# Modelado de un Sistema Multi-Agente mediante la aplicación de la metodología INGENIAS con el Ingenias Development Kit

Juan A. Botía
Convocatoria 2008/2009
Sistemas Multi-Agente y Sistemas Autónomos
5º Curso, Ingeniería Informática
Departamento de Ingeniería de la Información y las Comunicaciones
Universidad de Murcia

## 1 Introducción

En la práctica de la asignatura vamos a desarrollar un software que simule, de una forma un tanto *naive*, un sistema de pedidos de pizzas por Internet, que actúe en régimen competitivo (i.e. las pizzerias compiten por vender más pizzas que el resto). Por supuesto, el sistema será distribuido y cada pizzería será una entidad independiente, representada por un agente. También cabe la posibilidad de que podamos tener intermediarios. Intermediarios que actuarán como pizzerías pero que en realidad no lo son. Simplemente se encargan de gestionar un directorio de pizzerias propio. Pizzerias que, esta vez si, están representadas por agentes.

Con respecto al usuario, este podrá obtener una lista de todas las pizzerias, obtener una lista de todas las pizzas dentro de una pizzería, preguntar por el precio de una pizza concreta y realizar un pedido de una pizza concreta a una pizzería.

El planteamiento de la práctica será el siguiente. En esta misma memoria que estás leyendo, se incluirá, explicado, un modelado inicial del SMA que cubre algunos de los requisitos planteados en el escenario anterior. Se incluirán algunos diagramas INGENIAS, que habremos producido con el IDK. Con todo esto será suficiente para generar un modelo de SMA que especifique suficientemente el sistema que queremos construir.

A partir de ahí, la labor del alumno será: (1) entender y asimilar el resto de esta memoria y (2) **expandir el modelo de SMA** que se incluye aquí, con los detalles a implementar que se explican en la sección 3.

Como herramientas para esta práctica usaremos únicamente el IDK como editor de modelos INGENIAS. Esta herramienta está disponible en (Ojo, versión 2.7!!!)

http://sourceforge.net/project/showfiles.php?group_id=67902

La metodología INGENIAS se compone, sobre todo, de un lenguaje de modelado. Los modelos producidos con este lenguage se estructuran en cinco modelos (i.e. view points) diferentes, que aparecen en la figura 1. El Organization viewpoint tiene en cuenta la estructura del sistema multi-agente, sus roles, las relaciones sociales y los flujos de trabajo que se dan dentro del mismo. El Agent viewpoint tiene en cuenta que los agentes realizan tareas y persiguen objetivos. Por tanto, se incluyen los agentes, sus tareas, los objetivos que persiguen y su estado mental (i.e. lo que conocen inicialmente). El Goals and Tasks viewpoint identifica objetivos y tareas a perseguir y realizar por los agentes, y los descompone. En el Interaction viewpoint se identifican las interacciones entre agentes y en el Environment viewpoint aquellas entidades con las que interaccionan los agentes y su relación con ellas.

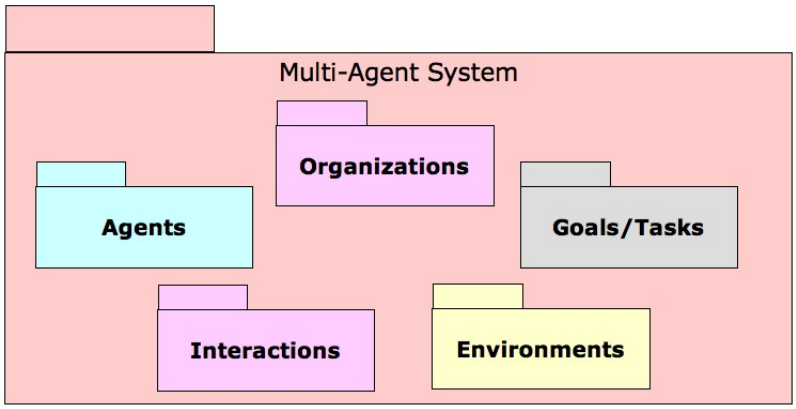


Figure 1: Los cinco puntos de vista de modelado de INGENIAS

# 2 El modelo inicial

En esta sección introducimos el modelo inicial del que se parte. La metodología INGENIAS es bastante flexible y nos permite comenzar el modelado por cualquier tipo de modelo, dependiendo de cuál sea el tópico más importante en nuestro caso, aunque esto no es obligatorio. También es posible seguir algún proceso de desarrollo, como RUP.

En este caso, lo más importante para nosotros van a ser las interacciones y, por tanto, vamos a comenzar el modelo desde aquí. De esta forma, comenzaremos generando un modelo de interacciones en el que incluiremos modelos de alto nivel de las mismas. A partir de aquí realizaremos modelos más específicos de las interacciones mediante la definición de protocolos. Al definir los protocolos detectaremos las tareas básicas que han de realizar los agentes, sus entradas y salidas. Con esta información generaremos un modelo de tareas y objetivos. Con las entradas y salidas de esas tareas generaremos una ontología. Tras esto, definiremos los agentes participantes en el modelo de agente. Después solamente nos restará definir una configuración de despliegue con la que lanzar el sistema. La secuencia de acciones del modelado viene representada esquemáticamente en la figura 2.

