Generación de sistemas multi-agente en forma de código JADE a partir de INGENIAS-IDK

Juan A. Botía
MASTER TITA, Convocatoria 2007/2008
Ingeniería de Agentes Software y Físicos
Departamento de Ingeniería de la Información y las Comunicaciones
Universidad de Murcia

1 Introducción

En la primera práctica 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 práctica vamos a ver cómo generar un SMA mínimo a partir de una versión reducida del modelo que hemos producido en la primera. Por tanto, el dominio de aplicación va a ser el mismo (i.e. un sistema de venta de pizzas con varios restaurantes implicados) pero vamos a prescindir de:

- una de las dos interacciones: la interacción `IntercambiarPizza`
- y por tanto, prescindiremos del protocolo asociado a la interacción
- y a su vez de todas las tareas asociadas a dicho protocolo.
- Así mismo, dentro de la especificación del protocolo correspondiente a la interacción `IntercambarPrecioPizza` eliminaremos las tareas `PreparaPreguntaPrecio`, `PreparaRespondePrecio` y `PreparaRespondeNoPrecio`.

Por restricciones de tiempo vamos a limitar el problema de la generación de código con el IDK a una interacción. En todo caso, todo lo que digamos de la interacción a estudio, `IntercambiarPrecioPizza` nos valdrá igualmente para la construcción de una segunda `IntercambiarPrecioPizza`.

2 Desarrollo de la práctica

El modelado de la práctica 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, el fichero fuente que utilizamos es el que encontramos en la página Web de la asignatura como [URL]. Las diferencias principales con el anterior son, por un lado, que ya no aparece ni la interacción `IntercambiarPizza`, ni su protocolo de interacción ni tampoco las tareas asociadas a las unidades de interacción correspondientes. Por el otro, se ha completado algo más tanto la aplicación Java que se va a comunicar con el agente de usuario para lanzar un proceso de petición del precio de una pizza como el código Java específico de las tareas que aparecen en el protocolo de interacción `IntercambiarPrecioPizza`.

Figure 1: La GUI de nuestro agente de usuario

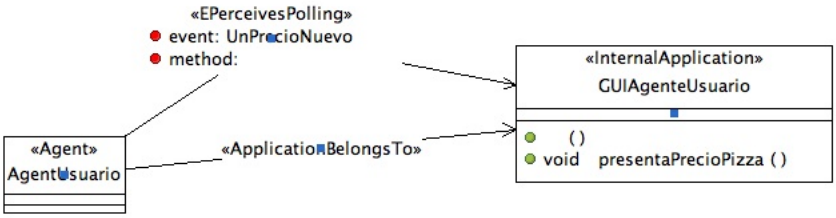

Figure 2: El agente de usuario y la aplicación

2.1 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 1. El código que genera esta ventana va a ser desarrollado por nostros. 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 1, 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.

2.1.1 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 2 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 1, 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)

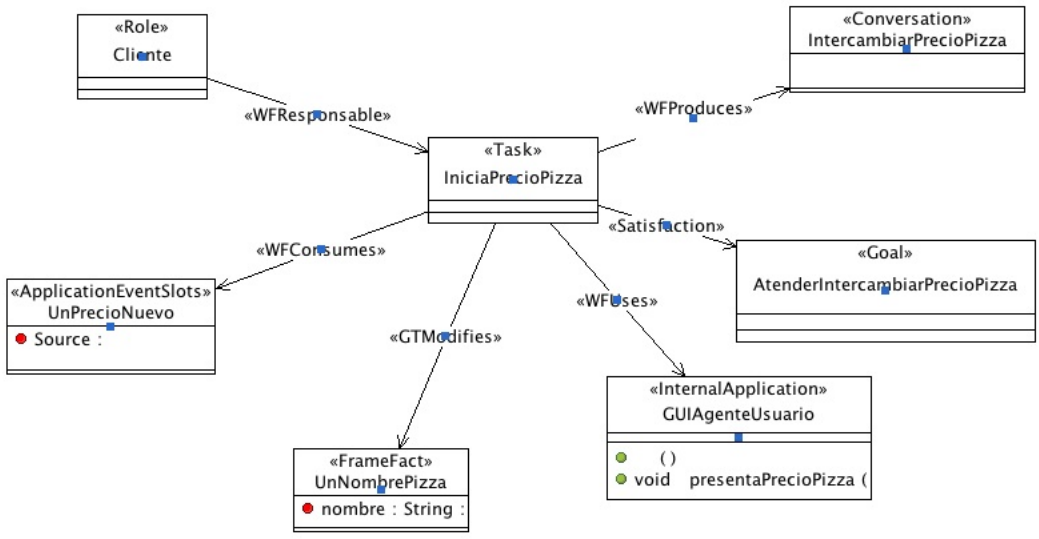

Figure 3: La tarea que inicia, a partir de un evento externo, la interacción para pedir el precio de una pizza

2.1.2 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 3. 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`.

2.1.3 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 4 y que en el modelo tiene la etiqueta `codigo aplicacion`. Ahí podemos ver que `GUIAgenteUsuario` está asociado con un `Componente`

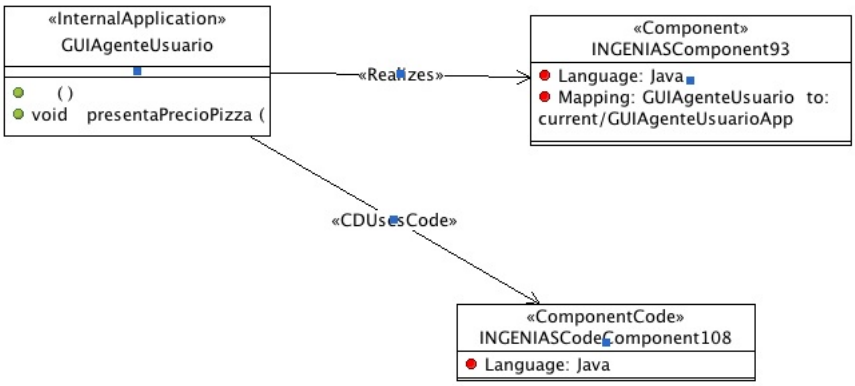

Figure 4: La aplicación interna y dos componentes para la GUI y para inicialización

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 1.

2.1.4 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.

3 Material a entregar por el alumno para superar la práctica

Partiendo de este modelo, que está 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, a lo sumo tres personas. Todas las entregas se habrán de realizar por correo-e a la dirección, `juanbot@um.es` hasta el 18 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);
  }
}
```