puede llamar mediante la `request` recibida, así evitamos errores

SimpleAchieveREInitiator/Responder

- Es una versión más compacta y rápida que el protocolo normal
- No permite registrar behaviours como *handlers*
- No permite tener más de un responder
- Si nuestro protocolo es 1:1 esta sería la mejor opción
- Podemos igualmente sobreescribir cualquier handler como en el caso anterior

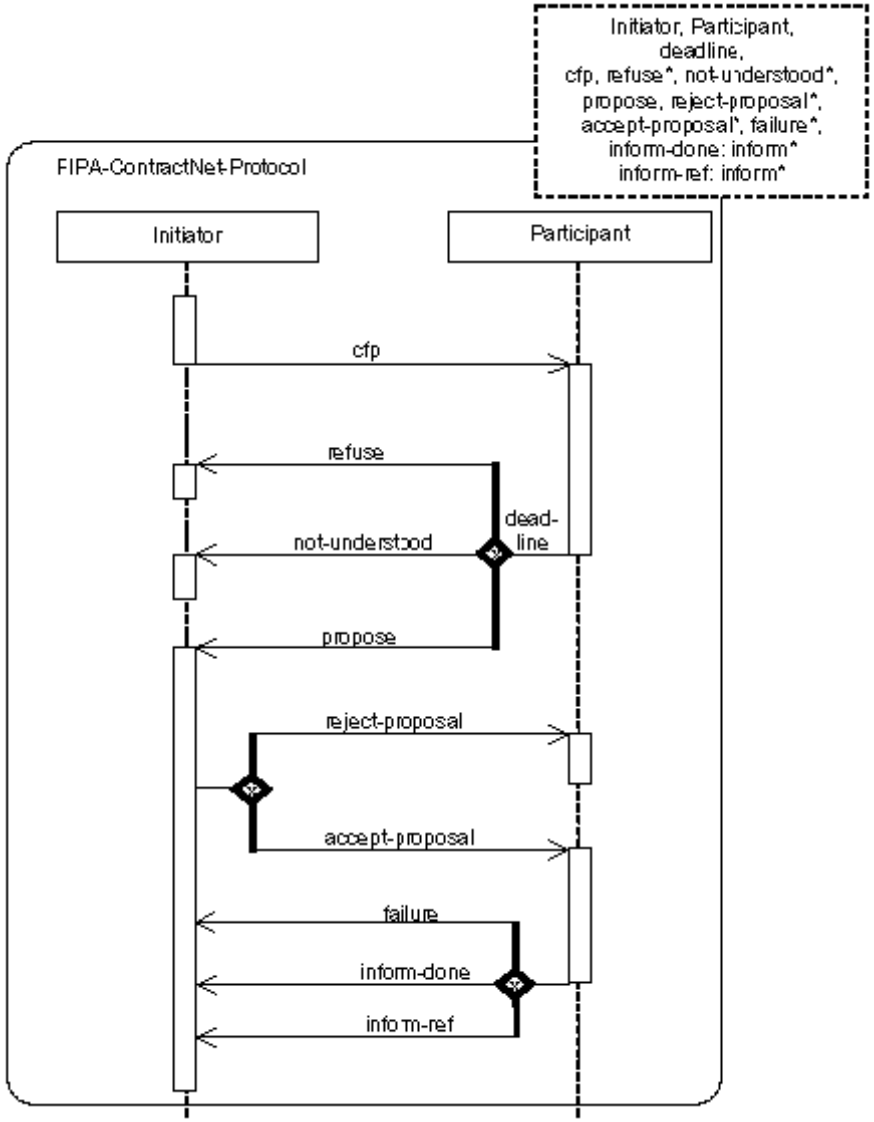
Ejemplo sencillo

```
ACLMessage request = new ACLMessage(ACLMessage.REQUEST)

request.setProtocol(FIPANames.InteractionProtocols.FIPA_REQUEST);
request.addReceiver(new AID("receiver", AID.ISLOCALNAME));
myAgent.addBehaviour( new AchieveREInitiator(myAgent, request) {
          protected void handleInform(ACLMessage inform) {
          System.out.println("Protocol finished. Rational Effect achieved.
          Received the following message: "+inform); }
});
```

```
MessageTemplate mt =
AchieveREResponder.createMessageTemplate(FIPANames.InteractionProtocols.FIPA_REQUEST);

myAgent.addBehaviour( new AchieveREResponder(myAgent, mt) {
 protected ACLMessage prepareResultNotification(ACLMessage requ, ACLMessage resp) {
          System.out.println("Responder has received the following message:"+request);
          ACLMessage informDone = request.createReply();
          informDone.setPerformative(ACLMessage.INFORM);
          informDone.setContent("inform done");
          return informDone; }
});
```