## 2.1 Modelo de interacciones

Una interacción es un proceso comunicativo entre entidades. En un SMA, pueden existir interacciones entre agentes y entre agentes y otros objetos del SMA. En el lenguaje de modelado INGENIAS solamente se tiene previsto el modelado explícito de interacciones entre agentes. El modelado entre agentes y otros objetos se realiza a través del modelo del entorno, de tal forma que los objetos se ven como aplicaciones.

Para modelar interacciones entre agentes, el proceso es siempre el mismo:

1. Crear un modelo de interacciones
2. Incluir interacciones dentro del modelo
3. Para cada interacción
   (a) Incluir roles iniciador y participante (posible indicar aridad en el participante)
   (b) Incluir el objetivo correspondiente para cada interacción
   (c) Incluir icono para especificación y
      i. Crear un nuevo modelo de interacción
      ii. Definir el protocolo de interacción en este nuevo modelo
      iii. Asociar el icono de la especificación a este nuevo modelo

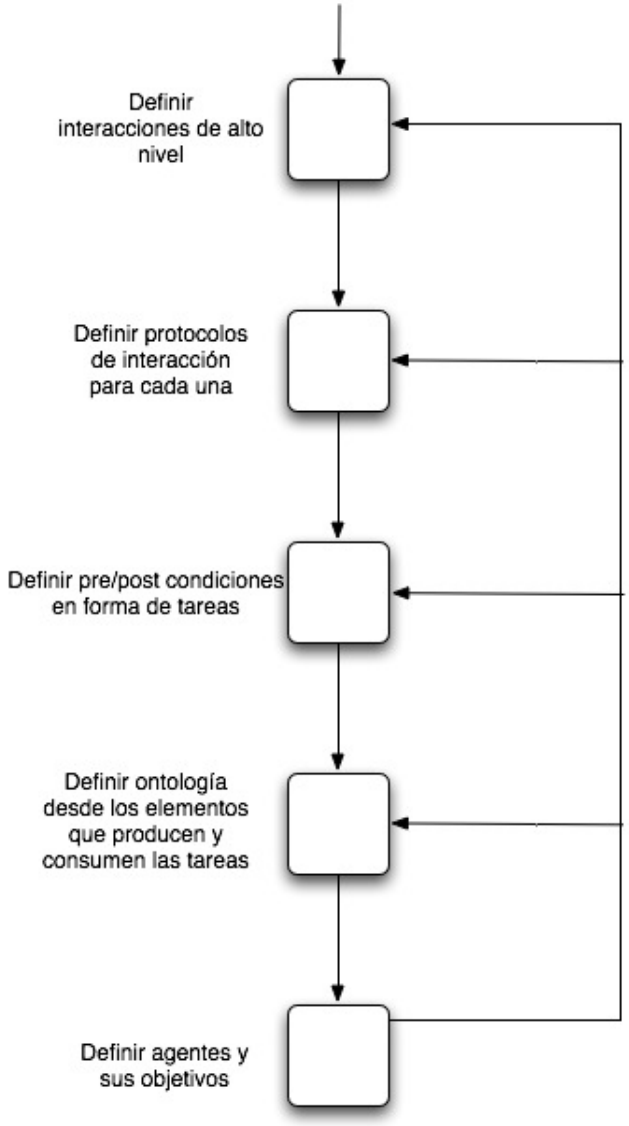


Figure 2: Pasos en el modelado INGENIAS para esta práctica, centrada en las interacciones

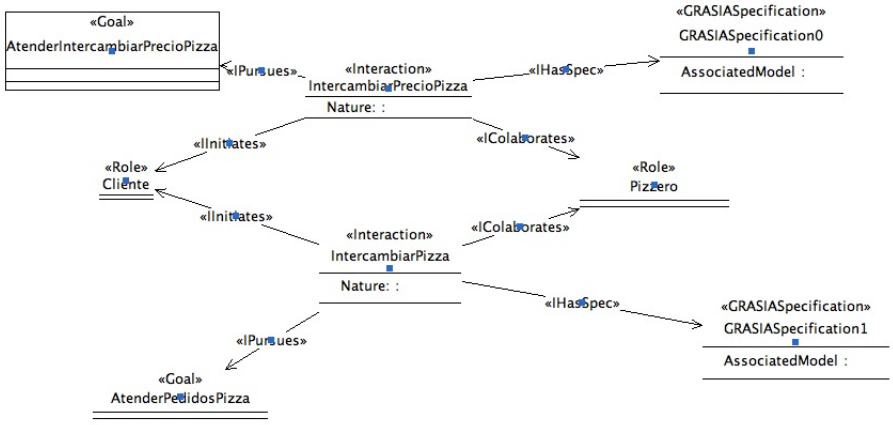


Figure 3: Diagrama general de las interacciones del sistema

en donde los pasos 3.(a), 3.(b) y 3.(c) se pueden aplicar en cualquier orden.

Inicialmente, vamos a tener dos interacciones diferentes. Por un lado, vamos a diseñar cómo un agente se hace con el precio de una pizza `IntercambiarPrecioPizza`, a partir de su nombre. Por el otro, se ha de pedir una pizza concreta a un agente determinado, lo especificaremos mediante la interacción `IntercambiarPizza`.

