PROYECTO FINAL DE CARRERA

Sistema Multi-Agente para la construcción automática de ontologías

Laura Prieto Rebollo
Ingeniero Técnico en Informática de Sistemas

Directores de proyecto:
Dr. Antonio Moreno Ribas
Dra. Aïda Valls Mateu

Escola Técnica Superior d'Enginyeria (ETSE)
Universitat Rovira i Virgili (URV)

ÍNDICE

1- INTRODUCCIÓN

En la actualidad, Internet ofrece acceso a una cantidad enorme de información, lo que puede ser de gran utilidad en muchos casos. Sin embargo, esta información no está organizada de ningún modo, y por ello no se le puede sacar todo el beneficio posible.

Los buscadores web como *Google* o *Yahoo!* son una herramienta que facilita en gran medida la búsqueda de información a través de Internet, pero aún así, carecen de precisión y de acierto, ya que las búsquedas que realizan son sólo sintácticas, es decir, que simplemente buscan las páginas que contienen el concepto introducido por el usuario y las ordenan según una serie de criterios. Esto hace que de las páginas web que nos devuelven cuando realizamos una búsqueda, sean pocas las que se adaptan a lo que realmente necesitamos.

Por ejemplo, si queremos encontrar información sobre un concepto tan común como es 'virus', el buscador desconoce si lo que nos interesan son los virus informáticos o los virus médicos, y nos devolverá páginas web relacionadas con ambas acepciones.

Una solución posible a este problema es intentar estructurar, de alguna forma, el conocimiento existente en Internet sobre un dominio (Ej. cáncer) y clasificar las páginas web relacionadas con el dominio en diferentes categorías (Ej. cáncer de piel, cáncer de próstata).

1.1- FIPA

La FIPA (*Foundation for Intelligent Physical Agents*) es una fundación sin ánimo de lucro, con sede en Ginebra (Suiza). Su principal función es la de establecer una serie de reglas para el diseño e implementación de un sistema multi-agente, y así conseguir interoperabilidad entre sistemas. Hizo pública su primera especificación en el año 1997, y desde entonces se ha ido imponiendo hasta convertirse en el estándar seguido en todo el mundo.

Este proyecto se ha realizado con la ayuda de una herramienta de desarrollo de sistemas multi-agente que cumple las especificaciones de la FIPA, por lo tanto comentaremos ampliamente estas reglas de diseño e implementación.

1.2- Banzai - GruSMA

El grupo de investigación en Inteligencia Artificial Banzai es uno de los equipos de investigación del departamento de ingeniería informática y matemáticas (DEIM) de la Universidad Rovira i Virgili de Tarragona (España). Se creó en el año 1992 y está formado por profesores y estudiantes de Doctorado.

Banzai tiene como una de sus líneas de investigación el diseño de aplicaciones de Inteligencia Artificial, y dentro de este encontramos el grupo de trabajo GruSMA, creado por Aïda Valls y Toni Moreno, que se encarga de desarrollar diversos proyectos basados en sistemas multi-agente.

GruSMA utiliza la herramienta JADE para el desarrollo de sistemas multi-agente, la cual cumple con las especificaciones de la FIPA. Por lo tanto, además de hacer una introducción sobre cómo se desarrollan este tipo de sistemas, comentaremos también cómo se implementan utilizando esta plataforma.

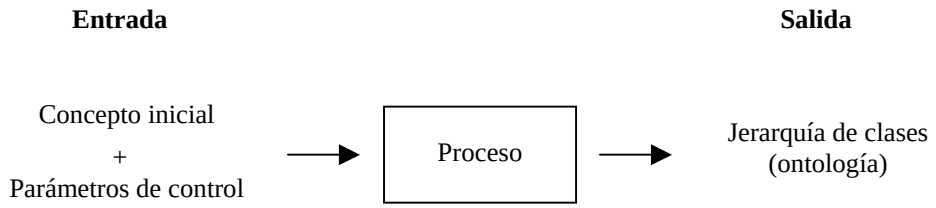
1.3- Objetivos del proyecto

El objetivo de este proyecto es diseñar un programa que utilice técnicas de Inteligencia Artificial y construya automáticamente una ontología de un cierto dominio. En este contexto, entenderemos como 'ontología' una jerarquía de clases en forma de árbol, donde cada clase corresponderá a un 'subdominio' del dominio inicial y tendrá asociado un conjunto de páginas web que contienen información potencialmente relevante para esa clase.

Para ello será preciso pedir al usuario que introduzca el nombre de la clase raíz de la ontología, junto con una serie de parámetros necesarios para el proceso. Cuando la creación de la ontología haya finalizado, se mostrará gráficamente el resultado, es decir, la estructura de la ontología en forma de árbol y las páginas web asociadas a cada una de las clases.

Además, la ontología se almacenará en un archivo para que pueda ser visualizada y reutilizada en un futuro. Un ejemplo de aplicación que podría tener la ontología creada es su reutilización por parte del proyecto europeo *h-techsight*, en el que colaboran los departamentos de Ingeniería Química e Ingeniería Informática de nuestra universidad. Su objetivo es desarrollar una herramienta que ayude a pequeñas y medianas empresas a acceder a la información publicada en la red. Concretamente, consiste en la creación de un buscador web que, basándose en ontologías y utilizando las especificaciones de dominios concretos definidos en ellas, pueda extraer las páginas web relacionadas con estos dominios, y en un futuro analizar esas páginas y extraer conocimiento que puede ser beneficioso para las empresas.

Por lo tanto, nuestro programa tendrá una serie de entradas y salidas, como muestra la figura:



1.4- Estructura del documento

En este documento explicaremos con detalle todo el proceso seguido para conseguir los objetivos propuestos. Dado que para entender correctamente el trabajo realizado se necesitan una serie de conocimientos básicos, dedicaremos los primeros capítulos a explicar todo aquello que consideramos imprescindible para entender el resto del documento.

La documentación se estructura de la siguiente forma: en el capítulo 2 se hace una introducción a los conceptos básicos sobre Agentes y Sistemas Multi-Agente. En el capítulo 3 se explica la herramienta de desarrollo y programación de Sistemas Multi-Agente utilizada (JADE). En el capítulo 4 explicaremos detalladamente un concepto muy importante en el proyecto, las ontologías, junto con lenguajes y herramientas relacionados. En el capítulo 5 hablaremos del resto de herramientas utilizadas en la realización del proyecto. En el capítulo 6 analizaremos el diseño del sistema. El capítulo 7 está dedicado a la explicación de las funciones y las principales características de cada uno de los módulos que forman la implementación del sistema. En el capítulo 8 se detallan las pruebas realizadas para la evaluación del sistema. En el capítulo 9 se explican las conclusiones sobre el proyecto y el trabajo futuro. En el capítulo 10 se comentan los recursos utilizados. El capítulo 11 está destinado a los manuales de usuario. Finalmente, en un apéndice encontramos todo el código fuente del sistema.

2- AGENTES Y SISTEMAS MULTI-AGENTE

Según la explicación ofrecida por el señor Wooldridge:

<< Un agente inteligente es un proceso computacional capaz de realizar tareas de forma autónoma y que se comunica con otros agentes para resolver problemas mediante cooperación, coordinación y negociación. Habitan en un entorno complejo y dinámico con el cual interaccionan en tiempo real para conseguir un conjunto de objetivos.>>

2.1- Propiedades de los Agentes

Las propiedades indispensables que debe tener un agente son:

- **Autonomía:** es la capacidad de operar sin la intervención directa de los humanos, y tener algún tipo de control sobre las propias acciones y el estado interno.

- **Sociabilidad / Cooperación:** los agentes han de ser capaces de interactuar con otros agentes a través de algún tipo de lenguaje de comunicación.

- **Reactividad:** los agentes perciben su entorno y responden en un tiempo razonable a los cambios detectados.

- **Pro-actividad o iniciativa:** deben ser capaces de mostrar que pueden tomar la iniciativa en ciertos momentos.

Otras propiedades destacables serían:

- **Movilidad:** posibilidad de moverse a otros entornos a través de una red.

- **Continuidad temporal:** los agentes están continuamente ejecutando procesos.

- **Veracidad:** un agente no comunicará información falsa premeditadamente.

- **Benevolencia:** es la propiedad que indica que un agente no tendrá conflictos, y que cada agente intentará hacer lo que se le pide.

- **Racionalidad:** el agente ha de actuar para conseguir su objetivo.

- **Aprendizaje:** mejoran su comportamiento con el tiempo.

- **Inteligencia:** usan técnicas de IA para resolver los problemas y conseguir sus objetivos.

2.2- Arquitectura de los agentes

Las arquitecturas utilizadas en el proceso de implementación de los agentes pueden ser de tres tipos:

- **Deliberativas:** dada una representación del mundo real, los agentes razonan según el modelo *BDI (Beliefs Desires and Intentions)*.

- **Reactivas:** estructura donde las acciones se llevan a cabo respondiendo a estímulos simples. La combinación de varios agentes reactivos proporciona un comportamiento inteligente.

- **Híbridas:** estructuras que mezclan diferentes tipos.

2.3- Tipos de Agentes

Existen diferentes tipos de agentes, podemos clasificarlos en:

- **Información:** gestionan y manipulan datos. Pueden responder a requisitos del usuario u otros agentes. Un ejemplo muy común son los buscadores de información en Internet.

- **Interfaz:** sistema dónde los agentes colaboran con el usuario para responder a un problema. La interacción con el individuo permite al agente desarrollar un aprendizaje basado en las acciones que se desarrollan.

- **Colaboración:** las características principales son la comunicación y cooperación con otros agentes para resolver un problema común. Utilizan técnicas de IA (Inteligencia Artificial) distribuida.

- **Móviles:** su característica principal es la posibilidad de poder moverse por una red electrónica recogiendo información o interactuando con otros *hosts*.

- **Reactivos:** su comportamiento se basa en la respuesta a estímulos según un patrón del estado actual en que se encuentra. Son muy simples, pero la combinación de diversos agentes reactivos puede generar comportamientos complejos.

- **Híbridos:** son combinaciones de diversos tipos anteriores intentando reunir las ventajas de unos y otros.

- **Heterogéneos:** se refiere a un conjunto integrado de al menos dos agentes que pertenecen a dos o más clases diferentes.

2.4- Sistemas Multi-Agente

Llamamos Sistema Multi-Agente o SMA a aquel en el que un conjunto de agentes cooperan, coordinan y se comunican para conseguir un objetivo común.

2.4.1- Ventajas de un sistema Multi-Agente

Las principales ventajas de la utilización de un sistema Multi-Agente son:

- **Modularidad**: se reduce la complejidad de trabajar con unidades más pequeñas, permiten una programación más estructurada

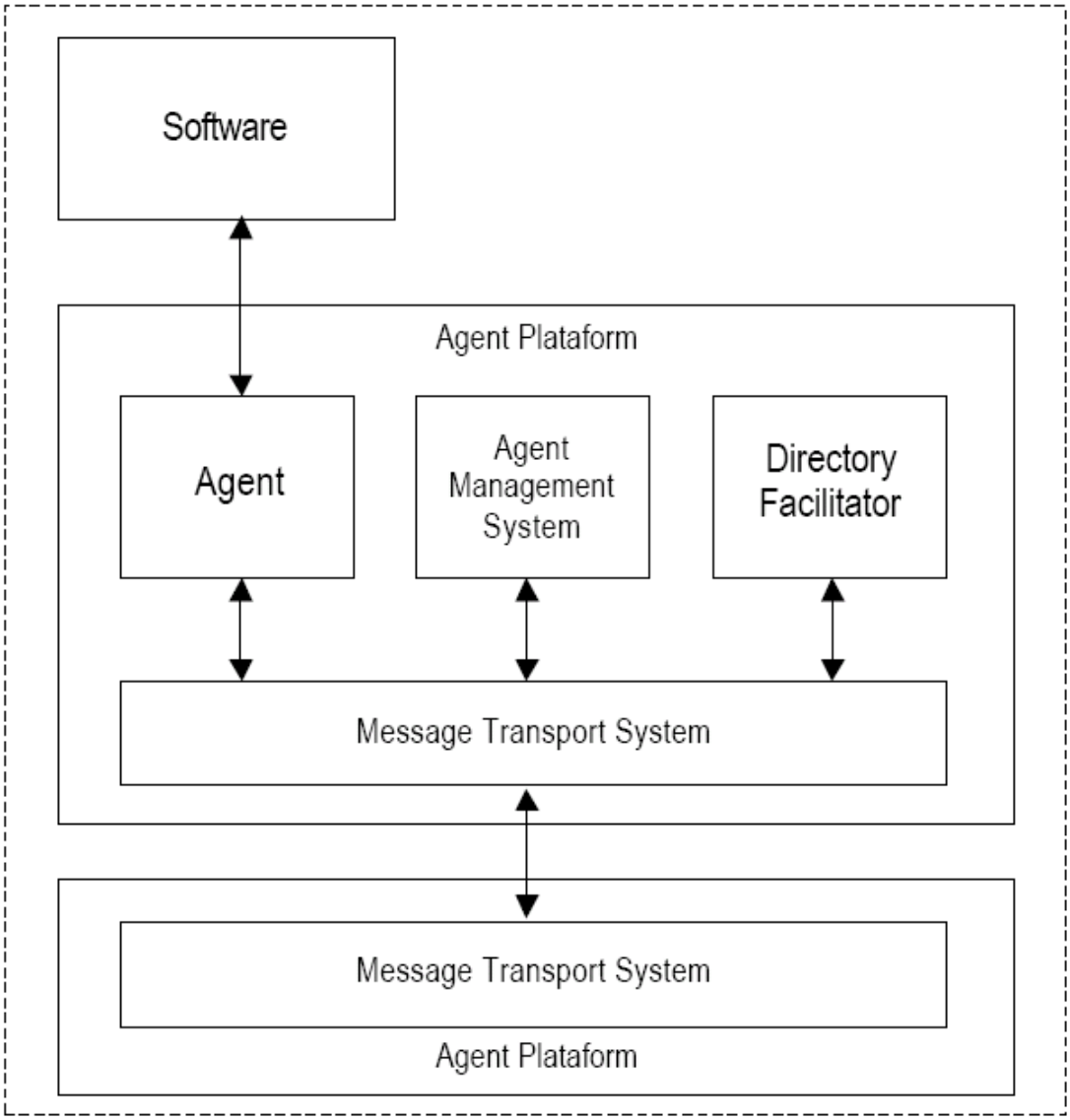
- **Eficiencia**: la programación distribuida permite repartir las tareas entre los agentes, consiguiendo paralelismo (agentes trabajando en diferentes máquinas).

- **Fiabilidad**: el hecho de que un elemento del sistema deje de funcionar no tiene que significar que el resto también lo hagan, además, se puede conseguir más seguridad replicando servicios críticos y así conseguir redundancia.

- **Flexibilidad**: se pueden añadir y eliminar agentes dinámicamente.

2.4.2- Gestión de un sistema Multi-Agente

La administración de agentes establece el modelo lógico para la creación, registro, comunicación, movilidad y destrucción de agentes. En este proyecto vamos a seguir los estándares establecidos por la *FIPA* (*Foundation for Intelligent Physical Agents*), organización que promueve el desarrollo de aplicaciones basadas en agentes. En la siguiente figura se puede ver el modelo para este entorno de administración de agentes:



Interior de un Sistema Multi-Agente

Los componentes que forman parte de la figura anterior son:

- **Agente**: unidad básica. Se podría definir como un programa que contiene y ofrece una serie de servicios.

- **Directory Facilitator (DF)**: agente que proporciona un servicio de *páginas amarillas* dentro del sistema, es decir, conoce los diferentes servicios que el resto de agentes de la plataforma ofrecen. Los agentes se han de registrar en el DF para ofrecer sus servicios.

- **Agent Management System (AMS)**: es el agente que controla el acceso y uso de la plataforma. Almacena las direcciones de los agentes, ofreciendo un servicio de *páginas blancas*.

- **Message Transport System (MTS)**: facilita la comunicación entre los agentes de diferentes plataformas.

- **Agent Platform (AP)**: proporciona la infraestructura básica para crear y ejecutar agentes.

- **Software**: cualquier programa accesible desde un agente.

2.4.3- Lenguaje de comunicación entre agentes (ACL)

Los agentes individualmente proporcionan una funcionalidad interesante, pero lo que los hace tan adecuados para ciertos sistemas es su capacidad de cooperar para resolver problemas. Para poder hacerlo, los agentes se tienen que de comunicar entre sí, utilizando un lenguaje común: ACL *(Agent Comunication Language)*. Para garantizar la homogeneidad y compatibilidad entre los diversos agentes, la *FIPA* determina que forma tiene que tener un mensaje (unidad básica de comunicación según la *FIPA*) y su utilización, para ello esta organización elabora las *FIPA Specifications* (especificaciones presentadas por la *FIPA*, que forman el estándar para la creación de agentes y SMAs).

2.4.3.1- Elementos de un mensaje

Todo mensaje estará compuesto por una serie de campos (o *slots*) que son los siguientes:

Slot	**Categoría que pertenece**
Performative	Definición del tipo de mensaje
Sender	Participante de la comunicación
Receiver	Participante de la comunicación
Reply-to	Participante de la comunicación
Content	Contenido
Language	Descripción del contenido
Encoding	Descripción del contenido
Ontology	Descripción del contenido
Protocol	Control de la conversación
Conversation-id	Control de la conversación
Reply-with	Control de la conversación
In-reply-to	Control de la conversación
Reply-by	Control de la conversación

- **Definición del tipo de mensaje**: indica que tipo de comunicación deseamos hacer.

Accept-proposal: aceptamos una propuesta hecha anteriormente para realizar una acción.
Agree: aceptación de una acción.
Cancel: cancelación de una acción.
Cfp: para proponer una acción.
Confirm: el emisor informa al receptor que una proposición es cierta, cuando el receptor dudaba.

Disconfirm: el emisor informa al receptor que una proposición es falsa, cuando el receptor pensaba que era cierta.
Failure: indica que la acción pedida anteriormente ha fallado por algún motivo.
Inform: el emisor informa al receptor que una acción se ha realizado correctamente.
Not-understood: el emisor informa al receptor que no ha entendido una petición realizada anteriormente.
Propose: acción de proponer la realización de una acción, dadas ciertas precondiciones.
Query-if: acción de preguntar a otro agente si un hecho es cierto o no.
Refuse: negación a la realización de una acción dada.
Request: el emisor pide al receptor la realización de una acción.
Subscribe: el emisor pide ser avisado en el momento que se cumpla una condición.