FIPA Contract-Net protocol

- `ContractNetInitiatorAgent`
- `ContractNetResponderAgent`
- Ejecuta los agentes
  - Consola
  - GUI sniffing
  - GUI introspector



Referencias para protocolos de interacción

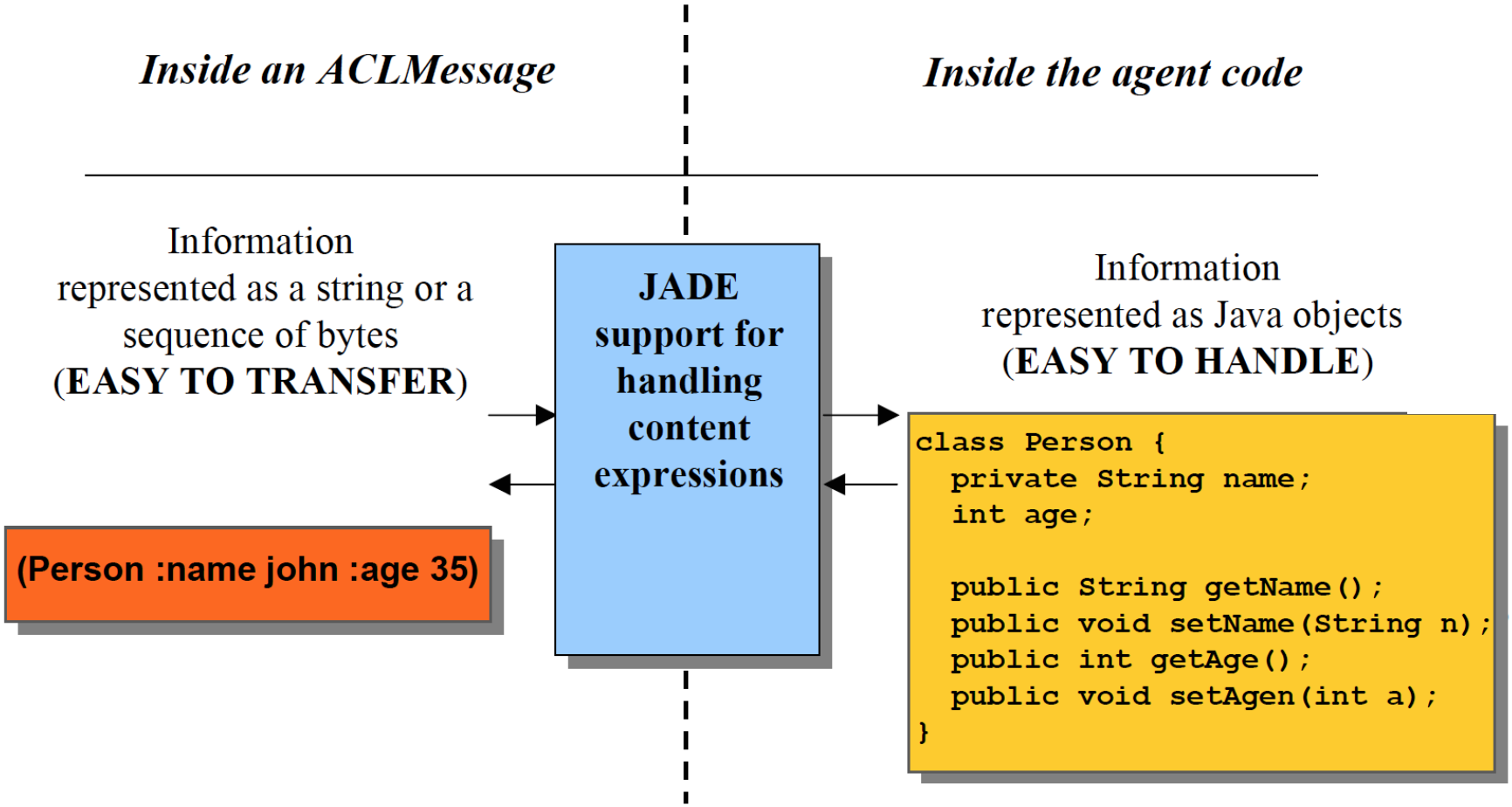
- Capítulo 3.5 de la guía de programación de JADE
- Documentación de la API (javadoc) de `jade.proto`
- Ejemplos de código en `examples.protocols` incluido en la distribución JADE