Por tanto, vamos a tener dos interacciones sencillas. Una de ellas se va a corresponder con el protocolo `fipa-query` y la otra con el `fipa-request`, respectivamente. Son interacciones en las que solo participan dos agentes por tanto, para ambas tendremos un iniciador y un participante (paso 3.(a)). Para ambas, iniciador y participante serán el mismo. Los denominaremos `Cliente` y `Pizzero`. Posteriormente, se debe indicar, para cada una de las interacciones, que tipo de diagrama vamos a usar para la especificación concreta de la interacción. Podemos utilizar varios tipos de diagramas entre los que tenemos AUML, que son diagramas de secuencia UML adaptados para agentes, y diagramas Grasiá, un tipo de diagrama más sencillo en cuanto a su notación y perteneciente a la metodología INGENIAS. Usaremos este último (paso 3.(c)). Lo mejor para especificar cada protocolo es definir un diagrama de interacción por separado para cada uno (ver sección 2.1.1). Por tanto, crearemos un nuevo diagrama de interacción por protocolo (paso 3.(c).ii). Así mismo, definiremos un objetivo que cada interacción persigue y con eso será suficiente por ahora (paso 3.(b)). Denominaremos a los objetivos con `AtenderIntercambiarPrecioPizza` y `AtenderIntercambiarPizza`. Una vez hemos indicado en el modelo de interacción del párrafo anterior que ambas interacciones tenían especificaciones Grasiá, hemos de definir ahora las interacciones de manera detallada. Para ello, como hemos dicho, creamos dos diagramas nuevos para estas dos nuevas interacciones (paso 3.(c).i). Sus nombres comenzarán con el prefijo `protocolo` y así los distinguiremos del resto. Definiremos ambos diagramas y cuando esto esté hecho, nos iremos al diagrama de interacciones original y, pinchando en el icono de cada especificación de interacción, asociaremos a la misma su protocolo correspondiente (paso 3.(c).iii).

### 2.1.1 Definición inicial de los protocolos de interacción

En el primer protocolo, vamos a incluir primero los dos roles que hemos indicado anteriormente iban a participar en la interacción. Para ello, en la ventana inferior izquierda aparece un panel de texto con el que podemos hacer búsquedas sobre el árbol de elementos que forman parte de nuestro modelo. Buscamos con `Cliente` y el árbol se despliega a la altura de los roles que tenemos definidos. Pinchamos en cliente y con el botón derecho del ratón indicamos que se añada al diagrama actual. Hacemos lo mismo con el rol `Pizzero`. Esta operación la haremos típicamente cada vez que en un modelo concreto necesitemos un elemento (e.g. un objetivo, tarea, agente, rol, frame fact, etc) que hemos definido previamente. Vemos como aparecen en el protocolo de la figura 4.

Ahora hay que incluir tres unidades de interacción. El protocolo va a consistir en una petición del

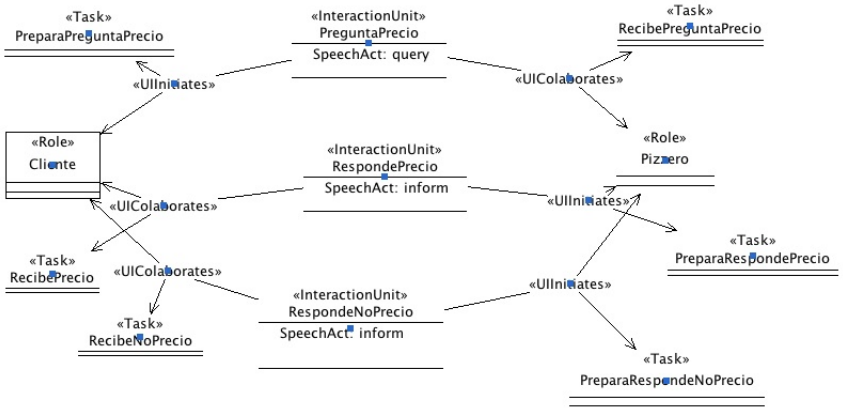


Figure 4: Protocolo de interacción (sin orden específico para las unidades de interacción) para pedir el precio de una pizza

precio de pizza y una respuesta que puede ser el precio o bien que la pizza no se conoce. Por tanto, incluiremos tres unidades de interacción, una por cada posible ocurrencia de envío de mensaje. Serán `PreguntaPrecio`, `RespondePrecio` y `RespondeNoPrecio`. Para la primera unidad de interacción, el iniciador (relación `IInitiates`) será el `Cliente` y `Pizzero` el participante (relación `ICollaborates`). Para las otras dos unidades de interacción, al contrario. Los objetos de tipo `Task` que aparecen en la figura 4 vienen explicados en la sección 2.1.2.

Abandonamos ese diagrama de interacción de especificación del protocolo para preguntar el precio de una pizza y creamos uno nuevo, con el mismo prefijo, para definir el protocolo de interacción para pedir una pizza. Este protocolo tendrá los mismos roles. Dado que también estamos realizando una petición, el diagrama de interacción será muy similar de tal forma que tendremos otras tres unidades de interacción.