- **Participante de la comunicación**: son los identificadores de los agentes. Hay 3: **sender** (quien envía el mensaje), **receiver** (quien lo recibe), **reply-to** (a quien tiene que ir destinado el siguiente mensaje de la conversación).

- **Contenido**: existen cuatro tipos de contenidos predefinidos que se utilizan en función de las necesidades que se tengan:

FIPA-CCL *(Constrant Choice Language)*: define una semántica que permite especificar predicados con restricciones.
FIPA-SL *(Semantic Language)*: permite formar expresiones lógicas, intercambiar conocimientos de forma óptima y expresar acciones a realizar. Es el lenguaje más general de todos y puede ser aplicado a muchos dominios diferentes. Es el que hemos utilizado e nuestro SMA.
FIPA-KIF *(Knowledge Interchange Format)*: permite expresar objetos como términos y proposiciones como sentencias.
FIPA-RDF *(Resource Description Framework)*: permite expresar objetos, interacciones y acciones entre ellos.

- **Descripción del contenido**: permite que el agente que reciba el mensaje pueda identificar qué se le está enviando y en qué formato. Se definen 3 descriptores:

Language: en qué está escrito el contenido, nosotros hemos utilizado FIPA-SL.
Enconding: indica si se utiliza algún tipo de codificación especial.
Ontology: es el campo más importante, ya que, define la ontología utilizada para dar significado al contendido del mensaje.

- **Control de la conversación**: identifica el tipo de conversaciones que se mantienen. Sus campos son:

Protocolo utilizado en el mensaje.
Conversation-id: identificador de la conversación.
Reply-with: identificador del mensaje que se utiliza para seguir los diferentes pasos de la conversación.
In-reply-to: identifica una acción pasada a la cual este mensaje es la respuesta.
Reply-by: marca de *time-out*: fecha / hora de caducidad del mensaje.

2.4.3.2- Protocolos de comunicación

Como hemos indicado anteriormente en *control de la conversación*, existe un campo que identifica el protocolo utilizado. Dicho protocolo es el que define una serie de reglas o pasos que se ha de seguir para desarrollar una conversación. Existen diferentes tipos dependiendo de su finalidad:

- **FIPA-Request**: permite pedir la realización de acciones y devolver los resultados.
- **FIPA-Request-When**: variante del anterior (con condición de final).
- **FIPA-Query**: se utiliza para hacer preguntas.
- **FIPA-Contract-Net**: es un protocolo de negociación (se proponen y se evalúan propuestas).
- **FIPA-Iterated-Contract-Net**: variante del anterior que permite la renegociación.
- **FIPA Propose**: simplificación del *FIPA-Contract-Net*, permite gestionar propuestas.
- **FIPA-Brokering**: gestiona los servicios disponibles para los agentes.
- **FIPA-Recruiting**: variante del anterior.
- **FIPA-Subscribe**: permite la notificación de hechos importantes.

3- JADE

JADE (*Java Agent Development Enviroment*) es una herramienta de programación que contiene un conjunto de librerías escritas en Java para el desarrollo de Sistemas Multi-Agente. Además de proporcionar las funciones de manejo de agentes a bajo nivel y las interfaces que facilitan la programación, también nos presenta un entorno de ejecución donde los agentes pueden ejecutarse e interactuar.

La versión utilizada para la implementación de este proyecto es la 3.0b1.

3.1- Paquetes de JADE

JADE está compuesto por una serie de paquetes en Java. Los principales son los siguientes:

- **jade.core:** es el núcleo del sistema. Proporciona la clase *jade.core.Agent* que es imprescindible para la creación de SMA. También contiene el paquete *jade.core.behaviour* que incluye las clases de los comportamientos que tiene que tener todo agente.

- **jade.content:** contiene las clases específicas para soportar un lenguaje de comunicación entre agentes (ACL). En el se encuentran los paquetes *jade.content.lang.acl* y *jade.content.lang.sl*. También contiene todas las clases necesarias para la implementación de una ontología (*jade.content.onto*).

- **jade.domain:** contiene todas las clases para representar la plataforma de agentes y los modelos del dominio, tales como entidades para la administración de agentes, lenguajes y ontologías (AMS, DF, ACC, etc.).

- **jade.proto:** paquete que contiene los protocolos de interacción entre agentes FIPA.

- **jade.tools:** encontramos algunas herramientas que facilitan el desarrollo de aplicaciones y la administración de la plataforma:

  Remote Management Agent (RMA): es el agente que se encarga de la interfaz gráfica de JADE para la administración y control del sistema.

  Dummy Agent: es una herramienta de monitorización y depuración, compuesta por una interfaz gráfica y un agente JADE. Permite crear mensajes ACL y enviarlos a otros agentes pudiendo visualizarlos.

  Sniffer: agente que intercepta mensajes ACL y los visualiza, muy útil para el seguimiento de una conversación.

3.2- Componentes principales de JADE

Para JADE, los agentes residen en un entorno predefinido llamado plataforma. Cada plataforma está dividida en contenedores y cada uno de ellos contendrá agentes. Los contenedores podrían asemejarse al dominio de los agentes.

La plataforma se enciende en una máquina determinada, mientras que los agentes podrían ejecutarse en cualquier estación. Esto nos ofrece un comportamiento distribuido del SMA.

Tal como se define en la FIPA, en cada plataforma se inician dos agentes que tienen unas funciones vitales para el funcionamiento del sistema:

- **Domain Facilitator (DF)***:* podría definirse como un servicio de "páginas amarillas", es decir, contendrá todos los servicios que proporcionan los agentes que se han dado de alta en la plataforma. Será de especial utilidad a la hora de requerir algún servicio, puesto que nos informará de quien puede proporcionárnoslo.

- **Agent Management System (AMS)***:* se podría considerar como un servicio de "páginas blancas", donde se registran las direcciones de los agentes que se han dado de alta. Dado que JADE se fundamenta en entornos distribuidos, será básico para conocer la localización de los agentes.

Gracias a estos agentes, la comunicación en la plataforma es muy sencilla de realizar y permite una gran flexibilidad y transparencia. Por ejemplo, si un agente A quiere enviar un mensaje a un agente B que desarrolla la tarea T, el agente A preguntará al DF qué agentes pueden proporcionar dicho servicio T. Cuando se quiera conocer la dirección física de un agente determinado, preguntará al AMS. Para que este sistema funcione correctamente, será necesario que, durante el proceso de inicialización del agente, éste informe al DF de qué servicios dispone y el tipo de agente qué es (para poder identificarlo), al hacerlo, AMS le dará una dirección física (dependiendo de dónde se ejecute), para que otros se puedan poner en contacto con él.

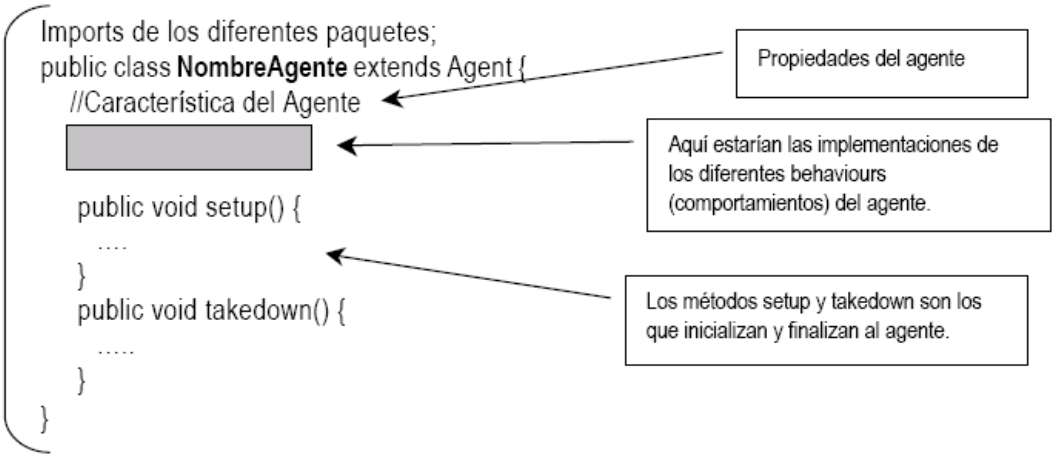
3.3- Agentes y comportamientos (Behaviours)

Para poder crear agentes en JADE, debemos extender la clase *jade.core.Agent*, que ya está definida, y rellenar los campos y las funciones básicas que nos proporciona la interfaz. Dos métodos muy importantes que se tienen que implementar son **setup()** y **takedown()** que inicializan y finalizan el agente respectivamente.

Como inciso de interés comentaremos la existencia de la clase *jade.gui.GuiAgent*, una extensión de la clase agente estándar, creada para facilitar la implementación de los agentes con GUI (*Graphic user interface*).

Una vez hayamos creado el agente debemos darle funcionalidad. Eso se consigue a través de la definición de los **comportamientos (behaviours)**, que son los que controlan sus acciones. Los *behaviours* son los encargados de darle dinamismo al agente y que reaccione a estímulos (mensajes) externos.

La siguiente figura muestra la estructura de un agente:



Ahora vamos a profundizar un poco más en el tema de los *behaviours*. JADE nos proporciona un conjunto de comportamientos básicos que permiten implementar diferentes protocolos de comunicación: **simples** y **compuestos**.

3.3.1- Comportamientos simples

Son los comportamientos que no pueden tener hijos. Estos comportamientos derivan de la clase *jade.core.behaviours.Behaviour*. Hay una serie de métodos que se pueden sobrecargar, como por ejemplo: **action()** (que engloba el conjunto de sentencias a ejecutar), **done()** (que devuelve un booleano que nos indica si el comportamiento ha finalizado) y el **reset()** (que reinicia el comportamiento). Hay tres clases de comportamientos simples:

- **SimpleBehaviour:** representa un comportamiento atómico que se ejecuta sin interrupciones. Podemos diferenciar entre:

  OneShotBehaviour*:* solo se ejecuta una vez.

  CyclicBehaviour: el comportamiento se ejecuta cíclicamente.

- **ReceiverBehaviour:** comportamiento que espera la recepción de un mensaje.

- **SenderBehaviour:** es equivalente al anterior pero desde el punto de vista de quien envía el mensaje.

3.3.2- Comportamientos compuestos

Son los comportamientos que pueden tener hijos, es decir, podemos asociar uno o más *subbehaviours* a un comportamiento complejo, que serán gestionados por un *scheduler* independiente del agente. Estos comportamientos derivan de la clase *jade.core.behviours.ComplexBehaviour*. Los métodos que se pueden sobrecargar son **onStart**() i **onEnd**(). El primero en ejecutarse es el *onStart*(), que es dónde añadimos

los subbehaviours. Cuando finaliza la ejecución de este método se lanzan todos los subbehaviours, y cuando todos hayan acabado se ejecuta el método *onEnd*().

Hay dos tipos de comportamientos complejos:

- **SequentialBehaviour**: ejecuta los hijos de manera secuencial.
- **ParallelBehaviour**: ejecuta los hijos según una política de *Round Robin*.

A continuación se muestra un ejemplo muy sencillo de la implementación y la utilización de un comportamiento simple (*OneShotBehaviour*):

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

public class AgentePrueba extends Agent {

    class PruebaBehaviour extends OneShotBehaviour {
        String texto;
        public EnviarPeticionBehaviour(Agent a,String texto) {
            super(a);
            this.texto = texto;
        }
        public void action() {
            System.out.println(texto);
        }
    }

    public void setup() {
        Behaviour b = new PruebaBehaviour(this, “Prueba simple”);
        addBehaviour(b));
    }
}
```

3.4- Mensajes

JADE también sigue las especificaciones de la FIPA en cuanto al formato de los mensajes, y nos proporciona la clase *jade.lang.acl.ACLMessage* para su creación y utilización. Los mensajes se dividen en *slots* (ver apartado 2.4.3.1) y sobre éstos, se definen funciones de consulta **(get)** y escritura **(set)**.

Los métodos básicos de qué disponemos para el uso de los mensajes entre agentes son:

- **send(ACLMessage)**: utilizado para enviar mensajes una vez se hayan rellenado todos sus parámetros.

- **ACLMessage receive()**: recibe un mensaje de la cola del agente. Se puede especificar qué tipo de mensaje se quiere recibir a través de los patrones de búsqueda *(MessageTemplate)*.

El lenguaje que utilizaremos en nuestro caso para escribir los mensajes será el FIPA-SL *(Semantic Language)*, que es el que tiene un ámbito más general. Permite formar expresiones lógicas, intercambiar información entre agentes y expresar acciones a realizar. La notación utilizada es parecida al *Lisp*: se define un conjunto de predicados genéricos y cada uno con una serie de atributos y parámetros.

3.5- Ontologías

JADE también nos permite crear ontologías. Éstas son muy importantes, ya que definen el vocabulario utilizado en nuestro SMA. Es por eso que antes de definir los agentes, hay que diseñar la ontología que se adapte a nuestras necesidades.

El elemento básico de una ontología es el **frame**. Éste a su vez se divide en **slots**, que pueden ser de diferentes tipos: **simples** (*integer*, *string*, *boolean*, etc.) o **otros frames**.

Para la creación de una ontología completa hay que tener 3 elementos en cuenta:

- **Conceptos o Objetos (frames)**: define los elementos básicos de la ontología.
- **Predicados**: define una serie de propiedades o características que pueden cumplir los objetos.
- **Acciones**: define las acciones que se podrán pedir a los agentes del sistema.

A continuación se muestran los pasos principales en la implementación de ontologías en JADE:

Fichero de especificación

- Paquetes a importar:

```
jade.content.onto.*
jade.content.schema.*
```

- Inicialización de la ontología:

```
private static Ontology theInstance = new OntologyName();
public static Ontology instance() {
    return theInstance;
}
private OntologyName () {
    super(NAME, BasicOntology.getInstance());
    ...
}
```

Especificación de los objetos

- Conceptos:

add(new ConceptSchema(NAME), Name.class);

- Predicados:

add(new PredicateSchema(NAME), Name.class);

- Acciones:

add(new AgentActionSchema(NAME), Name.class);

Definición de *slots*

- Conceptos:

```
ConceptSchema cs = (ConceptSchema)getSchema(NAME);
cs.add(SLOT_NAME,type,cardinality,optionality);
```

- Predicados:

```
PredicateSchema ps = (PredicateSchema)getSchema(NAME);
ps.add(SLOT_NAME,type,cardinality,optionality);
```

- Acciones:

```
AgentActionSchema as = (AgentActionSchema)getSchema(NAME);
as.add(SLOT_NAME,type,cardinality,optionality);
```

Implementación de clases

```
public class CONCEPT_NAME implements Concept { . . . }
public class PREDICATE_NAME implements Predicate { . . . }
public class ACTION_NAME implements AgentAction { . . . }
```

Para cada *slot* de la clase, tendremos que definir una propiedad y sus correspondientes métodos *set* y *get*.

3.6- Protocolos de comunicación

Una vez definida e implementada la ontología, debemos escoger el protocolo de comunicación entre agentes que más se adapte a nuestras necesidades.

El objetivo de los protocolos es establecer las normas de las transmisiones entre dos o más interlocutores.

Los protocolos definidos por la FIPA son: *Request, Query, Contract Net, Iterated Contract Net, Brokering, Recruiting, Subscribe, Propose.*

Los más utilizados e implementados en JADE son:

- **Request**: para pedir a otro agente que realice una determinada acción.
- **Query**: para preguntar a un agente información sobre algún hecho.
- **Contract Net**: permite realizar negociaciones entre dos o más agentes.

Los protocolos son implementados pos comportamientos (*behaviours*).

Para cada conversación, JADE distingue dos bandos:

- El que inicia la conversación: **initiator**.
- El que la sigue: **responder**.

Y las clases disponibles son:

- **AchiveREInitiator:** reemplaza a la clase FIPAXXXInitiatorBehaviour (dónde XXX es el nombre del protocolo) que ofrece la funcionalidad de *initiator* de la comunicación.

- **AchiveREResponder:** reemplaza a la clase FIPAXXXResponderBehaviour (dónde XXX es el nombre del protocolo) que ofrece la funcionalidad de *responder* de la comunicación.

3.7- Protocolo utilizado: FIPA-Request

El único protocolo utilizado en este proyecto es el protocolo *FIPA-Request*, que explicamos a continuación con más detalle.

Se utiliza para pedir a otro agente que realice una acción. El comportamiento que tendrán los agentes que utilicen este protocolo será el siguiente:

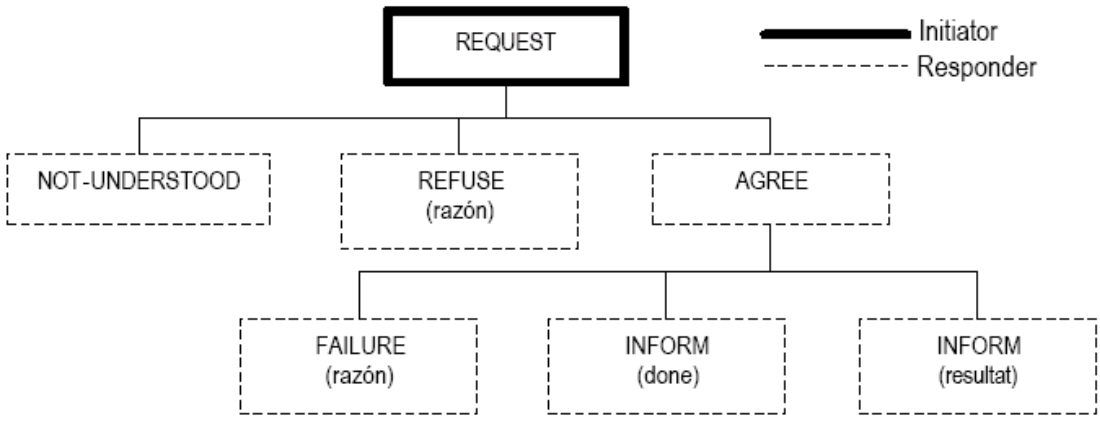
- El agente iniciador envía un mensaje de tipo *Request* al agente receptor para que realice una acción determinada.

- El receptor del mensaje lee el contenido y extrae la acción. Si el iniciador no ha formulado bien el mensaje y, por lo tanto, no lo puede descifrar, envía un mensaje de tipo *Not-understood*.

- Si el receptor entiende el mensaje, estudia la petición y determina si la puede realizar o no. En caso afirmativo, envía un mensaje de tipo *Agree* al iniciador y empieza a realizar la acción. Si por alguna razón no puede realizarla, envía un mensaje de tipo *Refuse*.

- En el caso de haber enviado un mensaje de tipo *Agree*, cuando el receptor acaba de realizar la acción envía un mensaje de tipo *Inform*. Este mensaje puede ser de dos tipos: *Inform-done*, que sólo notifica que la acción ha finalizado, e *Inform-ref*, que devuelve un resultado.

- Si la acción no ha finalizado correctamente, se enviará un mensaje de tipo *Failure* indicando la razón.



Protocolo FIPA-Request

A continuación se muestra la estructura de los métodos que se tienen que implementar para la utilización del protocolo FIPA-Request.

- Initiator:

```
Class RequestInitiator extends [Simple]AchieveREInitiator {

   public RequestInitiator (Agent myAgent, ACLMessage requestMsg) {
       Super(myAgent, requestMsg);
   }
   protected void handleAgree(ACLMessage msg) { … }
   protected void handleInform(ACLMessage msg) { … }
   protected void handleNotUnderstood(ACLMessage msg) { … }
   protected void handleFailure(ACLMessage msg) { … }
   protected void handleRefuse(ACLMessage msg) { … }
}
```

- Responder:

```
Class RequestResponder extends [Simple]AchieveREResponder {

   public RequestResponder (Agent myAgent, MessageTemplate mt) {
       Super(myAgent, mt);
   }
   protected ACLMessage prepareResponse(ACLMessage request) { … }
   protected ACLMessage prepareResultNotification (ACLMessage request,
                                                   ACLMessage response) { … }
}
```

3.8- Herramientas de JADE

JADE ofrece una interfaz gráfica para la administración de la plataforma, así como herramientas para facilitar la depuración y comprobación de las aplicaciones.

Para iniciar una sesión y crear los agentes, deberemos escribir la siguiente línea de comandos:
java JADE.Boot [options] –platform [lista de agentes]

En el campo de opciones podemos especificar que deseamos visualizar la interfaz gráfica de JADE utilizando el parámetro *–gui*.

En la lista de agentes estos estarán separados por espacios en blanco y tendrán el siguiente formato:
<NombreAgente>:ClaseAgenteJava

Si tenemos las clases dentro de un *package* debemos especificarlo:
<NombreAgente>:PackageClase.ClaseAgenteJava

De todas formas, existe la posibilidad de iniciar el entorno gráfico de JADE sin ningún agente y posteriormente ir cargando los agentes que se desee con las herramientas que ofrece.
Cuando se inicia el entorno gráfico, aunque no se inicialice ningún agente, se crean algunos automáticamente. Son los agentes básicos de la plataforma, encargados de que todo funcione correctamente (se comentaron en los puntos 3.1 y 3.2).

Disponemos de una serie de opciones que nos facilitarán el manejo y testeo de los agentes:
- Crear un nuevo agente: deberemos indicar el nombre, la clase que lo implementa
- Eliminar los agentes seleccionados
- Ejecutar los agentes seleccionados
- Paralizar los agentes seleccionados
- Continuar los agentes que estaban paralizados
- Enviar un mensaje a los agentes seleccionados

4- ONTOLOGÍAS

4.1- Bases teóricas de las ontologías

4.1.1- Introducción

Las ontologías son muy utilizadas en la Ingeniería de Conocimiento, Inteligencia Artificial y Ciencia Informática, en aplicaciones relacionadas con el manejo de conocimiento, procesamiento del lenguaje natural, e-comercio, integración inteligente de información, recuperación de información, diseño e integración de bases de datos, bio-informática, educación, y en nuevos campos emergentes como *the Semantic Web*.

El conocimiento declarativo es modelado por medio de ontologías mientras que los métodos de resolución de problemas precisan mecanismos de razonamiento genérico. Ambos tipos de componentes se pueden considerar como entidades complementarias que pueden ser utilizadas para configurar sistemas basados en el conocimiento a partir de componentes reutilizables existentes.

Las ontologías y los métodos de resolución de problemas han sido creados para compartir y reutilizar el conocimiento y el comportamiento del razonamiento a través de dominios y tareas. Las ontologías tratan el conocimiento de dominio estático, mientras que los métodos de resolución de problemas se ocupan de modelar procesos de razonamiento. Un método de resolución de problemas define (Benjamins y Gómez-Pérez, 1999) un camino para alcanzar el objetivo de una tarea. Tiene entradas y salidas, y puede descomponer una tarea en subtareas, y tareas en métodos. Además, un método de resolución de problemas especifica el flujo de datos entre sus subtareas. Un componente importante de los métodos de resolución de problemas es su ontología de método porque describe los conceptos utilizados por el método en el proceso de razonamiento, así como las relaciones entre tales conceptos.

La aparición de *the Semantic Web* ha marcado otra etapa en la evolución de las ontologías (y de los métodos de resolución de problemas). Según Berners-Lee (1999), *the Semantic Web* es una extensión de la actual *Web* a cuya información se le da un significado bien definido, permitiendo que los ordenadores y las personas trabajen mejor en cooperación. Esta cooperación puede ser alcanzada utilizando componentes de conocimiento compartidos, y así las ontologías y los métodos de resolución de problemas se han convertido en instrumentos clave en el desarrollo de *the Semantic Web*. Las ontologías representan conocimiento de dominio estático y los métodos de resolución de problemas serán utilizados dentro de los Servicios de *the Semantic Web*, que modelan procesos de razonamiento y tratan con ese conocimiento de dominio.

4.1.2- ¿Qué es una ontología?

La palabra ‘ontología’ fue tomada de la Filosofía, donde se refiere a una explicación sistemática de la existencia. En la última década, esta palabra ha llegado a ser relevante para la comunidad de la Ingeniería del Conocimiento. (Nicola) Guarino y Giaretta (1995) proponen usar las palabras ‘Ontology’ (con ‘o’ mayúscula) y ‘ontology’ para referirse al sentido filosófico y al de la Ingeniería del Conocimiento respectivamente. Existen muchas definiciones sobre lo que es una ontología, que han cambiado y

evolucionado a lo largo de los años. A continuación veremos estas definiciones y explicaremos las relaciones entre ellas.

Una de las primeras definiciones la dio Neches y compañeros (1991), quienes definieron una ontología de la siguiente manera:

<< Una ontología define los términos básicos y relaciones que constituyen el vocabulario del área de un tema, así como las reglas para combinar términos y relaciones para definir extensiones del vocabulario. >>

Esta descriptiva definición nos dice qué hacer para construir una ontología, y nos da algunas líneas a seguir: esta definición identifica términos básicos y relaciones entre términos, identifica reglas para combinar términos, y proporciona las definiciones de tales términos y relaciones. Nótese que, de acuerdo con la definición de Neches, una ontología no sólo incluye los términos explícitamente definidos en ella, sino también el conocimiento que puede ser deducido de ello.

Pocos años después, Gruber (1993) definió una ontología así:

<< Una ontología es una especificación explícita de una conceptualización. >>

Esta definición llegó a ser la más citada en literatura y por la comunidad de ontologías. Basadas en la definición de Guber, fueron propuestas muchas definiciones de lo que es una ontología. Borst (1997) modificó ligeramente la definición de Guber de esta manera:

<< Las ontologías se definen como una especificación formal de una conceptualización compartida. >>

Las definiciones de Guber y Borst han sido combinadas y explicadas por Studer y compañeros (1998) como sigue:

<< Una ontología es una especificación formal y explícita de una conceptualización compartida. ‘Conceptualización’ se refiere a un modelo abstracto de algún fenómeno en el mundo, habiendo identificado los conceptos relevantes de este fenómeno. ‘Explícita’ significa que el tipo de conceptos usados y las restricciones en su uso están explícitamente definidas. ‘Formal’ se refiere al hecho de que la ontología debería ser legible a máquina. ‘Compartida’ refleja la idea de que una ontología captura conocimiento consensual, es decir, que no es privado para algún individual, sino aceptado por un grupo. >>

En 1995 Guarino y Giaretta (1995) colectaron y analizaron las siete definiciones siguientes:

1- Ontología como una disciplina filosófica.
2- Ontología como un sistema conceptual informal.
3- Ontología como una relación semántica formal.
4- Ontología como una especificación de una conceptualización.
5- Ontología como una representación de un sistema conceptual vía una teoría lógica
   5.1- caracterizada por propiedades formales específicas.

5.2- caracterizada sólo por sus propósitos específicos.
6- Ontología como el vocabulario usado por una teoría lógica.
7- Ontología como una especificación de una teoría lógica.

Guarino y Giaretta propusieron considerar una ontología como:

<< Una teoría lógica que da una relación parcial explícita de una conceptualización. >>

donde una conceptualización es básicamente el concepto del mundo que una persona o un grupo de personas puede tener. Aunque en apariencia la idea de conceptualización es bastante similar a la idea de Studer y compañeros (1998), podemos decir que Guarino y Giaretta (1995) fueron más allá porque formalizaron la idea de conceptualización y establecieron como construir la ontología haciendo una teoría lógica. Por lo tanto, estrictamente hablando, esta definición sólo sería aplicable a ontologías desarrolladas en lógica. El trabajo de Guarino y Giaretta ha sido refinado (Guarino, 1998) y proporcionan la siguiente definición:

<< Un conjunto de axiomas lógicos diseñados para explicar el significado deseado de un vocabulario. >>

Hay otro grupo de definiciones basadas en el proceso seguido para construir la ontología. Estas definiciones también incluyen algunos puntos culminantes sobre la relación entre ontologías y bases del conocimiento. Por ejemplo, la definición dada por Bernaras y compañeros (1996) en el marco del proyecto KACTUS (Schreiber et al., 1995) es:

<<Una ontología proporciona los medios para describir explícitamente la conceptualización detrás del conocimiento representado en una base de conocimiento.>>

Nótese que esta definición propone ‘extraer’ la ontología de una base de conocimiento, lo que refleja la aproximación que utilizan los autores para construir ontologías. En esta aproximación, la ontología es construida siguiendo una estrategia *bottom-up*, a partir de una base de aplicación de conocimiento y por medio de un proceso de abstracción. Cuantas más aplicaciones se construyen, la ontología se vuelve más general y, por lo tanto, va más lejos de lo que sería una base de conocimiento.

Otra estrategia para construir ontologías es reutilizar grandes ontologías como SENSUS (Swartout et al., 1997) (con más de 70.000 nodos) para crear ontologías de dominio específico y bases de conocimiento:

<< Una ontología es un conjunto de términos estructurado jerárquicamente para describir un dominio que puede ser utilizado como el esqueleto de una base de conocimiento. >>

De acuerdo con esta definición, una misma ontología se puede utilizar para construir varias bases de conocimiento, las cuales deben compartir el mismo esqueleto o taxonomía. Debería ser posible realizar extensiones del esqueleto a bajo nivel añadiendo subconceptos de dominio específico, o a alto nivel añadiendo conceptos de dominio intermedio o superior que cubran nuevas áreas. Si unos sistemas se construyen con la

misma ontología, comparten una estructura subyacente común, por lo tanto, se haría más fácil combinar y compartir sus bases de conocimiento y mecanismos de inferencia.

A veces el concepto de ontología se diluye, en el sentido que las taxonomías se consideran ontologías completas (Studer et al., 1998). Por ejemplo, UNSPSC (http://www.unspsc.org/), e-cl@ss (http://www.eclass.org/) y RosettaNet (http://www.rosettanet.org/), propuestas para estándares en el domino del e-comercio, y el directorio Yahoo!, una taxonomía para buscar en la Web, se consideran también ontologías (Lassila y McGuiness, 2001) porque proporcionan una conceptualización consensual de un domino dado. La comunidad de ontologías distingue las ontologías que son principalmente taxonomías, de las ontologías que modelan el dominio de forma más profunda y proporcionan más restricciones en semántica de dominio. La comunidad las llama ontologías ligeras y pesadas respectivamente. Por una parte, las ontologías ligeras incluyen conceptos, taxonomías de conceptos, relaciones entre conceptos, y propiedades que describen conceptos. Por otra parte, las ontologías pesadas añaden axiomas y restricciones a las ontologías ligeras. Los axiomas y restricciones aclaran el significado de los términos recogidos en la ontología.

Puesto que las ontologías son muy utilizadas para diferentes propósitos (procesamiento del lenguaje natural, manejo del conocimiento, e-comercio, integración inteligente de la información, *the Semantic Web*, etc.) en diferentes comunidades (ingeniería de conocimiento, ingeniería de bases de datos y software), Uschold y Jasper (1999) proporcionaron una nueva definición de la palabra ‘ontología’ para popularizarla en otras disciplinas. Nótese que la comunidad de bases de datos, así como la comunidad de diseño orientado a objetos, también construyen modelos de dominio usando conceptos, relaciones, propiedades, etc., pero la mayoría de las veces ambas comunidades imponen menos restricciones semánticas que aquellas impuestas en las ontologías pesadas. Uschold y Jasper definieron así una ontología:

<< Una ontología puede tomar varias formas, pero incluirá necesariamente un vocabulario de términos y alguna especificación de su significado. Esto incluye definiciones y una indicación de cómo los conceptos están interrelacionados, lo que impone colectivamente una estructura en el dominio y contiene las posibles interpretaciones de los términos. >>

En este apartado hemos recogido las definiciones más relevantes de la palabra ‘ontología’, aunque se pueden encontrar más en la literatura de la Inteligencia Artificial. Diferentes definiciones proporcionan diferentes y complementarios puntos de vista sobre la misma realidad. Algunos autores proporcionan definiciones que son independientes de los métodos seguidos para construir la ontología y de su uso en aplicaciones, mientras que otras definiciones están influenciadas por su proceso de desarrollo. Como conclusión principal de este apartado, podemos decir que las ontologías pretenden capturar conocimiento consensual de un modo genérico, y que pueden ser reutilizadas y compartidas a través de aplicaciones software y por grupos de personas. Normalmente son construidas de forma cooperativa por diferentes grupos de personas en diferentes lugares.

4.2- Lenguajes para construir ontologías

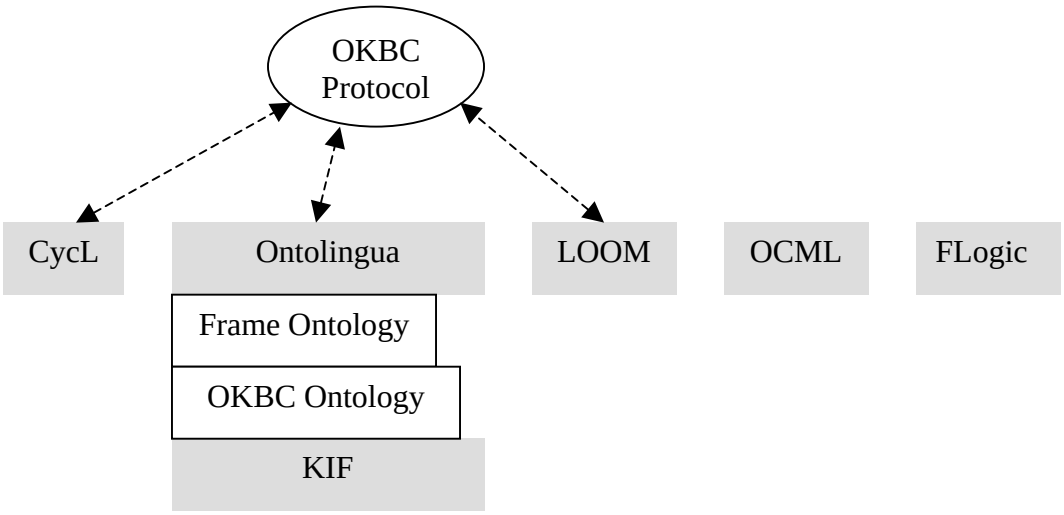
4.2.1- Introducción

Una de las decisiones clave a tomar en el proceso de desarrollo de ontologías es elegir el lenguaje (o conjunto de lenguajes) en el que se implementará la ontología. En las últimas décadas han sido creados muchos lenguajes de implementación de ontologías, y otros lenguajes y sistemas generales de Representación de Conocimiento han sido utilizados para implementar ontologías aunque no fueron creados específicamente para este propósito.

Normalmente, la elección de un lenguaje de ontologías no se basa en la Representación de Conocimiento y en los mecanismos de inferencia necesitados por la aplicación que usa la ontología, sino en las preferencias individuales del desarrollador. Una elección equivocada del lenguaje usado para implementar una ontología puede causar problemas una vez que la ontología está siendo utilizada en una aplicación.

4.2.2- Evolución de los lenguajes de ontologías

A principios de los 90, se creó un conjunto de lenguajes de ontologías basados en Inteligencia Artificial. Básicamente, los paradigmas de Representación de Conocimiento fundamentales en tales lenguajes de ontologías se basaban en primer orden lógico (Ej. KIF), en marcos combinados con primer orden lógico (Ej. CycL, Ontolingua, OCML y FLogic), y en lógica de descripción (Ej. LOOM). OKBC también fue creado como un protocolo para acceder a ontologías implementadas en diferentes lenguajes con un paradigma de Representación de Conocimiento basado en marcos. En la siguiente figura se muestra la disposición general de estos lenguajes:



Lenguajes de ontologías tradicionales

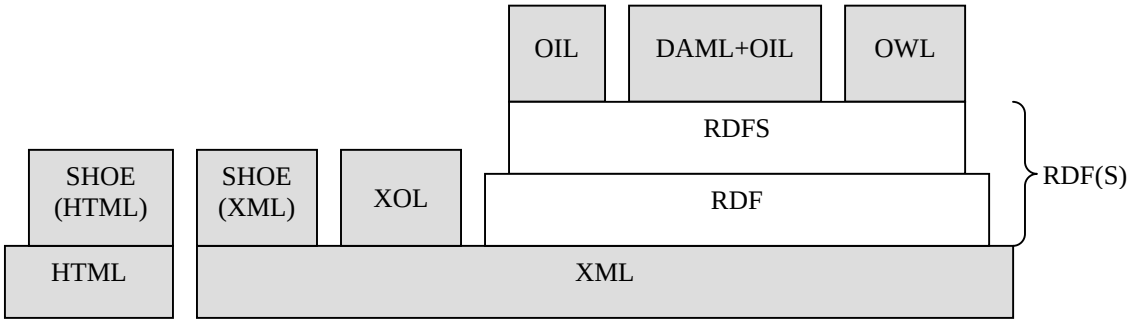
Del anterior conjunto de lenguajes, CycL (Lenat y Guha, 1990) fue el primero en ser creado. CycL está basado en marcos y primer orden lógico, y se utilizó para construir la Ontología Cyc.

KIF (Genesereth y Fikes, 1992; NCITS, 1998) fue creado más tarde, en 1992, y fue diseñado como un formato de intercambio de conocimiento; KIF está basado en primer orden lógico. Puesto que era difícil crear ontologías directamente en KIF, se creó Ontolingua (Farquhar et al., 1997) sobre él. Por lo tanto, Ontolingua se basa en KIF, y es el lenguaje de ontologías soportado por el Servidor de Ontolingua. Este lenguaje tiene una sintaxis semejante a Lisp (una familia de lenguajes, cuyo desarrollo empezó a finales de los 50, para el procesamiento de listas en Inteligencia Artificial) y sus paradigmas fundamentales de Representación de Conocimiento son marcos y primer orden lógico. Ontolingua fue considerado un estándar *de facto* por la comunidad de ontologías en los 90.

Al mismo tiempo se construyó LOOM (MacGregor, 1991), aunque no estaba destinado a implementar ontologías, sino para bases generales de conocimiento. LOOM está basado en lógica de descripción y reglas de producción, y proporciona características de clasificación automática de conceptos. OCML (Motta, 1999) se desarrolló más tarde, en 1993, como un tipo de 'Ontolingua operacional'. En realidad, la mayoría de las definiciones que se pueden expresar en OCML son similares a las definiciones correspondientes en Ontolingua. OCML fue creado para desarrollar ontologías ejecutables y modelos en métodos de resolución de problemas. Finalmente, en 1995 se desarrolló FLogic (Kifer et al., 1995) como un lenguaje que combinaba marcos y primer orden lógico, aunque no tenía una sintaxis semejante a Lisp.

En la primavera de 1997 comenzó *the High Performance Knowledge Base program* (HPKB). Este programa de investigación fue patrocinado por DARPA y su objetivo era solucionar muchos de los problemas que aparecen normalmente cuando se trata con grandes bases de conocimiento (acerca de eficiencia, creación de contenido, integración del contenido disponible en diferentes sistemas, etc.). Uno de los resultados de este programa fue el desarrollo del protocolo OKBC (*Open Knowledge Base Connectivity*) (Chaudhri et al., 1998). Este protocolo permite acceder a bases de conocimiento almacenadas en diferentes Sistemas de Representación de Conocimiento, que pueden estar basados en diferentes paradigmas de Representación de Conocimiento.

El auge de Internet provocó la creación de lenguajes de ontologías para explotar las características de la Web. Tales lenguajes se llaman normalmente 'lenguajes de ontologías basados en web' (*web-based ontology languages*) o 'lenguajes de ontologías de etiquetado' (*ontology markup languages*). Su sintaxis está basada en lenguajes de etiquetado existentes, tales como HTML (Raggett et al., 1999) y XML (Bray et al., 2000), cuyo propósito no es el desarrollo de ontologías, sino representación de datos e intercambio de datos respectivamente. En la siguiente figura se muestra la relación entre estos lenguajes:



Lenguajes de ontologías de etiquetado

El primer lenguaje de ontologías de etiquetado en aparecer fue SHOE (Luke y Heflin, 2000). SHOE es un lenguaje que combina marcos y reglas. Se construyó como una extensión de HTML, en 1996. Usaba etiquetas diferentes de las de la especificación de HTML, permitiendo de este modo la inserción de ontologías en documentos HTML. Más tarde su sintaxis se adaptó a XML.

El resto de lenguajes de ontologías de etiquetado aquí expuestos están basados en XML. XOL (Karp et al., 1999) se desarrolló como una 'XMLización' de un pequeño subconjunto de primitivas del protocolo OKBC, llamado OKBC-Lite. RDF (Lassila y Swick, 1999) fue desarrollado por el W3C (*the World Wide Web Consortium*) como lenguaje basado en redes semánticas para describir recursos *Web*. Su desarrollo comenzó en 1997, y RDF fue propuesto como una *W3C Recommendation* en 1999. El lenguaje RDF Schema (Brickley y Guha, 2003) también fue construido por el W3C como una extensión de RDF con primitivas basadas en marcos. Este lenguaje fue propuesto como una *W3C Candidate Recommendation* en 2000, y experimentó una revisión en noviembre de 2002, para que su documento de referencia fuera publicado como un *W3C Working Draft*. Fue revisado más tarde, en enero de 2003. La combinación de RDF y RDF Schema se conoce normalmente como RDF(S).

Estos lenguajes han establecido las bases de *the Semantic Web* (Berners-Lee, 1999). En este contexto, se han desarrollado tres lenguajes más como extensiones de RDF(S): OIL, DAML+OIL y OWL. OIL (Horrocks et al., 2000) se desarrolló a principios del año 2000 en el marco de *the European IST project On-To-Knowledge* (http://www.ontoknowledge.org/). Añade primitivas de Representación de Conocimiento basadas en marcos a RDF(S) y su semántica formal se basa en lógica de descripciones. DAML+OIL (Horrocks y van Harmelen, 2001) fue creado más tarde (entre los años 2000 y 2001) por un comité mixto de Estados Unidos y **EU** en el contexto del proyecto DARPA DAML (http://www.daml.org/). Se basaba en la especificación previa de DAML-ONT, que se construyó a finales de 2000, y en OIL. DAML+OIL añade primitivas de Representación de Conocimiento basadas en lógica de descripción a RDF(S). En 2001, el W3C formó un grupo de trabajo llamado *Web-Ontology (WebOnt) Working Group* (http://www.w3.org/2001/sw/WebOnt/). La intención de este grupo era hacer un nuevo lenguaje de ontologías de etiquetado para *the Semantic Web*. El resultado de su trabajo es el lenguaje OWL (Dean y Schreiber, 2003).

4.2.3- Lenguaje utilizado: OWL

Como hemos comentado anteriormente, OWL (*Web Ontology Language*) es el resultado del trabajo del *W3C Web Ontology (WebOnt) Working Group*, que se formó en noviembre de 2001. Este lenguaje deriva de DAML+OIL y lo suplanta. Abarca la mayoría de las características de DAML+OIL y renombra la mayoría de sus primitivas. Como los lenguajes previos, OWL va dirigido a publicar y compartir ontologías en la *Web*. En febrero de 2004 el W3C anunció la aprobación de OWL, que se convirtió en una *W3C Recommendation*. Esto marcó el surgimiento de *the Semantic Web* como una plataforma de grado comercial para información en la *Web*. El despliegue de este estándar en productos y servicios comerciales marca la transición de la tecnología de *the Semantic Web* de lo que fue en gran parte un proyecto de investigación y desarrollo avanzado durante los últimos cinco años, a una tecnología más práctica desplegada en herramientas de mercado común que permiten un acceso más flexible a información estructurada en la *Web*.

OWL está dividido en capas: OWL Lite, OWL DL y OWL Full. OWL Lite extiende RDF(S) y reúne las características más comunes de OWL, por lo tanto, está destinado a usuarios que sólo necesiten crear taxonomías de clases y restricciones simples. OWL DL incluye el vocabulario completo de OWL. Finalmente, OWL Full proporciona más flexibilidad para representar ontologías que OWL DL.

Al igual que DAML+OIL, OWL está construido sobre RDF(S). Por lo tanto, algunas primitivas de RDF(S) son reutilizadas por OWL, y las ontologías de OWL están escritas en XML o con la notación triple de RDF. Nosotros usaremos la sintaxis de XML para todos los ejemplos posteriores.

Como OWL deriva de DAML+OIL, comparte muchas características con este lenguaje. Las principales diferencias entre OWL y DAML+OIL son las siguientes:

- OWL no incluye restricciones numéricas cualificadas (daml:hasClassQ, daml:cardinalityQ, daml:maxCardinalityQ y daml:minCardinalityQ).
- OWL permite definir propiedades simétricas, que no fueron consideradas en DAML+OIL, con la primitiva ‘owl:SymmetricProperty’.
- OWL no renombra las primitivas de RDF(S) reutilizadas por el lenguaje, como ocurría en DAML+OIL. Por ejemplo, rdfs:subClassOf, rdfs:subPropertyOf, etc.
- En OWL han sido renombradas muchas primitivas de DAML+OIL. Por ejemplo, la primitiva ‘daml:toClass’ ha sido renombrada como ‘owl:allValuesFrom’.
- OWL no incluye la primitiva ‘daml:disjointUnionOf’, ya que se puede llevar a cabo combinando ‘owl:unionOf’ y ‘owl:disjointWith’.

4.2.3.1- Representación de conocimiento

Una ontología en OWL comienza con la declaración del nodo raíz de RDF. En este nodo debemos incluir los *namespaces* para las ontologías de Representación de Conocimiento de RDF, RDFS y OWL. Si se utilizan tipos de datos de XML Schema, puede ser útil incluir un *namespace* para XML Schema, que se prefija normalmente como ‘xds’ (y que apunta a la URL más reciente de XML Schema, como en RDF(S)).

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">
```