ACL Message

- Un ACL Message sigue una estructura fija:

```
(PERFORMATIVE
:sender (AgentAID)
:receiver (AgentAID)
:content ACL_CONTENT_EXPRESSION
:language [FIPA-SL0 | FIPA-LEAP | X]
:ontology ONTOLOGY_NAME
:protocol FIPA-XXX)
```

Gestionando el contenido



ACL: Content

- En este campo (*slot*) está el contenido del mensaje en forma de expresiones
- La librería de `commons-codec` se usa para codificar y decodificar estas expresiones que irán en dicho campo
  - En `jade.content.lang` encontramos SL y LEAP
  - Estas codifican las expresiones según el estándar especificado FIPA-SL y FIPA-LEAP
    - El primero es más legible
  - Podríamos especificar otros codecs (XML, RDF, OWL)

Content Manager

- JADE, a través del Content Manager
  - Permite la codificación/decodificación del contenido de los mensajes (usando los codecs)
  - Verifica que sean correctos (formatos, tipos) (usando la ontología)
  - Como para los agentes JADE la manipulación es más cómoda con objetos Java que con Strings, requiere que la ontología este en ese formato
  - Básicamente serializa en un formato y lo deserializa. Al hacer esto último crea las instancias al reificar
  - Podríamos hacer agentes que manipulan los mensajes de una manera más compleja (`AgentOWL`)

Ontología para Jade

- Necesitamos una ontología
  - Definición de elementos de la ontología
  - Definición de las clases Java correspondiente
- Podríamos hacernos la ontología mediante clases Java tal y como define JADE (es decir, siguiendo sus convenciones)
  - Mirad en referencias (al final de las slides) - Tutorial
- Para facilitarnos la vida podemos usar un plugin para Protégé que nos permitirá transformar una ontología en las clases Java para JADE
  - https://protegewiki.stanford.edu/wiki/OntologyBeanGenerator
  - Necesitaremos una versión antigua de Protégé (3.4)

Usando la ontología (I)

- Nos importaremos el código generado al proyecto
- Crearemos instancias usando las clases (tanto para conceptos como propiedades)
- En el `setup` del agente registraremos en este el lenguaje y ontología

```
getContentManager().registerLanguage(new SLCodec(), FIPANames.ContentLanguage.FIPA_SL0);

getContentManager().registerOntology(NameOfOntology.getInstance());
```

Usando la ontología (II)

- Crearemos el `ACLMessage` donde indicaremos la ontología

```
ACLMessage requestMsg = new ACLMessage(ACLMessage.REQUEST);
requestMsg.addReceiver(((RequesterAgent) myAgent).engager);
requestMsg.setLanguage(FIPANames.ContentLanguage.FIPA_SL0);
requestMsg.setOntology(EmploymentOntology.NAME);
```

- Mediante `myAgent.getContentManager.fillContent`
  - Añadiremos al mensaje creado las instancias

```
try {
  myAgent.getContentManager().fillContent(requestMsg, a);
} catch (Exception pe) {
}
```

- En el receptor, utilizaremos la función `getContentManager().extractContent(msg)`
  - Después podemos comprobar con `instanceOf` de que clase son las cosas (si hiciera falta)

Usando la ontología (III)

- En general utilizaríamos algún lenguaje de contenido como FIPA-SL para construir expresiones indicando si, por ejemplo, tenemos alguna cosa, si no tenemos esa cosa, si podemos descargar, etc…
  - Por ejemplo, para indicar que una persona no trabaja en X
    - `(not (Works (Person :name XXX) (Company :name YYY)))`
    - `AbsPredicate not = new AbsPredicate(SLVocabulary.NOT);`
      `not.set(SLVocabulary.NOT_WHAT,…);`

- Ojo, para las acciones (como descargar) tenemos que crear un objeto `Action` y después definir el actor y la acción concreta (`setActor` y `setAction`)
  - Ese será el contenido del mensaje

Referencias Content language y ontología

- Tutorial: APPLICATION-DEFINED CONTENT LANGUAGES AND ONTOLOGIES
  - http://jade.tilab.com/doc/tutorials/CLOntoSupport.pdf

Referencias JADE

- http://jade.tilab.com/documentation/tutorials-guides/
  - JADE Administrador's Guide
  - JADE Programming for Beginners
- http://jade.tilab.com/documentation/examples/

Knowledge Engineering and Machine Learning Group

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Luis Oliva Felipe

Ignasi Gómez

Luis Oliva

Sergio Álvarez

{loliva,igomez,ia,salvarez}@cs.upc.edu

SID2022