### 2.1.2 Definición inicial del modelo de tareas

Con esta forma de modelar el sistema multi-agente, el elemento central del modelo es el conjunto de interacciones. Y a partir de este surge el resto de elementos. Ahora vamos a ver cómo surgen las tareas. Para ello, debemos fijarnos en cada una de las unidades de interacción de los protocolos que acabamos de definir, tomemos el de la figura 4 como ejemplo.

En la comunicación de agentes, el enviar un mensaje de un agente a otro se considera una acción de primer orden (i.e. tiene precondiciones y postcondiciones). En el contexto de la planificación convencional dentro de la Inteligencia Artificial, una acción dentro de un plan viene especificada por una precondición y una postcondición. Por tanto, una unidad de interacción (i.e. un mensaje a intercambiar entre agentes) también vendrá especificado igualmente.

Por un lado, para que una interacción se inicie, el objetivo que cumple debe estar en el estado mental del agente como pendiente de cumplirse. Por el otro, para que una interacción comience mediante el envío de la primera unidad de interacción (e.g. `PreguntaPrecio`), esta interacción debe tener disponible en el estado mental del agente aquellos hechos que necesita para empezar. Estos hechos son las precondiciones. Por tanto, para que una interacción entre dos o más agentes se inicie, (1) el agente iniciador debe querer hacerlo y (2) se deben dar las condiciones necesarias para que se haga.

A parte de las precondiciones, podemos especificar tareas que se han de ejecutar tras esas precondiciones (en el lado del agente que envía el mensaje) y antes de que el mensaje se haya enviado concretamente. Estas tareas pueden servirnos para generar el cuerpo del mensaje, por ejemplo. La tarea correspondiente a la precondición de una unidad de interacción determinada se especifica transformando la relación `UIInitiates` entre rol y unidad de interacción de binaria en una relación ternaria, incluyendo la tarea. Fijémonos, por tanto, en la tarea `PreparaPreguntaPrecio`, que se ejecutará en el agente con rol `Cliente`, inmediata-

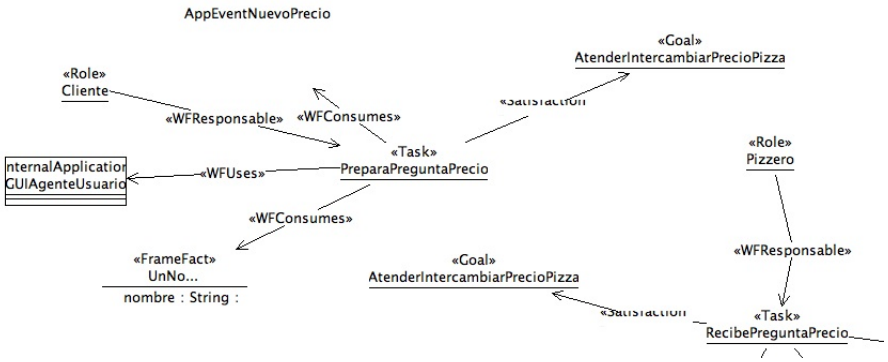


Figure 5: Especificación de una tarea con el lenguaje de modelado de INGENIAS

mente antes de ejecutar la unidad de interacción `PreguntaPrecio`. Otras son `PreparaRespondePrecio` y `PreparaRespondeNoPrecio`. El hecho de que para cada relación `UInitiates` aparezca en el modelo una asociación a una tarea no quiere decir que toda unidad de interacción necesite, antes de enviarse, que se ejecute una tarea. En nuestro caso, para este ejemplo concreto, sí va a ser así. Por tanto, asociamos un tarea a cada relación de este tipo.

Con respecto a las postcondiciones, se van a corresponder con los efectos de posibles tareas que podemos ejecutar en el lado del agente receptor, cuando el mensaje de la unidad de interacción correspondiente se ha recibido. Para indicar la tarea a ejecutar en el lado del agente receptor cuando se recibe un mensaje, se transforma la relación `UICollaborates` que relaciona el rol y la unidad de interacción de binaria a ternaria, como hacemos para las precondiciones.

Así, unidad de interacción a unidad de interacción, mirando a las pre y postcondiciones de cada una, generaremos las tareas necesarias al menos para la comunicación. Las creamos en el diagrama correspondiente al protocolo, por otro lado definimos un diagrama de tareas en donde especificamos cada tarea de manera más específica y finalmente definimos el código de cada una en un diagrama de tareas a parte.

Para definir una tarea determinada, necesitamos especificar para ella las entradas a la tarea (i.e. las precondiciones), las salidas o los hechos que la tarea produce, los objetivos que persigue, el rol responsable de la tarea (i.e. el que la ejecuta). Haremos eso para cada una de las tareas detectadas a partir de las interacciones. En la figura 5 vemos un ejemplo de la especificación de una tarea.