Como en otros lenguajes, el orden en que aparecen las definiciones en las ontologías de OWL no es relevante, ya que se basa en RDF(S). Sin embargo, las ontologías de OWL normalmente definen primero el encabezado de la ontología y después los términos.

El encabezado de una ontologías de OWL puede incluir: la documentación de la ontología (rdfs:comment); la versión de la ontología (owl:versionInfo); y las ontologías importadas (owl:imports). Nosotros importaremos ontologías sobre unidades y tipos de datos. No es necesario (ni recomendado) importar la ontología de Representación de Conocimiento de OWL. El encabezado de la ontología también puede incluir información sobre control de versión con las primitivas 'owl:backwardCompatibleWith', 'owl:incompatibleWith' y 'owl:priorVersion'. Dentro de la ontología también podemos encontrar definiciones de clases y propiedades *deprecated* con las primitivas 'owl:DeprecatedClass' y 'owl:DeprecatedProperty'.

```
<owl:Ontology rdf:about="">
  <owl:versionInfo>1.0</owl:versionInfo>
  <rdfs:comment>Sample ontology for travel agencies</rdfs:comment>
  <owl:imports rdf:resource="http://delicias.dia.fi.upm.es/owl/units"/>
  <owl:imports rdf:resource="http://www.w3.org/2001/XMLSchema"/>
</owl:Ontology>
```

Los conceptos son conocidos como clases en OWL, y se crean con la primitiva 'owl:Class'. Además de su nombre, una clase puede contener su documentación (con 'rdfs:comment') y cualquier número de expresiones con la siguiente lista de primitivas:

- 'rdf:subClassOf' contiene expresiones de clase. Permite definir las superclases de la clase.
- 'owl:disjointWith' afirma que la clase no puede compartir instancias con la expresión de clase en esta primitiva.
- 'owl:equivalentClass' también contiene expresiones de clase. Esta primitiva define condiciones necesarias y suficientes para la clase (es decir, que se utiliza para redefinir conceptos ya definidos).
- 'owl:oneOf' define una clase enumerando exhaustivamente todas sus instancias. Es una forma de definir una clase extensamente.
- 'owl:intersectionOf', 'owl:unionOf' y 'owl:complementOf' definen una expresión de clase como una conjunción, una disyunción o una negación de otras expresiones de clase respectivamente.

Las dos primeras primitivas ('rdf:subClassOf' y 'owl:disjointWith') definen condiciones necesarias para la clase (se pueden utilizar en la definición de conceptos primitivos), mientras que el resto de primitivas definen condiciones necesarias y suficientes para la clase (es decir, que se utilizan para definir conceptos definidos). En OWL Lite, las únicas primitivas que se pueden utilizar son: 'rdf:subClassOf',

‘owl:equivalentClass’ y ‘owl:intersectionOf’. En todos los casos, sólo se pueden utilizar con identificadores de clase y restricciones de propiedad.

De acuerdo con la terminología de Lógica de Descripción (DL), OWL es un lenguaje SHOIN. Esto significa que se pueden construir expresiones de clase con los siguientes constructores:

- Conjunción (owl:intersectionOf), disyunción (owl:unionOf) y negación (owl:complementOf) de expresiones de clase.
- Colecciones de individuales (owl:oneOf).
- Restricciones de propiedad (owl:Restriction). Contienen una referencia a la propiedad a la que se aplica la restricción con la primitiva ‘owl:onProperty’ y un elemento para expresar la restricción. Se pueden aplicar las siguientes restricciones a las propiedades: restricción de valor (owl:allValuesFrom), restricción existencial (owl:someValuesFrom), filtros de roles (owl:hasValue) y restricción de número (owl:cardinality, owl:maxCardinality, owl:minCardinality).
- Además, las expresiones de rol pueden expresar roles inversos (owl:inverseOf) y jerarquías de roles (rdfs:subPropertyOf).

La gramática de OWL para construir expresiones de clase es muy similar a la de DAML+OIL. Nótese que las expresiones de booleanos y enumeraciones se encuentran entre las etiquetas ‘<owl:Class>..</owl:Class>’.

class-name |
<**owl:Class**> boolean-expr </**owl:Class**> |
<**owl:Class**>
  <**owl:oneOf rdf:parseType**="Collection**">**
    instance-expr+
  </**owl:oneOf**>
</**owl:Class**> |
<**owl:Restriction**>
  <**owl:onProperty**> property-name </**owl:onProperty**>
  restriction-expr
</**owl:Restriction**>

‘boolean-expr’ se define así:

<**owl:intersectionOf**>
    class-expr class-expr+
</**owl:intersectionOf**> |
<**owl:unionOf**> class-expr class-expr+ </**owl:unionOf**> |
<**owl:complementOf**> class-expr </**owl:complementOf**>

‘instance-expr’ representa una instancia de una clase. Se define así:

<**owl:Thing rdf:resource**="#instanceName**"/**>

‘restriction-expr’ representa una restricción sobre una propiedad cuando es aplicada a la clase. Puede ser alguna de las siguientes:

<**owl:allValuesFrom**> class-expr </**owl:allValuesFrom**> |
<**owl:someValuesFrom**> class-expr </**owl:someValuesFrom**> |
<**owl:hasValue**> instance-name </**owl:hasValue**> |
<**owl:cardinality rdf:datatype**="&xsd;nonNegativeInteger">
non-neg-integer
</**owl:cardinality**> |
<**owl:maxCardinality rdf:datatype**="&xsd;nonNegativeInteger">
non-neg-integer
</**owl:maxCardinality**> |
<**owl:minCardinality rdf:datatype**="&xsd;nonNegativeInteger">
non-neg-integer
</**owl:minCardinality**>

En OWL Lite las expresiones de clase sólo pueden contener nombres de clase y restricciones de propiedad. La primitiva 'owl:hasValue' no se puede utilizar en restricciones de propiedad. Además, las primitivas 'owl:allValuesFrom' y 'owl:someValuesFrom' sólo contienen identificadores de clase o tipos de datos nombrados, y las restricciones de cardinalidad sólo toman los valores 0 ó 1. OWL DL no impone ninguna de estas restricciones.

Las funciones no son componentes del modelo de conocimiento de OWL, aunque las funciones binarias se pueden representar con la primitiva 'owl:FunctionalProperty'.

Los axiomas formales tampoco son componentes del modelo de conocimiento de OWL.

Finalmente, las instancias se definen utilizando sólo vocabulario de RDF. En OWL, debemos utilizar el atributo 'rdf:datatype' para expresar el tipo de valor de las propiedades de tipos de datos.

4.2.3.2- Mecanismos de razonamiento

La semántica de modelo teórico de OWL está descrita por Patel-Schneider y compañeros (2003). Esta semántica se describe de dos formas diferentes: como una extensión de la teoría de modelo RDF(S) y como una semántica directa de modelo teórico de OWL. Ambas tienen las mismas consecuencias semánticas en las ontologías de OWL, y están basadas en la semántica de modelo teórico de DAML+OIL, teniendo en cuenta las diferencias entre ambos lenguajes.

OWL permite incluir algunas declaraciones adicionales (triples de RDF) en sus ontologías, a parte de aquellas explícitamente definidas en el lenguaje. Sin embargo, oculta las consecuencias semánticas (o su carencia) de tales triples adicionales.

Carroll y De Roo (2003) han definido un conjunto de procesos de prueba, incluidas pruebas de implicación, pruebas de no implicación, pruebas de coherencia, pruebas de incoherencia, etc., que ilustran la correcta utilización de OWL y del significado formal de sus constructores.

Debido a sus similitudes con OIL y DAML+OIL, los motores de inferencia utilizados por estos lenguajes (FaCT, RACER, TRIPLE, etc.) se pueden adaptar fácilmente para

razonar con él. Aún no hay muchos motores de inferencia disponibles para razonar con OWL, pero se prevé que los habrá pronto. Un motor de razonamiento ya disponible es Euler (http://www.agfa.com/w3c/euler/).

Al igual que con otros lenguajes, estos motores de inferencia permitirán llevar a cabo clasificaciones automáticas de conceptos de ontologías de OWL, y detectar anomalías en taxonomías de conceptos de OWL.

Además, podemos decir que la herencia múltiple está permitida en ontologías de OWL. En la semántica de OWL, sin embargo, no hay explicación de como pueden ser resueltos los conflictos en la herencia múltiple. Se pueden realizar comprobaciones de restricciones en los valores de propiedades y sus cardinalidades.

OWL asume razonamiento *monotonic*, aunque las definiciones de clase y las definiciones de propiedad están separadas en diferentes recursos Web. Esto significa que los hechos y las implicaciones declarados explícitamente u obtenidos con motores de inferencia sólo pueden ser añadidos, nunca eliminados, y que información nueva no puede negar información previa.

Las herramientas capaces de editar ontologías de RDF(S) también se pueden utilizar para desarrollar ontologías de OWL a condición de que el desarrollador de la ontología use las primitivas de Representación de Conocimiento de OWL. Además, se pueden utilizar motores de consulta de RDF(S), sistemas de almacenamiento y analizadores para manejar ontologías de OWL, ya que se pueden serializar en RDF(S). Finalmente, debemos añadir que ya hay disponibles sistemas que transforman ontologías de DAML+OIL en ontologías de OWL (http://www.mindswap.org/2002/owl.html).

4.3- Herramientas para ontologías

4.3.1- Introducción

Construir ontologías es una tarea compleja y consume mucho tiempo, y lo es aún más si los desarrolladores de ontologías las tienen que implementar directamente en un lenguaje de ontologías, sin ningún tipo de herramienta de apoyo. Para facilitar esta tarea, a mediados de los 90 se crearon los primeros entornos de construcción de ontologías. Proporcionaron interfaces que ayudaban a los usuarios a realizar algunas de las principales actividades del proceso de desarrollo de ontologías, como conceptualización, implementación, comprobación de consistencia y documentación. En los últimos años, el número de herramientas para ontologías ha aumentado mucho y se han diversificado.

4.3.2- Evolución de las herramientas para ontologías

La tecnología de las herramientas de ontologías ha mejorado enormemente desde la creación de los primeros entornos. Si consideramos la evolución de las herramientas de desarrollo de ontologías desde que aparecieron a mitad de los 90, podemos distinguir dos grupos:

- Las herramientas cuyo modelo de conocimiento proyecta directamente a un lenguaje de ontologías. Estas herramientas se desarrollaron como editores de ontologías para un lenguaje específico. En este grupo se incluyen: el Servidor de Ontolingua (Farquhar et al., 1997), que soporta la construcción de ontologías con Ontolingua y KIF; OntoSaurus (Swartout et. al, 1997) con Loom; WebOnto (Domingue, 1998) con OCML; y OilEd (Bechhofer et al., 2001) primero con OIL y más tarde con DAML+OIL.

- Los conjuntos integrados de herramientas cuya principal característica es que tienen una arquitectura extensible y cuyo modelo de conocimiento normalmente es independiente del lenguaje de ontología. Estas herramientas proporcionan un conjunto de servicios relacionados con ontologías y se extienden fácilmente con otros módulos para proporcionar más funciones.

A continuación comentaremos como tuvo lugar está evolución y remarcaremos las características más relevantes de las herramientas de cada grupo.

Como hemos dicho antes, la principal característica del primer grupo de herramientas es que tienen una fuerte relación con un lenguaje de ontologías específico. Estas herramientas se crearon para permitir editar y leer ontologías en sus lenguajes correspondientes y para importar y exportar ontologías de / a otros lenguajes de ontologías, pero requieren que los usuarios tengan conocimiento de su lenguaje de ontologías fundamental.

El **Servidor de Ontolingua** (Farquhar et al., 1997) fue la primera herramienta de ontologías creada. Apareció a mediados de los 90 y se construyó para facilitar el desarrollo de las ontologías de Ontolingua con una interfaz basada en Web.

En paralelo al desarrollo del Servidor de Ontolingua, **OntoSaurus** (Swartout et al., 1997) se implementó como un editor y navegador Web para ontologías de LOOM.

En 1997 se publicó **WebOnto** (Domingue, 1998). WebOnto es un editor para ontologías de OCML.

OilEd (Bechhofer et al., 2001) se desarrolló en 2001 como un editor para ontologías OIL. Con la creación de DAML+OIL, OilEd se adaptó para manejar ontologías de DAML+OIL y más tarde a las de OWL.

En los últimos años, se ha desarrollado una nueva generación de entornos de ingeniería de ontologías. El diseño racional de estos entornos es mucho más ambicioso que el de las herramientas anteriores.

Protégé-2000 (Noy et al., 2000) es una aplicación autónoma de código fuente libre con una arquitectura extensible. El núcleo de Protege-2000 es su editor de ontologías, que se puede extender con *plug-ins* que añaden más funciones al entorno, como importación y exportación de lenguajes de ontologías (Flogic, Jess, OIL, XML, Prolog), acceso a OKBC, creación y ejecución de restricciones (PAL), mezcla de ontologías (PROMPT), etc.

WebODE (Arpírez et al., 2003) también es un conjunto de ingeniería de ontologías extensible basada en un servidor de aplicaciones, cuyo desarrollo empezó en 1999. El

núcleo de WebODE es su servicio de acceso a ontologías, que es utilizado por todos los servicios y aplicaciones del servidor.

OntoEdit (Sure et al., 2002) es un entorno extensible y flexible basado en una arquitectura *plug-in*. Su editor de ontologías es una aplicación autónoma que permite editar y leer ontologías, e incluye funciones para la construcción colaborativa de ontologías, la deducción, el manejo de léxicos de dominio, etc.

El conjunto de herramientas **KAON** (Maedche et al., 2003) es un entorno de ingeniería de ontologías extensible de código fuente libre. El núcleo de este conjunto de herramientas es el API de ontologías, que define su modelo de conocimiento fundamental basado en una extensión de RDF(S).

4.3.3- Herramienta utilizada: Protégé-2000

Protégé-2000 (http://protege.stanford.edu/) (Noy et al., 2000) es la última versión de la rama de herramientas de Protégé, creada por el grupo *the Stanford Medical Informatics* (*SMI*) en la *Stanford University*. La primera herramienta de Protégé se creó en 1987 (Musen, 1989); su principal objetivo era simplificar el proceso de adquisición de conocimiento para sistemas expertos. Para alcanzar este objetivo, se utilizaba el conocimiento adquirido en etapas anteriores del proceso para generar formularios a medida para adquirir más conocimiento. Desde entonces, Protégé ha pasado por varias publicaciones y se ha enfocado a diferentes aspectos de la adquisición de conocimiento (bases de conocimiento, métodos de resolución de problemas, ontologías, etc.), cuyo resultado es Protégé-2000. La historia de la rama de herramientas de Protégé fue descrita por Gennari y compañeros (2003). Tiene alrededor de 7000 usuarios registrados.

Protégé-2000 está orientado a la tarea de desarrollo de ontologías y bases de conocimiento.

4.3.3.1- Arquitectura

Protégé-2000 es una aplicación autónoma basada en Java para ser instalada y ejecutada en un ordenador local. El núcleo de esta aplicación es el editor de ontologías, descrito posteriormente.

Protégé-2000 tiene una arquitectura extensible para crear e integrar fácilmente nuevas extensiones (*plug-ins*). Normalmente estas extensiones llevan a cabo funciones que no proporciona la distribución estándar de Protégé-2000 (otros tipos de visualización, nuevos formatos de importación y exportación, etc.), implementan aplicaciones que utilizan las ontologías de Protégé-2000, o permiten configurar el editor de ontologías. La mayoría de estos *plug-ins* están disponibles en *the Protégé Plug-in Library* (http://protege.stanford.edu/plugins.html), donde se pueden encontrar contribuciones de muchos grupos de investigación diferentes.

A continuación describimos los tres grupos de *plug-ins* que puede desarrollar Protégé-2000 con ejemplos actuales de tales tipos de *plug-ins*:

- **Tab Plug-ins**: son los más comunes en Protégé-2000 y proporcionan funciones que no cubre la distribución estándar del editor de ontologías. Para llevar a cabo su tarea, los *tab Plug-ins* extienden el editor de ontologías con un tabulador adicional para que los usuarios puedan acceder a sus funciones desde él. Las siguientes funciones están cubiertas por algunos de los *plug-ins* disponibles: visualización gráfica de ontologías (*Jambalaya tab* y *Onto Viz tab*), mezcla y versionado de ontologías (*PROMPT tab*), manejo de grandes fuentes de conocimiento *on-line* (*UMLS* y *WordNet tabs*), acceso a ontologías de OKBC (*OKBC tab*), construcción y ejecución de restricciones (*PAL tab*), y motores de inferencia utilizando *Jess* (Friedman-Hill, 2003), *Prolog*, *FLogic*, *FaCT* y *Algernon* (*Jess*, *Prolog*, *FLORA*, *OIL* y *Algernon tabs* respectivamente).

- **Slot widgets**: se utilizan para visualizar y editar valores de *slots* sin las facilidades por defecto de visualización y edición. También hay *slot widgets* para visualizar imágenes, vídeo y audio, y para manejar datos, unidades de medida, cambiar valores entre *slots*, etc.

- **Backends**: permiten a los usuarios exportar e importar ontologías en diferentes formatos: *RDF Schema*, *XML*, *XML Schema*, etc. Hay un *backend* para almacenar y recuperar ontologías en bases de datos para que no sólo puedan ser almacenadas en ficheros de *CLIPS* (formato de almacenamiento por defecto usado por Protégé-2000) sino que también se puedan almacenar en cualquier base de datos compatible con *JDBC*.

4.3.3.2- Modelo de conocimiento

El modelo de conocimiento de Protégé-2000 está basado en marcos y primer orden lógico. Es compatible con OKBC, lo que significa que los principales componentes de modelo de Protégé-2000 son clases, *slots*, *facets* e instancias. Las clases están organizadas en jerarquías de clases donde está permitida la herencia múltiple y los *slots* pueden estar organizados también en jerarquías de *slots*. El modelo de conocimiento permite expresar restricciones en el lenguaje *PAL*, que es un subconjunto de *KIF*, y permite expresar metaclases, que son clases cuyas instancias también son clases.

Las clases en Protégé-2000 pueden ser concretas o abstractas. La primera puede tener instancias directas mientras que la última no puede; las instancias de las clases se deben definir como instancias de alguna de sus subclases en la taxonomía de clases.

Los *slots* son globales a la ontología (dos *slots* diferentes no pueden tener el mismo nombre en una ontología) y pueden ser obligatorios en las clases adjuntas.

4.3.3.3- Editor de ontologías

El editor de ontologías de Protégé-2000 muestra y edita la taxonomía de clases de la ontología utilizando una estructura de árbol, define *slots* globales, adjunta *slots* a clases, crea instancias de clases, etc. Proporciona búsqueda común, las funciones *copy&paste* y *drag&drop*, entre otras, y también diferentes menús *pop-up* de acuerdo con el tipo y las características del componente de ontologías que está siendo editado.

Una de las características destacadas del editor de ontologías de Protégé-2000, en comparación con otros editores de ontologías, es que podemos diseñar las capas de

pantalla utilizadas para crear instancias. Los desarrolladores de ontologías pueden elegir que tipo de formularios se presentarán, donde estarán situados los campos del formulario para cada *slot*, que *slot widgets* quieren utilizar para cada *slot*, etc.

El editor de ontologías de Protégé-2000 también contiene un tabulador de consultas (*Queries tab*), con el que los usuarios pueden crear consultas sobre instancias que tienen o no un valor específico para un *slot*, sobre instancias cuyo valor de *slot* es mayor o menor que un número determinado, etc. Las consultas se almacenan en una librería de consultas y se pueden combinar.

El editor de ontologías genera diferentes tipos de documentación de ontologías: documentos HTML y estadísticas de ontologías.

4.3.3.4- Interoperabilidad

Una vez se ha creado una ontología en Protégé-2000, hay muchas formas de acceder a las ontologías de Protégé-2000 desde aplicaciones basadas en ontologías.

Se puede acceder a todos los términos de ontologías con el *Protégé-2000 Java API*. Por lo tanto, es fácil para las aplicaciones basadas en ontologías acceder a ontologías, así como utilizar otras funciones proporcionadas por diferentes *plug-ins*.

Las ontologías de Protégé-2000 se pueden exportar e importar con alguno de los *backends* proporcionados en la publicación estándar o como *plug-ins*: RDF(S), XML, XML Schema y XMI. Una vez generado y guardado localmente el correspondiente archivo de salida, puede ser utilizado por cualquier aplicación local capaz de manejar este formato. En el caso de XMI, el modelo de UML traducido se puede utilizar para obtener clases de Java de él.

5- OTRAS HERRAMIENTAS UTILIZADAS

5.1- HTMLParser

HTMLParser es una librería de Java que sirve para analizar *HTML* en tiempo real. Lo que ha atraído del *HTMLParser* a la mayoría de desarrolladores ha sido su simplicidad en diseño, velocidad y habilidad para manejar *HTML*.

Sus dos utilidades fundamentales son la extracción y la transformación, y contiene filtros, *visitors*, etiquetas propias y *JavaBeans* fáciles de utilizar. Es un paquete rápido, robusto y bien testado.

En general, para utilizar el *HTMLParser* sólo es necesario ser capaz de escribir código en lenguaje Java. Aunque se proporcionan algunos ejemplos de programas que pueden ser muy útiles, es más que probable que necesitemos (o queramos) crear nuestros propios programas o modificar los provistos para conseguir la aplicación deseada.

Para utilizar la librería hay que añadir el *htmllexer.jar* o *htmlparser.jar* al *classpath* cuando compilemos y ejecutemos. El *htmllexer.jar* proporciona acceso a bajo nivel a *strings* genéricos, comenta y etiqueta nodos de una página de manera lineal, llana y secuencial. El *htmlparser.jar*, que incluye las clases encontradas en *htmllexer.jar*, proporciona acceso a una página como una secuencia de etiquetas diferenciadas que contienen nodos con *strings*, comentarios y otras etiquetas.

El *parser* trata de nivelar etiquetas de apertura con etiquetas de cierre para presentar la estructura de la página, mientras que el *lexer* simplemente separa nodos. Si nuestra aplicación sólo requiere conocimiento estructural modesto de la página y trata primordialmente con nodos individuales y aislados, deberíamos considerar la utilización del *lexer*. Pero si nuestra aplicación requiere conocimiento de la estructura en conjunto de la página (Ej. procesamiento de tablas), probablemente necesitemos el *parser*.

5.1.1- Extracción

La extracción abarca todos los programas de recuperación de información que no sirven para conservar la página fuente. Se utiliza para:

- Extracción de texto, para utilizarlo como entrada en bases de datos de motores de búsqueda de texto, por ejemplo.
- Extracción de *links*, para moverse a través de páginas web o conseguir direcciones de e-mail.
- *Screen scraping*, para entrada de datos programados de páginas web.
- Extracción de recursos, recogiendo imágenes y sonido.
- Comprobación de *links*, para asegurar que son válidos.
- Control de *site*, comprobando diferencias de páginas más allá de diferencias simplistas.

Hay varias facilidades en el código base del HTMLParser para ayudar con la extracción, incluyendo filtros, *visitors* y *JavaBeans*.

5.1.2- Transformación

La transformación incluye todo proceso donde la entrada y la salida son páginas de *HTML*.

- Reescritura de *URLs*, modificando algunos o todos los *links* de una página.
- Captura de *site*, moviendo contenido de la *Web* al disco local.
- Censura, eliminando palabras y frases ofensoras de páginas.
- Limpieza de *HTML*, corrigiendo páginas erróneas.
- Eliminación de anuncios, quitando *URLs* que referencian publicidad.
- Conversión a *XML*, transformando páginas web existentes a *XML*.

5.2- Lucene

Lucene, del proyecto Apache Jakarta, es una librería completamente equipada escrita enteramente en Java. Es un motor de búsqueda e indexado de alto rendimiento. "Indexado" significa partir oraciones en palabras individuales para almacenarlas en una cierta clase de directorio. Este directorio se puede utilizar para búsquedas rápidas de palabras o de combinación de palabras. El indexado y la búsqueda no son, de ningún modo, tecnología simple, aunque Lucene es muy fácil de utilizar. Se necesitan pocas líneas de código para tener un indexador y buscador simple terminado. Si se quiere refinar el programa, hay muchas posibilidades gracias a la abierta arquitectura de Lucene. Es una tecnología conveniente para casi cualquier uso que requiera búsqueda de texto completo, especialmente *cross-platform*.

Las clases más importantes son:

- **IndexWriter:** para crear y construir el índice.

- **IndexSearcher:** para consultar el índice.

- **Document:** es lo que se da al *IndexWriter* y lo que se obtiene del *IndexSearcher*.

- **Query:** contiene una consulta.

- **QueryParser:** construye un objeto *Query* (de un *string*).

- **Analyzer:** contiene las políticas para extraer las palabras o *tokens* de los documentos fuente. El *IndexWriter* y el *QueryParser* utilizan los *Analyzer*s.

El *IndexWriter* y el *QueryParser* deben utilizar el mismo tipo de *Analyzer*, sino podemos obtener resultados incorrectos en la consulta.

5.2.1- Analyzers

El proceso de analizar el texto y de encontrar todas las palabras no es tan simple como se podría pensar. Algunas de las cosas que se tendrán que considerar son:

- ¿Qué hacer con las letras mayúsculas y minúsculas?

- ¿Qué hacer con palabras comunes como “the”, “a” y “of”? ¿Se deben indexar?

- ¿Qué hacer con las formas singulares y plurales de sustantivos?

- ¿Qué hacer con las abreviaturas? Es I.B.M. lo mismo que IBM?

- ¿Debe separarse una dirección de e-mail en partes más pequeñas?

Lucene ofrece un conjunto de *Analyzer*s, utilizando *tokenizers* y filtros que cumplen varios propósitos:

- **SimpleAnalyzer**: es un *Analyzer* que filtra *LetterTokenizer* (un *tokenizer* que divide el texto en las no-letras) con *LowerCaseFilter* (transforma las letras a minúsculas).

- **StopAnalyzer**: filtra *LetterTokenizer* con *LowerCaseFilter* y *StopFilter* (elimina las *stop words* como “the”, y “a” de un flujo de *tokens*).

- **StandardAnalyzer**: filtra *StandardTokenizer* (un *tokenizer* basado en la gramática) con *StandardFilter*, *LowerCaseFilter* y *StopFilter*.

- **WhitespaceAnalyzer**: un *Analyzer* que utiliza *WhitespaceTokenizer* (un *tokenizer* que divide el texto en los espacios en blanco. Las secuencias adyacentes de caracteres que no sean espacios en blanco, forman *tokens*).

Nota: Las *stop words* son las palabras más comunes de un lenguaje, como por ejemplo los artículos, las preposiciones o las conjunciones.