## 2.2 Definición inicial de la ontología

Es aconsejable, por claridad, el tener un modelo aparte en donde aparezcan todos los frame facts que se van a utilizar como entrada o salida de las tareas. Estos formarán la ontología básica de nuestro sistema. Los elementos de la ontología (i.e. los *frame facts*, f.f.) pueden verse como tokens de información a intercambiarse por los agentes. Por regla general, nada que no vaya a salir al exterior de la base de creencias de un agente necesita ser modelado. Es decir, solamente modelaremos como f.f. aquellos que deba ir en el cuerpo de los mensajes intercambiados por los agentes. Por tanto, todos los f.f. de la ontología surgen de forma natural a partir de la especificación de los protocolos de interacción, más específicamente a partir de la especificación de cada unidad de interacción.

## 2.3 Modelo de agentes

Una vez que tenemos los roles, tenemos también determinados los agentes. Ahora es necesario un modelo de agentes, que relaciones agentes con roles. Un agente de usuario `UserAgent` jugará el rol de `Cliente` y un `AgentePizzeria` jugará el rol de `Pizzero`. En este modelo también indicaremos los objetivos que sigue cada agente de tal forma que al intentar cumplirlos, se intentará lanzar las interacciones con las que hemos comenzado el diseño. Vemos así, la especificación del modelo de agentes en la figura 6.

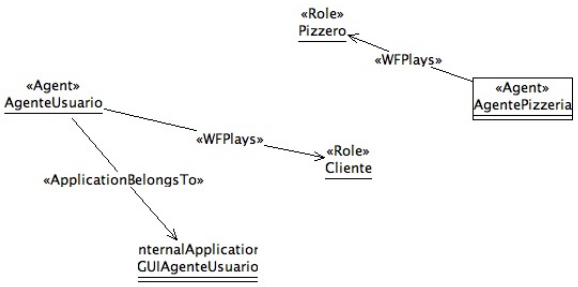


Figure 6: Especificación de los agentes con el lenguaje de modelado de INGENIAS

Obsérvese que este modelo es muy simple pero cada agente puede jugar más de un rol, y varios tipos de agentes pueden compartir roles.

### 2.4 Interacción de los agentes con el resto del mundo

Los agentes software suelen interaccionar con otros elementos del entorno. Es decir, un SMA no es un sistema cerrado en el que los agentes solamente interaccionan con otros agentes. Tendrán conexiones con el exterior. Estas conexiones con el exterior al final derivan en interacciones agente-objeto. Sin embargo, estas interacciones no se consideran comunicación al estilo de la que realizan los agentes. Se consideran interacciones más simples, basadas en el uso de APIs. Por ejemplo, un agente puede estar conectado con una aplicación mediante servicios Web, interfaces del tipo Corba, Java-RMI, etc.

En INGENIAS, todo lo que tiene que ver con software de aplicaciones se considera también en el modelado.

### 2.5 Depliegue del sistema multi-agente

El despliegue del sistema multi-agente es la parte del modelo en donde podemos especificar qué tipo de agentes queremos lanzar, cuántos de cada tipo y con qué valores iniciales, dentro de un límite. De esta forma, podemos tener varias configuraciones de despliegue y usar en cada momento la que más nos interese, dependiendo de las pruebas que necesitemos si estamos desarrollando o dependiendo de las configuraciones de SMA según el entorno y condiciones si hemos terminado el desarrollo.

## 3 Desarrollo de la práctica

En las primeras secciones hemos visto como usar el IDK para generar modelos de sistemas multi-agente (SMA) haciendo uso de la notación (i.e. lenguaje de modelado) INGENIAS. En esta segunda parte de la práctica se pide generar un SMA mínimo que cumpla efectivamente con lo que se pretende. Por tanto, el dominio de aplicación va a ser el mismo (i.e. un sistema de venta de pizzas con varios restaurantes implicados).

El modelado anterior, si no atendemos a detalles específicos de la generación de código, puede considerarse como correcto. Sin embargo, ahora tenemos que hacer ese modelo adecuado a la generación de código tal y como el IDK lo lleva a cabo. Por tanto, en esta sección vamos a detallar en la medida de lo posible, qué modificaciones va a sufrir el modelo para generar un SMA que nos permita preguntar precios de Pizzas a otros agentes.

Partimos de la base de que ahora, tendremos que modificar el modelo producido hasta ahora para adecuarlo a los requerimientos de la práctica, tal y como se detalla en las siguientes líneas.


Figure 7: La GUI de nuestro agente de usuario

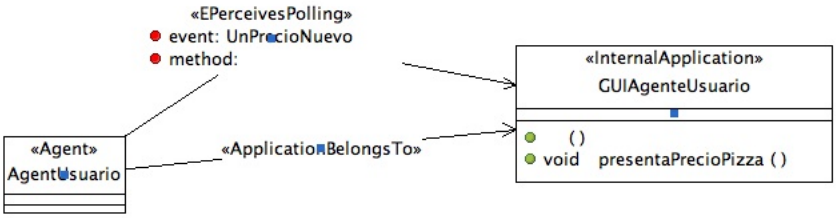


Figure 8: El agente de usuario y la aplicación

## 3.1 La aplicación Java

Cuando un usuario del SMA que vamos a generar desea preguntar el precio de una pizza, ¿cómo lo hace? ¿Cuál es el mecanismo que podemos usar para comunicar al usuario con el correspondiente agente de usuario que nos aparece en el modelo? Lo haremos a través de una GUI. Esta será muy sencilla y el alumno deberá ampliarla. La GUI básica tiene el aspecto que aparece en la figura 7. El código que genera esta ventana va a ser desarrollado por nosotros. Pero para ello, en el modelo tenemos que haber especificado antes que la aplicación con la que se va a conectar el agente está alojada en un fichero determinado. Así el IDK generará una vez el esqueleto de la aplicación y no lo modificará más. De esta manera, nosotros podemos modificarlo para incluir nuestro código.