Para entender mejor su funcionamiento, aquí tenemos dos ejemplos donde se muestra el análisis de dos oraciones con cada uno de los *Analyzer*s:

Análisis de “The quick brown fox jumped over the lazy dogs”

- WhitespaceAnalyzer:
  [The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

- SimpleAnalyzer:
  [the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

- StopAnalyzer:
  [quick] [brown] [fox] [jumped] [over] [lazy] [dogs]

- StandardAnalyzer:
  [quick] [brown] [fox] [jumped] [over] [lazy] [dogs]

Análisis de “XY&Z Corporation - xyz@example.com”

- WhitespaceAnalyzer:
  [XY&Z] [Corporation] [-] [xyz@example.com]

- SimpleAnalyzer:
  [xy] [z] [corporation] [xyz] [example] [com]

- StopAnalyzer:
  [xy] [z] [corporation] [xyz] [example] [com]

- StandardAnalyzer:
  [xy&z] [corporation] [xyz@example] [com]

5.2.2- Consultas

Las consultas dadas al *QueryParser* pueden ser bastante complejas. En la página web de Lucene se puede leer más sobre la sintaxis (http://jakarta.apache.org/lucene/docs/queryparsersyntax.html), pero para hacernos una idea aquí hay algunos ejemplos:

- Java AND Eclipse
- Java OR Tomcat (igual que: Java Tomcat)
- Java AND NOT Microsoft
- Java* (todo lo que comienza con “Java”)
- “Java program” (las palabras “Java” y “program” deben estar juntas)

5.2.3- Utilidad en el proyecto

Utilizamos esta herramienta para analizar palabra a palabra el texto de páginas web. De los diferentes analizadores que proporciona *Lucene*, hemos decidido utilizar el *StopAnalyzer* porque además de ignorar todo lo que no sean letras (Ej. símbolos y números), ignora también las *stop words*, y no nos interesa que en la ontología construida aparezcan este tipo de palabras. Por ejemplo, si queremos construir la ontología a partir de la palabra *virus* no aparecerán clases como *the virus* o *your virus*.

Incluso acepta una lista de las *stop words* que debe ignorar. La lista que utilizamos en el proyecto se encuentra en la dirección:

http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words

y nos pareció lo suficientemente completa. Está almacenada en un archivo de texto, de este modo el usuario puede decidir su propia lista y simplemente tiene que sustituir el archivo de texto.

5.3- Snowball

Antes de explicar qué es Snowball vamos a comentar dos conceptos con los que está muy relacionado: Recuperación de datos y *Stemming*.

5.3.1- Recuperación de datos

La recuperación de datos (IR) es esencialmente una cuestión de decidir qué documentos dentro de una colección se deben recuperar para satisfacer la necesidad de información de un usuario. Esta necesidad de información se representa por una consulta o un perfil, y contiene uno o más términos de búsqueda más, quizá, una cierta información adicional como grados de importancia. Por lo tanto, la decisión de la recuperación se toma comparando los términos de la consulta con los términos índice (las palabras o las frases importantes) que aparecen en el propio documento. La decisión puede ser binaria (recuperar / rechazar), o puede implicar la estimación del grado de importancia que tiene que el documento con relación a la consulta.

Desafortunadamente, las palabras que aparecen en documentos y en consultas tienen a menudo muchas variantes morfológicas. Así, los pares de términos tales como *computing* y *computation* no serán reconocidos como equivalentes sin un cierto tipo de proceso de lenguaje natural (NLP).

5.3.2- Stemming

En la mayoría de los casos, las variantes morfológicas de palabras tienen interpretaciones semánticas similares y se pueden considerar como equivalentes para el propósito de aplicaciones de IR. Por esta razón, se han desarrollado un número de *stemming Algorithms* o *stemmers*, que tratan de reducir una palabra a su *stem* o raíz. Así, los términos clave de una consulta o de un documento son representados por *stems* y no por las palabras originales. Esto no sólo significa que diversas variantes de un término se puedan reducir a una sola forma representativa, también reduce el tamaño del diccionario, es decir, el número de términos distintos que se necesitan para representar un conjuntos de documentos. Un tamaño menor del diccionario da lugar a un ahorro del espacio de almacenaje y del tiempo de procesamiento.

Para los propósitos de IR, no importa generalmente si los *stems* generados son auténticas palabras o no (así, *computation* podría ser reducido a *comput*) a condición de que diversas palabras con el mismo ‘significado base’ se reduzcan a la misma forma y que las palabras con significados distintos se mantengan separadas. Un algoritmo que trata de convertir una palabra en su raíz lingüísticamente correcta (*compute* en este caso) se suele llamar un *lemmatizer*.

Los *stemmers* y *lemmatizers* también se utilizan extensamente dentro del campo de la lingüística computacional.

Un ejemplo de algoritmo de *stemming* es el *Porter stemming algorithm* (o *Porter stemmer*), creado por Martin Porter en 1980. Es un proceso para eliminar las terminaciones morfológicas e inflexionales más comunes de palabras en inglés. Su uso principal es como parte de un proceso de normalización de términos que se hace normalmente cuando se crean sistemas de recuperación de datos.

El algoritmo se ha utilizado, cotizado y adaptado extensamente en los últimos 20 años. Desafortunadamente, abundan variantes suyas que reclaman ser implementaciones verdaderas, y esto puede causar confusión.

5.3.3- Snowball

Snowball es un lenguaje que sirve para definir algoritmos de *stemming* y a partir del cuál se pueden generar rápidos programas *stemmer* en ANSI C o Java.

Snowball se creó por dos razones principales:

- La falta de algoritmos de *stemming* fácilmente disponibles para idiomas diferentes del inglés.

- La falta de promoción de implementaciones exactas del *Porter stemming algorithm*.

En este proyecto se ha utilizado una versión precompilada de los *stemmers* de Snowball.

Para ver como funciona el *Analyzer* de Snowball, usaremos los dos ejemplos que hemos utilizado con los *Analyzers* de Lucene,. donde se muestra el análisis de dos oraciones:

Análisis de “The quick brown fox jumped over the lazy dogs”

- SnowballAnalyzer:
  [quick] [brown] [fox] [jump] [over] [lazi] [dog]

Análisis de “XY&Z Corporation - xyz@example.com”

- SnowballAnalyzer:
  [xy&z] [corpor] [xyz@exampl] [com]

5.3.4- Utilidad en el proyecto

Lo utilizamos para comprobar si dos palabras pertenecen a la misma familia, es decir, contienen la misma raíz:

- Entre palabras de una misma clase, para evitar clases como *biosensor optical biosensors* o *cancerous ovarian cancer*, ya que tanto *biosensor* y *biosensors* como *cancerous* y *cancer*, son de la misma familia.

- Entre palabras de diferentes clases, para evitar que dos clases de la ontología se diferencien únicamente por dos palabras de la misma familia. Por ejemplo, las clases

optical biosensor y *optic biosensor*, ya que *optical* y *optic* pertenecen a la misma familia; y las clases *worlds civil war* y *civil world war*, ya que sólo se diferencian por las palabras *worlds* y *world*, que pertenecen a la misma familia.

5.4- WordNet

Antes de explicar qué es WordNet, haremos una introducción sobre las ontologías lingüísticas, a las que pertenece.

5.4.1- Ontologías lingüísticas

El propósito de este tipo de ontologías es describir conceptos, y no modelar un dominio específico. Ofrecen una cantidad bastante heterogénea de recursos, utilizados principalmente en procesamiento de lenguaje natural. La principal característica de estas ontologías es que están ligadas a la semántica de unidades gramaticales (palabras, grupos nominales, adjetivos, etc.). Algunos ejemplos de ontologías lingüísticas son: WordNet, EuroWordNet, Generalized Upper Model (GUM), Mikrokosmos y SENSUS.

La mayoría de ontologías lingüísticas usan palabras como unidades gramaticales. En realidad, de los ejemplos nombrados anteriormente, sólo GUM y SENSUS recogen información en unidades gramaticales mayores que las palabras. Otras ontologías se centran en el significado de las palabras (Ej. WordNet). Por otra parte, en algunas de las ontologías hay una relación 'uno a uno' entre conceptos y palabras en un lenguaje natural (Ej. wordnets de EuroWordNet), mientras que en otras, muchos conceptos pueden no tener relación con ninguna palabra en un lenguaje o pueden tener relación con más de una en el mismo lenguaje (Ej. Mikrokosmos).

También hay diferencias con respecto a su grado de dependencia en el lenguaje; algunas ontologías lingüísticas dependen totalmente de un único lenguaje (Ej. WordNet); otras son plurilingües, es decir, que son válidas para varios lenguajes (Ej. GUM); algunas otras contienen una parte dependiente del lenguaje y una parte independiente del lenguaje (Ej. EuroWordNet); y otras son independientes del lenguaje (Ej. Mikrokosmos).

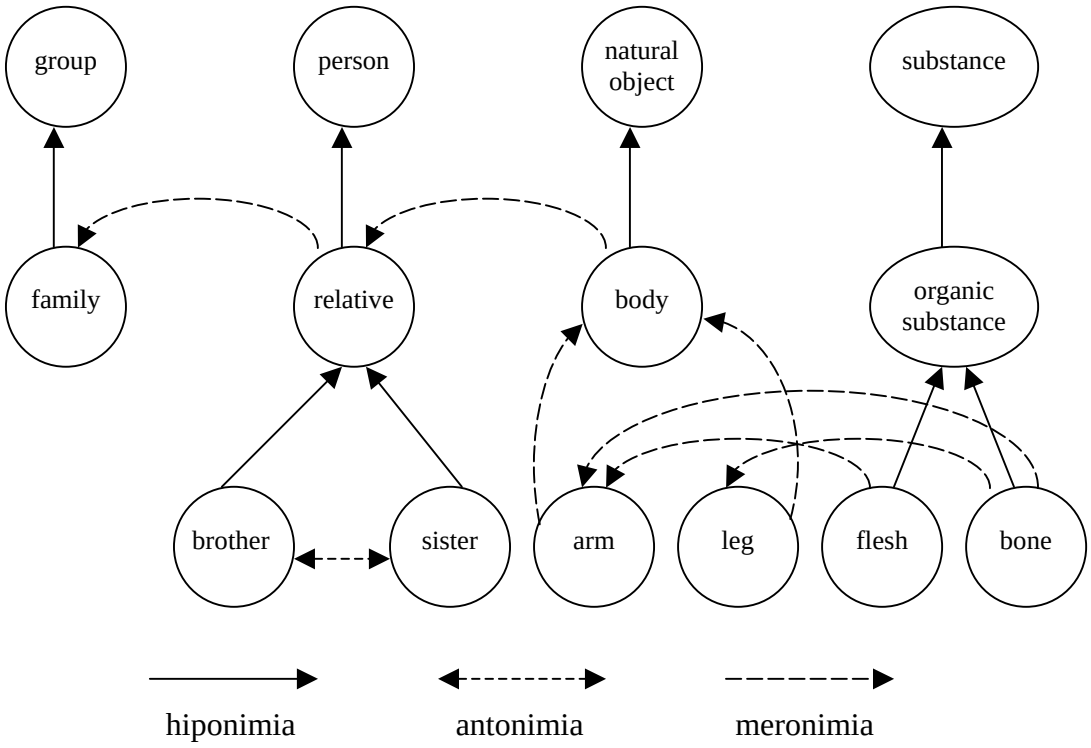
El origen y motivaciones de estas ontologías son variados y de este modo tenemos: bases de datos léxicas *online* (Ej. WordNet), ontologías para traducción a máquina (Ej. SENSUS), ontologías para generación de lenguaje natural (Ej. GUM), etc.

5.4.2- WordNet

WordNet (Miller et al., 1990; Miller, 1995) es una base de datos léxica muy extensa para la lengua inglesa, creada en la *Princeton University* y basada en teorías psicolingüísticas. La psicolingüística es un campo interdisciplinario de investigación interesado en las bases cognitivas de la capacidad lingüística (Fellbaum y Miller, 1990). WordNet es considerado el recurso más importante disponible para los investigadores en lingüística computacional, análisis de textos, y muchas áreas relacionadas. Intenta organizar información léxica en cuanto a significados de palabras y no en cuanto a

clases de palabras, aunque también se considera la morfología inflexional. Por ejemplo, si buscamos *trees* en WordNet, obtendremos el mismo acceso que si buscamos *tree*.

WordNet 1.7 contiene 121.962 palabras y 99.642 conceptos. Está organizado en 70.000 conjuntos de sinónimos (*synsets*), cada uno representando un concepto léxico fundamental. Los *synsets* están vinculados mediante relaciones tales como sinonimia y antonimia, hiperonimia e hiponimia ('subclase de' y 'superclase de'), meronimia y holonimia ('parte de' y 'tiene un'). Aproximadamente la mitad de los *synsets* incluye breves explicaciones de su sentido intuitivo en inglés. WordNet divide el léxico en cinco categorías: nombres, verbos, adjetivos, adverbios, y palabras de función. Los nombres están organizados en jerarquías de tópicos. La siguiente figura muestra parte de la jerarquía de nombres, donde términos acerca de una persona (sus componentes, sus substancias, la organización de su familia) aparecen relacionados. Las únicas relaciones que podemos ver en la figura son: meronimia, antonimia y hiponimia, ya que es una vista muy reducida de la jerarquía de nombres.



Vista parcial de la categoría de nombres de WordNet

Los verbos están organizados de acuerdo con una variedad de relaciones de implicación. Por ejemplo, los verbos *succeed* y *try* están relacionados mediante una 'implicación hacia atrás', y *buy* y *pay* están relacionados mediante una inclusión temporal. En cuanto a los adjetivos y adverbios, las relaciones de semejanza y antonimia juegan un papel importante. Por ejemplo, *dry* está relacionado con *sere*, *anhydrous*, *arid*, etc., mediante la relación de semejanza. *Wet* también está relacionado con *humid*, *watery* o *damp*

mediante la relación de semejanza. Además, *dry* y *wet* están relacionados por medio de la relación de antonimia.

5.4.3- Utilidad en el proyecto

En el proyecto utilizamos esta herramienta para llevar a cabo dos funciones muy importantes: categorización de palabras y tratamiento de sinónimos.

5.4.3.1- Categorización de palabras

Consiste en comprobar la categoría gramatical de las nuevas palabras que entrarán a formar parte de la ontología y seleccionar únicamente aquellas que son nombres o adjetivos. La razón de escoger los nombres y adjetivos es para obtener realmente subclases de la clase inicial. Por ejemplo, si creamos la ontología a partir de *cancer*, se consideran subclases *breast cancer* o *advanced cancer* (seleccionando el nombre *breast* o el adjetivo *advanced* de entre las posibles palabras), pero en cambio no consideramos como subclase *having cancer* (rechazaríamos el verbo *having*).

5.4.3.2- Tratamiento de sinónimos

Esta función se realiza gracias a los *synsets* (conjuntos de sinónimos) en los que organiza *WordNet* las palabras. Lo que hacemos es:

- Comprobar si existe una relación de sinonimia entre palabras de una misma clase. Por ejemplo, no se aceptarán clases como *warfare civil war*, ya que *warfare* y *war* son palabras sinónimas.

- Comprobar si dos clases se diferencian únicamente por dos palabras sinónimas. Por ejemplo, las clases *west war* y *occident war*, ya que *west* y *occident* son sinónimos; y las clases *skin extended cancer* y *prolonged pelt cancer*, ya que se diferencian por dos parejas de sinónimos: *skin* / *pelt*, y *extended* / *prolonged*.

6- DISEÑO DEL SISTEMA

6.1- Descripción detallada de la entrada y salida

Mediante una interfaz gráfica, se pedirá al usuario que introduzca una serie de parámetros, necesarios para la construcción de la ontología:

- El concepto básico a partir del cual desea construir la ontología. Este concepto debería ser en inglés, ya que el sistema se ha implementado pensando sólo en esa lengua.
- El máximo nivel de profundidad que podrá tener la ontología.
- El número máximo de páginas web a considerar (analizar) de cada clase.
- El número máximo de subclases que tendrá una clase.
- La forma de decidir cuando se genera una nueva clase. Las opciones son: el porcentaje de páginas web o el número de páginas web en que aparezca.
- El porcentaje de páginas web a partir del cual se genera una nueva clase.
- El número de páginas web a partir del cual se genera una nueva clase.

El concepto básico a partir del cual se generará la ontología, por razones obvias, es un parámetro que debe introducir el usuario obligatoriamente. El resto de parámetros son opcionales, ya tienen asignado un valor por defecto.

Basándose en estos parámetros, se construirá la ontología en lenguaje OWL y se almacenará en un archivo para que pueda ser visualizada por alguna herramienta de visualización de ontologías, por ejemplo Protégé, que es la que hemos utilizado para visualizar los resultados que obteníamos.

Además, el usuario también podrá visualizar la ontología en la interfaz gráfica del sistema, y no sólo cuando haya finalizado su construcción, sino que a medida que se van generando nuevas clases, ya se muestran en la interfaz gráfica junto con sus páginas web asociadas.

6.2- Justificación del uso de un SMA

Para resolver el problema hemos decidido diseñar un Sistema Multi-Agente, ya su utilización proporciona una serie de ventajas muy importantes:

- Modularidad: en lugar de tener un sólo programa que se encargue de hacer todo el trabajo, utilizando un SMA podemos dividir este trabajo en servicios menos complejos y asignárselos a agentes distintos. En nuestro sistema, por ejemplo, un agente puede encargarse de analizar información y otro de construir la ontología.

- Eficiencia: en un SMA los agentes se pueden ejecutar paralelamente (en máquinas diferentes) o concurrentemente (en la misma máquina), mejorando el tiempo de cálculo en cualquier caso. En nuestro sistema podrá haber, por ejemplo, varios agentes que realicen búsquedas y análisis de páginas web a la vez, lo que es una grandísima ventaja, sobretodo a medida que alcanzamos mayor profundidad en la ontología que vamos creando.

- Flexibilidad: en un SMA se pueden crear o eliminar agentes según las necesidades de la aplicación. En nuestro ejemplo esto es de gran ayuda, ya que los parámetros de entrada son totalmente variables (los decide el usuario), y por lo tanto, la cantidad de trabajo a realizar variará también; por lo que nos interesa poder crear más o menos agentes dinámicamente dependiendo de eso, de la cantidad de trabajo que haya que realizar.

- Cooperación: un SMA permite que los diferentes agentes puedan comunicarse e intercambiar información. En nuestro caso, un agente podría encargarse de analizar y seleccionar información para después proporcionársela a otro que la utilice para construir la ontología.

6.3- Arquitectura del SMA

Para resolver el problema hemos decidido implementar un Sistema Multi-Agente con dos tipos de agentes:

- Agente constructor
- Agente buscador

Sólo será necesario un agente constructor a lo largo de todo el proceso. El número de agentes buscadores, en cambio, variará en función de los parámetros introducidos por el usuario, concretamente dependerá del número de búsquedas a realizar. Por este motivo, sólo se encuentra inicialmente en ejecución el agente constructor, que es el encargado de ir generando los agentes buscadores dinámicamente.