Si nos fijamos en la ventana gráfica de la figura 7, lo único que ahí aparece es el nombre del agente con el que está relacionado la ventana y un botón. Ambas cosas las programaremos nostros. Las partes del modelo que están relacionadas con el GUI son varias. Vamos a verlas todas.

### 3.1.1 El agente y la aplicación

La relación del agente (de usuario en este caso), se especifica en el modelo de entorno que nosotros llamamos `entorno` en en IDK. En la figura 8 podemos apreciar la relación entre el agente y la aplicación. Hay dos relaciones entre la aplicación y el agente. La primera es del tipo `ApplicationBelongsTo` que indica que la aplicación es interna al agente (i.e. de su uso exclusivo). Esto va a permitir que desde el agente se tenga acceso a su código mediante la definición de una variable con una referencia a la correspondiente instancia de `GUIAgenteUsuario`. Así, una vez que la pizza (su precio) llegue a `AgenteUsuario`, este podrá llamar a su método

```
void presentaPrecioPizza()
```

para que la ventana muestre el precio que ha conseguido. La otra relación es del tipo `EPerceivesPolling` que significa que, de la ventana al agente también pueden llegar eventos y que se obtienen haciendo un polling[1]. De esta forma, cuando el usuario pulsa el botón de la figura 7, generará un evento que irá a parar al estado mental del agente. Una vez llegue este evento esto activará una tarea nueva que usamos para iniciar el sistema.

[1] en.wikipedia.org/wiki/Polling_(computer_science)

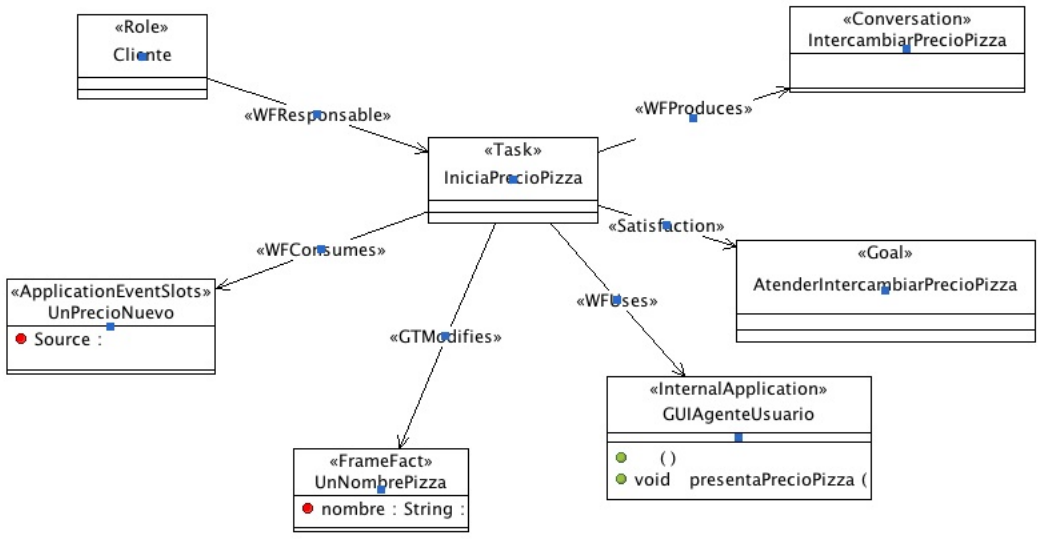


Figure 9: La tarea que inicia, a partir de un evento externo, la interacción para pedir el precio de una pizza

### 3.1.2 Las tareas y la aplicación

En el modelo de la práctica anterior no nos habíamos preocupado de qué mecanismo usar para lanzar la interacción encargada de preguntar el precio de la pizza. Sin embargo, es necesario, a la hora de generar código, el definir un mecanismo para esto. Por un lado, el evento que la GUI generará llegará al estado mental del agente y será consumido por una tarea. A su vez, esta tarea, activada al llegar el evento, iniciará la interacción que se encargará de preguntar por la pizza. Podemos verla en la figura 9. Aquí tenemos varios elementos nuevos que no hemos visto hasta ahora. El primero es un objeto de tipo conversación, `IntercambiarPrecioPizza`. El hecho de que se llame como la interacción no es casual. Cuando se crea un objeto de tipo `Conversation`, se debe asociar a una de las interacciones que previamente hemos definido. Este objeto hace referencia a una conversación concreta que responde a las características del protocolo definido en la interacción con que se asocia. Por tanto, la tarea `IniciaPrecioPizza` produce una instancia de conversación. Aquí tenemos el elemento que lanza la interacción. Otro detalle interesante es que ahora hay un f.f. `UnNombrePizza` que se une a la tarea, en lugar de con la relación `WFConsumes`, `GTModifies`. La razón para hacer esto es que el f.f. `UnNombrePizza` siempre debe estar en el estado mental del agente. Si hubieramos utilizado `WFConsumes`, el f.f. se habría consumido y no podríamos volver a lanzar una segunda interacción o posteriores. Fijaos que ahora la tarea efectivamente consume un evento (ya que se genera uno por cada pulsación del botón de la GUI). Esta tarea está ubicada ahora en un nuevo diagrama de tareas denominado `tareas`.

### 3.1.3 El código de la GUI

En este apartado se explica cómo se programa la ventana y la relación de ésta con el IDK. En la notación INGENIAS existen dos tipos de aplicaciones. Están las internas, que se crean a partir de que se crea el SMA o se ajustan para adaptarlo a sus propósitos. Las aplicaciones externas, en cambio, podemos verlas como aplicaciones heredadas que posiblemente ya estaban ahí antes que la necesidad de desarrollar un SMA. La nuestra es de las primeras. En la aplicación va a haber una parte de inicialización de la misma y otra parte en la que programamos el código de la ventana. En el IDK 2.6, la parte del modelo relativo a inicialización de aplicaciones ha de incluirse bajo una carpeta que se llame `system init`. De ahí que aparezca en nuestro nuevo modelo. Ahí podemos ver la especificación que aparece en la figura 10 y que en el modelo tiene la etiqueta `codigo aplicacion`. Ahí podemos ver que `GUIAgenteUsuario` está asociado con un `Componente`

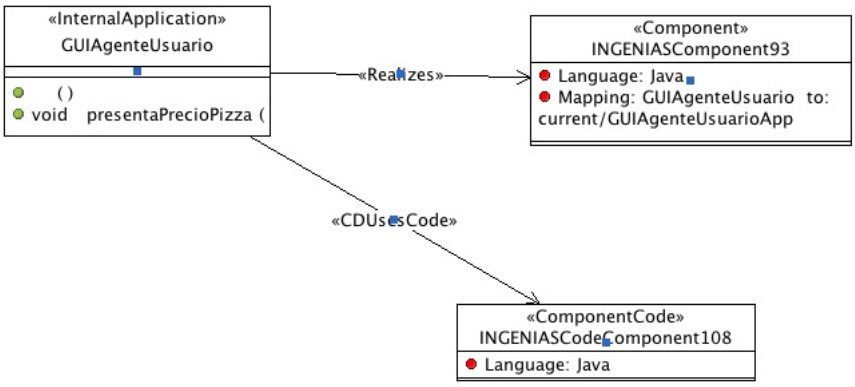


Figure 10: La aplicación interna y dos componentes para la GUI y para inicialización

y con un `ComponentCode`. El primero se diferencia del segundo en que todo él es una clase Java pública y por ello ocupa un fichero complejo (nótese que en el campo `Mapping` aparece el nombre de un fichero `GUIAgenteUsuarioApp`. Esto equivale a decir que el fichero en donde programaremos todo (cuyo esqueleto generará el IDK), se denomina `GUIAgenteUsuarioAppImp.java`. Se adjunta el fichero como apéndice.

Por otro lado, es necesario, además, incluir un código adicional de inicialización, representado en el modelo por el objeto de tipo `ComponentCode`. Si hacemos un doble click dentro del modelo en ese objeto, veremos que nos aparece una ventana con el siguiente código Java:

```
final GUIAgenteUsuarioAppImp appF = app;
new Thread(){
public void run(){
  appF.showGUI();
}}.start();
```

que básicamente realiza una llamada diferida, no bloqueante, (i.e. se realizará cuando el scheduler de la JVM pueda, para activar la ventana de la figura 7.

### 3.1.4 El código de las tareas

Cada una de las tareas que aparecen en el modelo tienen una parte de código Java en donde se incluye, sobre todo, el proceso de las entradas para generar las salidas. Se puede encontrar la parte del modelo relativa a este particular también en el paquete `system init` en un modelo con nombre `codigo tareas`.

La tarea `IniciaPrecioPizza` tiene el código

```
System.out.println("Estamos ejecutando tarea IniciaPrecioPizza");
MentalUtils.setSlotObjectValue(eiUnNombrePizza,"nombre","Margarita");
```

que como vemos, accede al método estático de la clase MentalUtils que actualiza el slot `nombre` de la variable eiUnNombrePizza, nombre generado automáticamente por el generador de código y que alberga un f.f. del tipo `UnNombrePizza`. Lo de `ei` hace referencia a entity input. Una entidad de entrada. Si nos fijamos en el modelo, en la definición de la tarea `IniciaPrecioPizza`, vemos que la relación de la tarea con el f.f. `UnNombrePizza` es del tipo `GTModifies` por tanto, la tarea modifica el f.f. pero sigue dentro del estado mental. Listo para entrar como input a la unidad de interacción que inicia la conversación que está relacionada con la misma tarea.