Las funciones del agente constructor serán las siguientes:

- Crear la interfaz gráfica para pedir los parámetros necesarios al usuario y para mostrarle el resultado obtenido (jerarquía de clases creada y páginas web asociadas a cada clase).
- Generar los agentes buscadores precisos, así como destruirlos cuando hayan realizado su trabajo.
- Comunicarse con cada agente buscador para informarle de la acción que debe realizar y para obtener el resultado de esta acción.
- Analizar la información que le envían los agentes buscadores y desechar parte de ella si fuese necesario.
- Ir construyendo la ontología a medida que va recibiendo la información necesaria por parte de los agentes buscadores hasta que se llegue al nivel de profundidad determinado por el usuario o a un nivel en el que no seamos capaces de hacer una nueva descomposición.

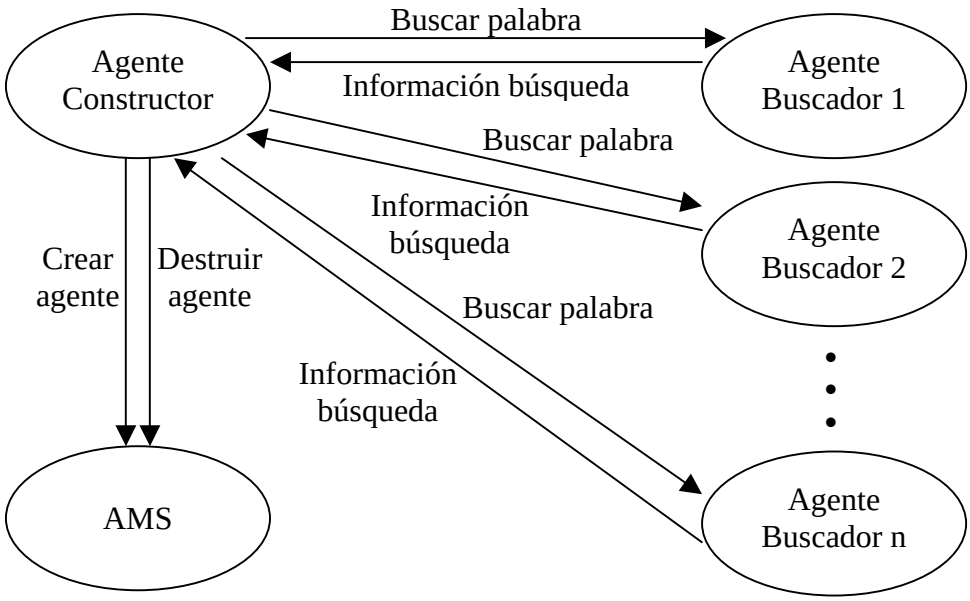
Y estas serán las funciones de cada agente buscador:

- Buscar en el *Google* las páginas web asociadas al nombre de la clase recibida por parte del agente constructor.
- Analizar las páginas obtenidas y seleccionar las subclases que se generarán a partir de esa clase:
    - Mirar qué palabras acompañan a la palabra clave.

- Seleccionar los nombres y adjetivos.
- Comprobar en cuantas páginas aparece cada palabra seleccionada.
- Seleccionar las más frecuentes.

- Enviar la información al agente constructor (las nuevas subclases de la ontología y las páginas web asociadas a cada una de ellas).

El gráfico general de la arquitectura del sistema multi-agente sería el siguiente:



6.4- Funcionamiento del SMA

Cuando el agente constructor obtiene por parte del usuario el concepto a partir del cual se construirá la ontología (Ej. *cancer*), crea al primer agente buscador para que realice la búsqueda de esta primera clase y la añade ya a la ontología.

El agente buscador busca en el *Google* el nombre de clase y para ello utiliza las funciones que ofrece el *API* de *Google*, descartando los resultados que correspondan a archivos '.doc' o '.pdf '.

Cuando obtiene las páginas web extrae el contenido de cada una mediante la herramienta *HTMLParser*, y va analizando el texto separándolo por palabras gracias al analizador *StopAnalyzer* de la herramienta *Lucene*, que ya se encarga de ignorar todo lo que no sean letras (Ej. símbolos y números) y las *stop words*.

Después, el agente buscador realiza una serie de comprobaciones sobre cada una de las palabras del texto extraído de la página web, y cuando alguna palabra no cumple alguna de las condiciones, la desecha sin necesidad de seguir efectuando el resto de comprobaciones sobre ella. Son las siguientes:

- Si la palabra es inmediatamente anterior al nombre de la clase de la que se ha hecho la búsqueda.
- Si no es demasiado corta. Hemos decidido que la longitud mínima de las palabras seleccionadas sea de tres caracteres, ya que poniendo dos nos salían peores resultados y poniendo cuatro perdíamos algunas palabras interesantes de tres caracteres.
- Si la palabra no se encuentra ya en el nombre de la clase.
- Si el nombre de la clase no contiene ya ninguna palabra de la misma familia (con la misma raíz) que la que se está analizando. Esta comprobación se realiza con ayuda de la herramienta *Snowball*.
- Si la palabra es un nombre o un adjetivo. Esta comprobación se realiza mediante la herramienta *WordNet*.
- Si el nombre de la clase no contiene ninguna palabra sinónima a la que se está analizando, tanto en sus diferentes significados como nombre, como en sus diferentes significados como adjetivo. Esta comprobación se realiza gracias a la herramienta *WordNet*.

Cuando finalizan las comprobaciones el agente tiene que seleccionar, de entre las palabras que han cumplido todas las condiciones, las *n* que han aparecido un mayor número de veces en las páginas web, siendo *n* un factor que decide el usuario, y que será además el número de subclases máximo que tendrá cada clase.

Pero aún hay una última condición que deben cumplir estas *n* palabras, y es que tienen que haber aparecido en un número mínimo de páginas web o en un porcentaje mínimo del total de páginas web analizadas, según el criterio que haya elegido el usuario.

Una vez que el agente buscador tiene las palabras definitivas (Ej. *breast* y *prostate*) y, por lo tanto, las nuevas subclases que se crearán en nuestra ontología (Ej. *breast cancer* y *prostate cancer*), le envía al agente constructor la siguiente información:

- Las nuevas subclases que debe añadir a la ontología.
- Las páginas web que ha obtenido al buscar la clase padre y en las que no aparece ninguna de las nuevas subclases. Por ejemplo, si ha buscado la clase *cancer* y ha obtenido las subclases *breast cancer* y *prostate cancer*, enviará como páginas web asociadas a la clase padre todas las que ha analizado donde aparezca *cancer* pero no *breast cancer* ni *prostate cancer*.

Cuando el agente constructor recibe información por parte del agente buscador, lo destruye y analiza una a una las nuevas clases que le ha enviado, eliminando las que son equivalentes a alguna que hubiera recibido con anterioridad, por alguna de las siguientes razones:

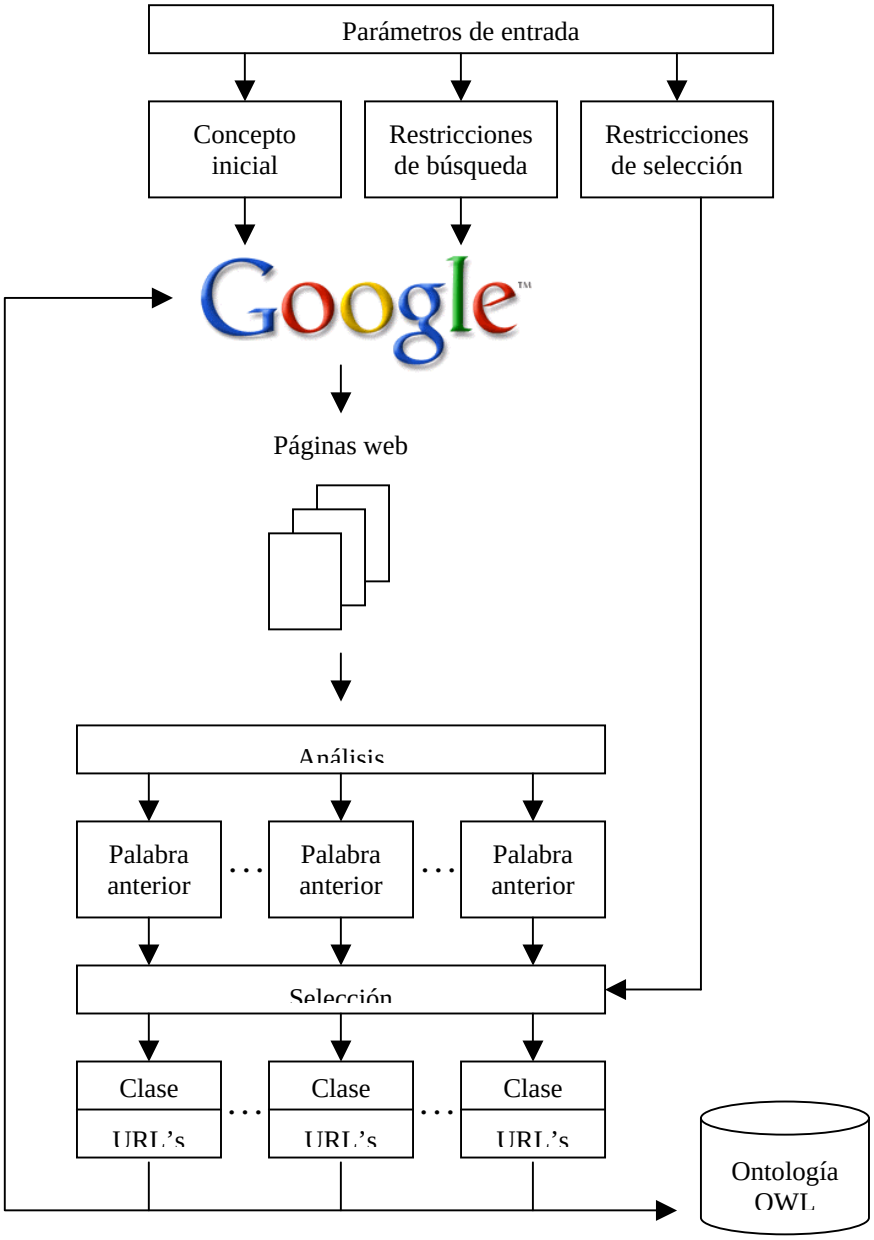
- Porque contienen las mismas palabras pero con diferente orden.
- Porque se distinguen únicamente por palabras de la misma familia (con la misma raíz). Esta comprobación la realiza con la ayuda de la herramienta *Snowball*.
- Porque se distinguen únicamente por palabras que son sinónimas, ya sea en alguno de sus diferentes significados como nombre o en alguno de sus diferentes significados como adjetivo. Esta comprobación la realiza con la ayuda de la herramienta *WordNet*.

Después de realizar las comprobaciones, el agente constructor añade a la ontología las subclases definitivas, las relaciones de parentesco entre ellas y su clase padre, y las páginas web asociadas a cada una.

A continuación, el agente constructor crea tantos agentes buscadores como nuevas clases se han creado, y le asigna una de estas clases a cada uno (en nuestro ejemplo, se crearán dos agentes buscadores, uno que se encargará de buscar *breast cancer* y otro que buscará *prostate cancer*). De este modo, volvemos al punto en que cada agente buscador busca en el *Google*, extrae el texto de las páginas web, lo analiza, etc.

El proceso finaliza cuando se ha llegado al nivel de profundidad determinado por el usuario o a un nivel en el que no somos capaces de hacer una nueva descomposición.

El siguiente gráfico resume de forma general el proceso que acabamos de explicar:



6.5- Comunicación entre agentes

6.5.1- Intercambio de mensajes entre agentes

El protocolo que utilizaremos para la comunicación (transmisión de mensajes) entre el agente constructor y los agentes buscadores, es el protocolo *FIPA-Request*. Este es, sin duda, el que mejor se adapta a las necesidades del proceso, ya que el agente constructor debe enviar a cada buscador un mensaje con la acción que debe realizar, junto con los parámetros necesarios para ello, y pretende recibir el resultado de esta acción. Y este protocolo sirve precisamente para eso, para pedir un servicio a un agente y obtener un resultado.

El agente constructor inicia un protocolo *FIPA-Request* con cada agente buscador enviándole un mensaje *Request* pidiendo como servicio que realice la acción *SearchWord*, es decir, para que realice la búsqueda y análisis de las páginas web.

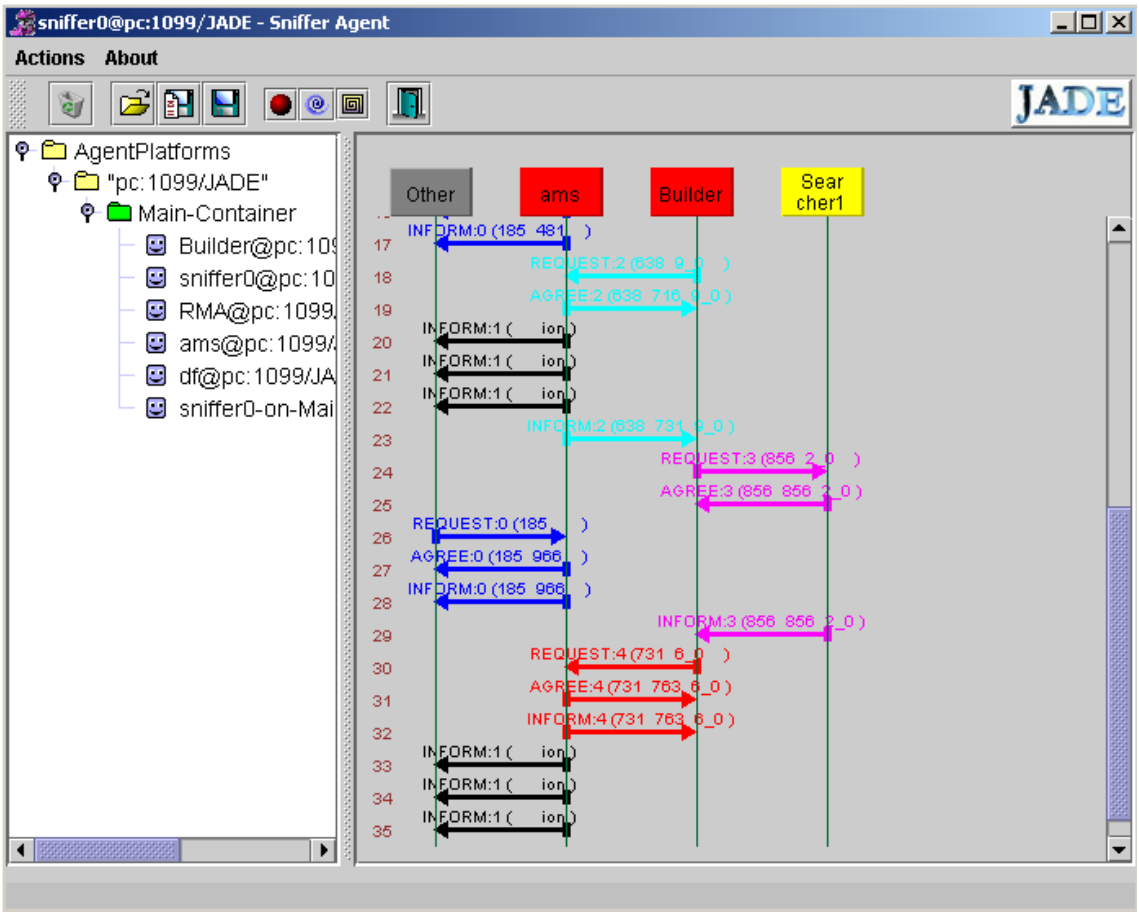
Cuando el agente buscador recibe el mensaje *Request* comprueba si entiende el contenido y puede realizar el servicio requerido (si la acción que contiene es *SearchWord*), en caso afirmativo envía al constructor un mensaje *Agree*. Finalmente, cuando ha realizado todo su trabajo envía al constructor un mensaje tipo *Inform* con el resultado de la acción, es decir, con la información necesaria para construir la ontología.

El agente constructor también se comunica con el agente AMS para que éste cree o elimine a uno de los agentes buscadores. El protocolo utilizado en este caso también es el *FIPA-Request*, ya que un agente pide a otro que realice un servicio determinado.

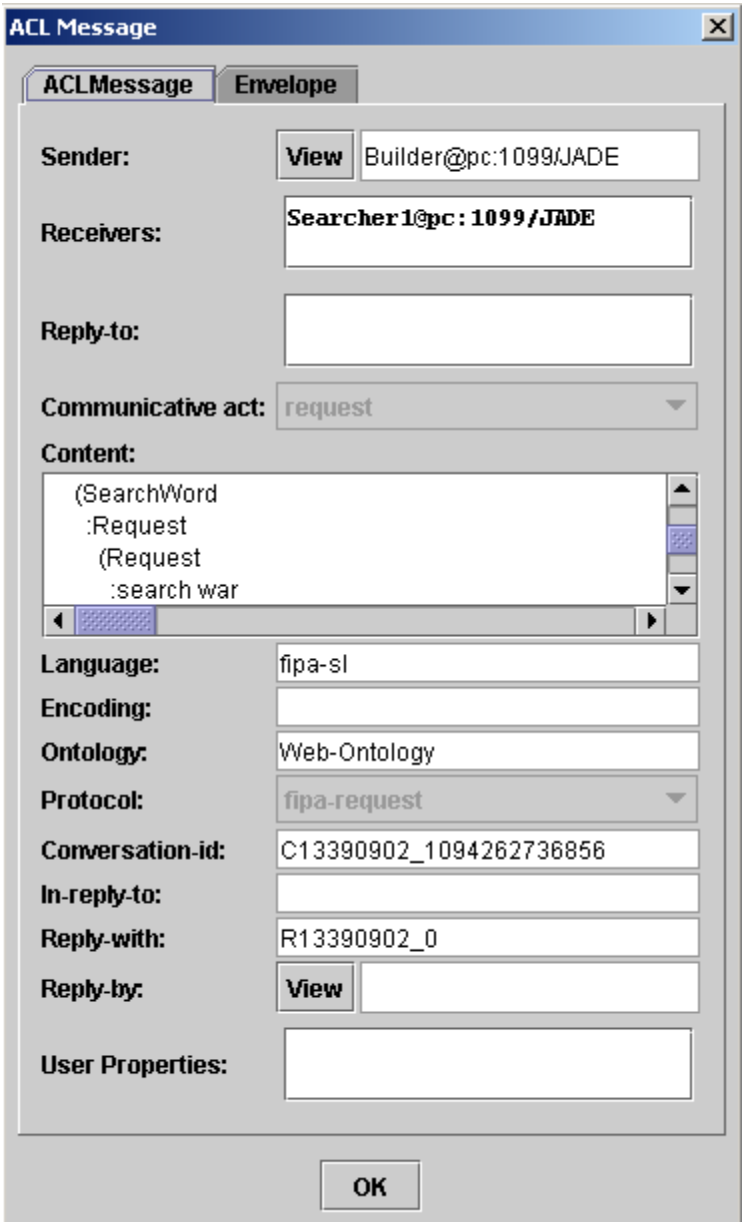
Este proceso será igual que el anterior: el agente constructor inicia el protocolo *FIPA-Request* y envía al agente AMS un mensaje *Request* pidiendo como servicio que realice la acción *CreateAgent* o *KillAgent*, dependiendo de si quiere que cree o destruya un agente buscador.