La tarea `RecibePreguntaPrecio` tiene el siguiente código

```
String nombre = MentalUtils.getSlotObjectValue(eiUnNombrePizza,"nombre").toString();
System.out.println("El agente recibe peticion de pizza con nombre " +
```

```
  nombre);
if(Math.random() < 0.5){
  int precio = (int)(Math.random() * 20);
  System.out.println("La pizza es conocida y el precio: " + precio);
  this.removeExpectedOutput("NoPizza");
  MentalUtils.setSlotObjectValue(eoUnPreciodePizza,"precio",precio);
  MentalUtils.setSlotObjectValue(eoUnPreciodePizza,"nombre",nombre);
}else{
  this.removeExpectedOutput("UnPreciodePizza");
  MentalUtils.setSlotObjectValue(eoNoPizza,"nombre",nombre);
  System.out.println("No se conoce la pizza");
}
```

Como no hemos implementado cuestiones relativas a qué agente tiene que pizzas en catálogo, introducimos un componente aleatorio que simula el hecho de que es posible el que no se conozca la pizza. Nótese que dependiendo de si es así o no, se elimina un hecho del estado mental del agente u otro. Si es conocida, eliminamos `NoPizza` con lo cual a la salida de la tareas solamente se generará `UnPreciodePizza` con lo que solamente se enviará la unidad de interacción que la lleva en el cuerpo del mensaje. Análogamente para el caso contrario.

La tarea `RecibeRespondePrecio` tiene el código

```
String nombre =
  MentalUtils.getSlotObjectValue(eiUnPreciodePizza,"nombre").toString();
int precio =
  Integer.parseInt(MentalUtils.getSlotObjectValue(eiUnPreciodePizza,"precio").toString());
System.out.println("Recibimos " + nombre +  ", " + precio);
eaGUIAgenteUsuario.presentaPrecioPizza(nombre,precio);
```

que precisamente es encarga de notificar al usuario del resultado afirmativo de la pregunta.

Por último, la tarea `RecibeRespondeNoPrecio` tiene el código

```
String nombre = MentalUtils.getSlotObjectValue(eiNoPizza,"nombre").toString();
System.out.println("Recibimos " + nombre +  ", desconocida");
eaGUIAgenteUsuario.presentaPrecioPizza(nombre, -1);
```

en el que se hace lo mismo para el caso de que la pizza no esté en el catálogo.

# 4 Material a entregar por el alumno para superar la práctica

Partiendo de este modelo, que deberá estar funcionando completamente, el grupo de prácticas deberá, para aprobar la asignatura, alcanzar los siguientes mínimos:

- ampliar el modelo con una interacción que permita realizar un pedido de pizzas (solamente Margarita, como en el ejemplo),
- entregar un documento, basado en el HTML generado por la herramienta, que explique el diseño (al nivel de detalle que se ha realizado en esta misma memoria) y
- entregar el fichero xml con la especificación del proyecto y cualquier fichero adicional que se deba incluir (e.g. un nuevo `GUIAgenteUsuarioAppImp.java`).

Para subir nota, se contemplará, a partir del uso del manual del IDK y ampliando los conocimientos en JADE

- que cada agente Pizzería tenga un catálogo de Pizzas,

- que el usuario pueda indicar el nombre de la pizza a consultar o pedir en el GUI y que el SMA lo use.

Se deberá tener en cuenta que se han de realizar las prácticas en grupos de 2 personas. Todas las entregas se habrán de realizar por correo-e a la dirección, `juanbot@um.es` hasta el 13 de enero inclusive.

### Código fuente de GUIAgenteUsuarioAppImp.java

```
package ingenias.jade.components;

import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.*;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import ingenias.editor.entities.ApplicationEventSlots;
import java.util.*;
import ingenias.jade.exception.*;

public class GUIAgenteUsuarioAppImp extends GUIAgenteUsuarioApp
{
  JFrame userGUI=new JFrame();
  JButton pidePrecio=new JButton("Pide el precio de pizza margarita");
  JLabel result=new JLabel("");
  public GUIAgenteUsuarioAppImp(){
    super();
  }
  public void showGUI(){
    Dimension dim=java.awt.Toolkit.getDefaultToolkit().getScreenSize();
    userGUI.setLocation(dim.width/2,dim.height/2);
    userGUI.setTitle(getOwner().getName() + " - Interface para Pizzerias");
    Box box=javax.swing.Box.createVerticalBox();
    userGUI.getContentPane().add(box);
    box.add(new JLabel("agent: "+this.getOwner().getLocalName()));
    box.add(pidePrecio);
    box.add(result);

    pidePrecio.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent arg0) {
      pidePrecio.setEnabled(false);
      result.setText("Buscando precio de pizza");
      ApplicationEventSlots nevent= new ApplicationEventSlots("UnPrecioNuevo");
      getOwner().getMSM().addMentalEntity(nevent);
    }
  });

    userGUI.setSize(new Dimension(400,100));
    userGUI.doLayout();
    userGUI.setVisible(true);
  }

  public void presentaPrecioPizza(String nombre, int precio){
  //TODO: INSERT HERE YOUR CODE
     pidePrecio.setEnabled(true);
     result.setText("El precio de la pizza " + nombre + " es " + precio);
     userGUI.doLayout();
     userGUI.setVisible(true);
  }
}
```