A continuación, el agente AMS envía al constructor un mensaje *Agree* si ha entendido el mensaje y cuando haya realizado el servicio le envía un mensaje de tipo *Inform* sin ninguna información adicional.

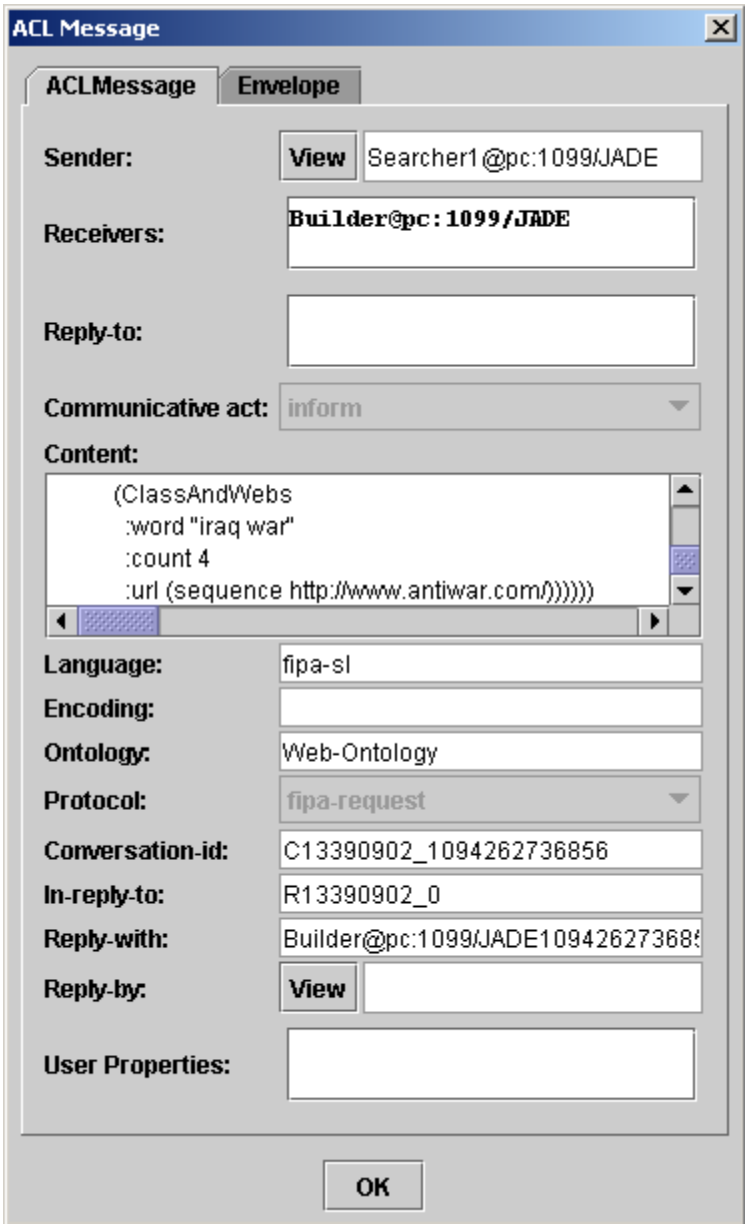
En la siguiente figura, que corresponde a la ventana del *sniffer* de JADE, podemos observar tres ejemplos de protocolo *FIPA-Request*, uno para cada una de las situaciones que acabamos de explicar. Primero, la comunicación entre el agente constructor (*Builder*) y el agente AMS para que este último cree un agente buscador (*Searcher1*). Luego, la comunicación entre el agente constructor (*Builder*) y el agente buscador (*Searcher1*) para que el segundo realice su servicio y envíe al primero la información necesaria. Y por último, la comunicación entre el agente constructor (*Builder*) y el agente AMS para que este último destruya al agente buscador (*Searcher1*).



A continuación se muestra una figura con la estructura de un mensaje de tipo *Request* que envía el agente constructor a un buscador. Si observamos los campos del mensaje (explicados en el apartado de agentes, 2.4.3.1 concretamente) vemos que el emisor (*sender*) es el agente constructor (*Builder*), que el único receptor (*receiver*) es el agente buscador (*Searcher1*), que el tipo de mensaje (*communicative act*) es *Request*, que el lenguaje utilizado (*language*) es el FIPA-sl y la ontología utilizada en la comunicación (*ontology*) es nuestra ontología, Web-Ontology. Además, en el campo del contenido (*content*) se puede ver el servicio requerido, *SearchWord*, y el concepto que se tiene que buscar, *war*.



Y en la siguiente figura lo que observamos es la estructura de un mensaje de tipo *Inform* que envía un agente buscador al constructor. En este caso el emisor (*sender*) es el agente buscador (*Searcher1*), el único receptor (*receiver*) es el agente constructor (*Builder*), el tipo de mensaje (*communicative act*) es *Inform*, el lenguaje utilizado (*language*) es el FIPA-sl y la ontología utilizada en la comunicación (*ontology*) es nuestra ontología, Web-Ontology. En el campo del contenido (*content*) se puede ver parte de la información para construir la ontología, concretamente observamos la clase *iraq war*, las veces que ha aparecido y una de las páginas web donde se ha encontrado.



6.5.2- Ontología del SMA

La ontología del SMA es la ontología que se ha utilizado en la comunicación entre los agentes. A continuación detallaremos los diferentes conceptos, predicados y acciones que contiene la ontología.

6.5.2.1- Conceptos

- ClassAndWebs: estructura de datos que contiene el nombre de una clase, el número de veces que se ha encontrado en la búsqueda realizada y sus páginas web asociadas.

Nombre propiedad	**Tipo**
word	String
count	int
url	ArrayList

- Request: estructura de datos que contiene toda la información introducida por el usuario (concepto a partir del cual se creará la ontología, nivel máximo de la ontología, número máximo de páginas web a tratar en cada búsqueda, número máximo de subclases a generar para cada clase, porcentaje de páginas web a partir del cual se genera una nueva clase, número de páginas web a partir del cual se genera una nueva clase y elección del método a seguir para decidir cuando se genera una nueva clase) y que envía el agente constructor a cada uno de los agentes buscadores.

Nombre propiedad	**Tipo**
search	String
level	int
numberMaxWebs	int
numberOfSubclasses	int
percentageOfWebs	int
numberOfWebs	int
percentageOrNumber	boolean

- Inform: estructura de datos que contiene una lista de elementos del tipo ClassAndWebs que es el resultado que un agente buscador de vuelve al constructor cuando ha realizado su trabajo.

Nombre propiedad	**Tipo**
subClasses	ArrayList

6.5.2.2- Predicados

En la ontología utilizada no es necesario ningún predicado, ya que cuando el agente constructor (iniciador del protocolo *FIPA-Request*) envía un mensaje, no espera simplemente que el agente buscador le devuelva información, sino que pretende que realice una acción.

6.5.2.3- Acciones

- SearchWord: acción que debe realizar todo agente buscador cuando recibe un mensaje del tipo *Request* del agente constructor, y que consiste en buscar una palabra en el *Google*, analizar las páginas web obtenidas y devolver al agente constructor la información necesaria mediante un mensaje del tipo *Inform* para que vaya creando la ontología. Esta acción contiene un concepto del tipo *Request*.

Nombre propiedad	**Tipo**
req	Request

7- IMPLEMENTACIÓN

El código del proyecto está dividido en dos paquetes: Agents y Ontology.

7.1- Paquete Agents

En él se encuentra el código fuente de los dos tipos de agentes (BuilderAgent.java y SearcherAgent.java), además de una estructura de datos necesaria para la ejecución de estos (SearchAndLevel.java), y el código de la interfaz gráfica del sistema (OntologyBuilderGUI.java).

7.1.1- BuilderAgent.java

Lo primero que hace el agente constructor es registrar el lenguaje (*FIPA-SL*) y la ontología (*WebOntology*) utilizados en la comunicación entre los agentes.

Después crea un objeto *OntologyBuilderGUI* para mostrar la interfaz gráfica al usuario, y cuando éste presione el botón para empezar a construir la ontología, el agente obtiene todos los parámetros introducidos y añade el comportamiento *BuilderFirstBehaviour*.

En el comportamiento *BuilderFirstBehaviour* crea la ontología en *OWL* y añade ya la clase raíz que ha introducido el usuario. Además, añade el comportamiento *AMSClientBehaviour* para enviar un *Request* al agente *AMS* con la acción *CreateAgent* y que este cree el primer agente buscador.

Cuando recibe el *Inform* del agente AMS, es decir, cuando el agente buscador ha sido creado, el agente constructor prepara un mensaje *Request* con una instancia de la acción *SearchWord* y el nombre de la clase raíz, que será la que tendrá que buscar el primer agente buscador, e inicia un protocolo *FIPA-Request* añadiendo el comportamiento *RequestInitiator*, para enviar el *Request* al buscador.

Cuando recibe el *Inform* del agente buscador, extrae el contenido y añade el comportamiento *AMSClientBehaviour* para enviar un *Request* al agente *AMS* con la acción *KillAgent* y que este destruya el agente buscador. Añade un nodo, con la información de la clase que acaba de ser tratada por el buscador, al árbol de la interfaz gráfica que representa la ontología que se está creando. Si en el resultado que le ha devuelto el agente buscador no hay ninguna subclase, es decir, si la clase que se acaba de tratar es una clase terminal de la ontología (una hoja), añade el comportamiento *BuilderThirdBehaviour*. Sino, recorre la lista de subclases y para cada una de ellas:

- Comprueba que no exista ya en la ontología alguna clase cuyo nombre contenga las mismas palabras pero con diferente orden, se distinga únicamente por palabras de la misma familia (con la misma raíz) o se distinga únicamente por palabras que son sinónimas, ya sea en alguno de sus diferentes significados como nombre o en alguno de sus diferentes significados como adjetivo.

- Si ya existe alguna clase ‘equivalente’ en la ontología, rechaza la subclase y sigue con el resto. Sino, añade a la ontología en OWL: la nueva subclase, la relación de

superclase-subclase con su clase padre correspondiente, y las páginas web relacionadas con la subclase como propiedades de clase. Antes de añadir el nombre de una clase sustituye los espacios en blanco por guiones, ya que Protégé no acepta espacios en blanco en los nombres de clases.

- Después comprueba si se ha llegado al nivel máximo de la ontología (parámetro introducido por el usuario), lo que significa que la subclase es una clase terminal de la ontología (una hoja) y en caso afirmativo añade un nodo con su información al árbol de la interfaz gráfica.

- Finalmente, añade el comportamiento *BuilderSecondBehaviour* (uno por cada nueva subclase).

El comportamiento *BuilderSecondBehaviour* comprueba si se ha llegado al nivel máximo de la ontología. Si es así, y la subclase que toca tratar es la última de las que quedan 'pendientes', significa que el proceso de construcción de la ontología ha terminado, por lo tanto crea el fichero '.owl' y almacena en él la ontología en *OWL*. Si no se ha llegado al nivel máximo, añade el comportamiento *AMSClientBehaviour* para enviar un *Request* al agente *AMS* con la acción *CreateAgent* y que este cree un agente buscador para que se encargue de la subclase que toca tratar. De este modo se vuelve al punto en el que el agente constructor recibe el *Inform* del *AMS*, envía un *Request* al buscador, etc., que ya hemos explicado anteriormente.

El comportamiento *BuilderThirdBehaviour* comprueba si quedan subclases 'pendientes'; en caso contrario significa que el proceso de construcción de la ontología ha terminado, y por lo tanto crea el fichero '.owl' y almacena en él la ontología en *OWL*.

En resumen, el proceso de construcción de la ontología puede terminar de dos formas: cuando se llega al nivel máximo de la ontología y se tratan todas las subclases terminales (este caso lo detecta el *BuilderSecondBehaviour*), o cuando en algún nivel no se encuentran subclases de ninguna de las clases y no se puede desarrollar más la ontología (este caso lo detecta el *BuilderThirdBehaviour*).

7.1.2- SearcherAgent.java

Lo primero que hace el agente buscador es registrar el lenguaje (*FIPA-SL*) y la ontología (*WebOntology*) utilizados en la comunicación entre los agentes.

Después añade el comportamiento *RequestResponder* y cuando recibe un mensaje *Request* del agente constructor, comprueba que entiende el contenido (que contiene una instancia de *SearchWord*) y le envía un mensaje *Agree*.

A continuación, busca en el *Google* el nombre de la clase que contenía el *Request* (función *searchGoogle*). Para ello es necesario disponer de una clave que ofrece *Google* a los usuarios registrados y que permite realizar 1000 consultas diarias. Al realizar la búsqueda, pone el nombre de la clase entre comillas, esto es importante ya que puede estar formado por varias palabras y nos interesa encontrar las páginas web donde aparezcan juntas y en ese mismo orden.

Una vez obtiene las páginas web, el agente pasa al proceso de análisis del texto y selección de subclases (función *analysisWebs*). Este proceso ya se ha explicado con detalle en el apartado de 'Funcionamiento del SMA'.

Cuando ya ha seleccionado las subclases definitivas, envía un mensaje *Inform* con la información al agente constructor, que destruirá al agente buscador cuando lo reciba.

7.1.3- SearchAndLevel.java

Clase que contiene como propiedades un *string* y un entero, además de las funciones para asignarles un valor o consultar el que tienen; y que sirve para que el agente constructor pueda relacionar cada búsqueda con el nivel de la ontología al que pertenece.

7.1.4- OntologyBuilderGUI.java

Clase que se encarga de la interfaz gráfica del sistema, que consiste en un *JFrame* dividido en 3 partes:

- Arriba: un *JPanel* con varios *JLabel*, *JSpinner* no editables, *JComboBox* no editables, *JButton* y *JToolTip* explicativos, para que el usuario introduzca los parámetros.

- En el centro: un *JSplitPane*, con un *JTree* dentro de un *JSrollPane* a la izquierda que representará la ontología construida, y una *JTextArea* no editable dentro de un *JSrollPane* a la derecha donde se podrán ver las páginas web de la clase que se desee.

- Abajo: un *JPanel* con una *JLabel* que indicará el estado del proceso.

Además contiene *Listenners* para detectar cuando el usuario aprieta el *JButton* (que no se activa hasta que hay algo escrito en el campo de la palabra inicial de la ontología), y para detectar cuando se clica en algún nodo del *JTree*, y así mostrar las páginas web correspondiantes.

7.2- Paquete Ontology

En él se encuentra el código fuente de la ontología utilizada (WebOntology.java), de los conceptos (Request.java e Inform.java) y de la acción (SearchWord.java), y de las estructuras de datos necesarias (ClassAndWebs.java e IntegerComparator.java).

7.2.1- WebOntology.java

Clase que se encarga de añadir a la ontología utilizada en la comunicación entre los agentes de nuestro SMA, los conceptos *ClassAndWeb*, *Request* e *Inform*, y la acción *SearchWord*, junto con las propiedades de cada uno.

7.2.2- Request.java

Clase que representa un concepto de nuestra ontología y que contiene como propiedades un *string*, cinco enteros y un booleano, además de las funciones para asignarles un valor o consultar el que tienen. Sirve para almacenar los parámetros introducidos por el usuario y para que el agente constructor se los pueda enviar a un agente buscador en un mensaje de tipo *Request*.

7.2.3- Inform.java

Clase que representa un concepto de nuestra ontología y que contiene como propiedad una lista de *ClassAndWeb*, además de las funciones para asignarle un valor o consultar el que tiene. Sirve para que un agente buscador almacene la lista de nuevas subclases que ha seleccionado junto con sus páginas web asociadas y para que se lo pueda enviar al agente constructor en un mensaje de tipo *Inform*.

7.2.4- SearchWord.java

Clase que representa una acción de nuestra ontología y que contiene como propiedad un *Request*, además de las funciones para asignarle un valor o consultar el que tiene. Sirve para que el agente constructor pueda pedir a un agente buscador, por medio de un mensaje de tipo *Request*, que realice el servicio de buscar en *Google* y analizar las páginas web.

7.2.5- ClassAndWebs.java

Clase que contiene como propiedades un *string*, un entero y una lista de *strings*, además de las funciones para asignarles un valor o consultar el que tienen; y que sirve para que un agente buscador pueda almacenar cada nombre de clase junto con el número de veces que ha aparecido en las páginas analizadas y el nombre de todas las páginas web donde la ha encontrado. De este modo, cuando le envía la información al agente constructor, este puede añadir fácilmente a la ontología cada clase junto con sus páginas asociadas.

7.2.6- IntegerComparator.java

Clase que compara enteros y sirve para ordenar listas. En nuestro caso, sirve para que un agente buscador ordene la lista de posibles nuevas subclases según el número de veces que han aparecido en las páginas web analizadas, y así coger simplemente las *n* primeras, que serán las mas frecuentes.

8- EVALUACIÓN

Se han realizado varias pruebas para comprobar el funcionamiento del sistema, variando los parámetros introducidos.

A continuación comentaremos algunas de ellas, indicando para cada una los parámetros introducidos y mostrando la ontología obtenida por medio de capturas de pantalla tanto de la interfaz gráfica del sistema como del Protégé.

8.1- Cancer Ontology

Con el concepto *cancer* es con el que mejores resultados se han obtenido. Mostraremos imágenes de tres pruebas diferentes (cambiando los parámetros) y compararemos los resultados.

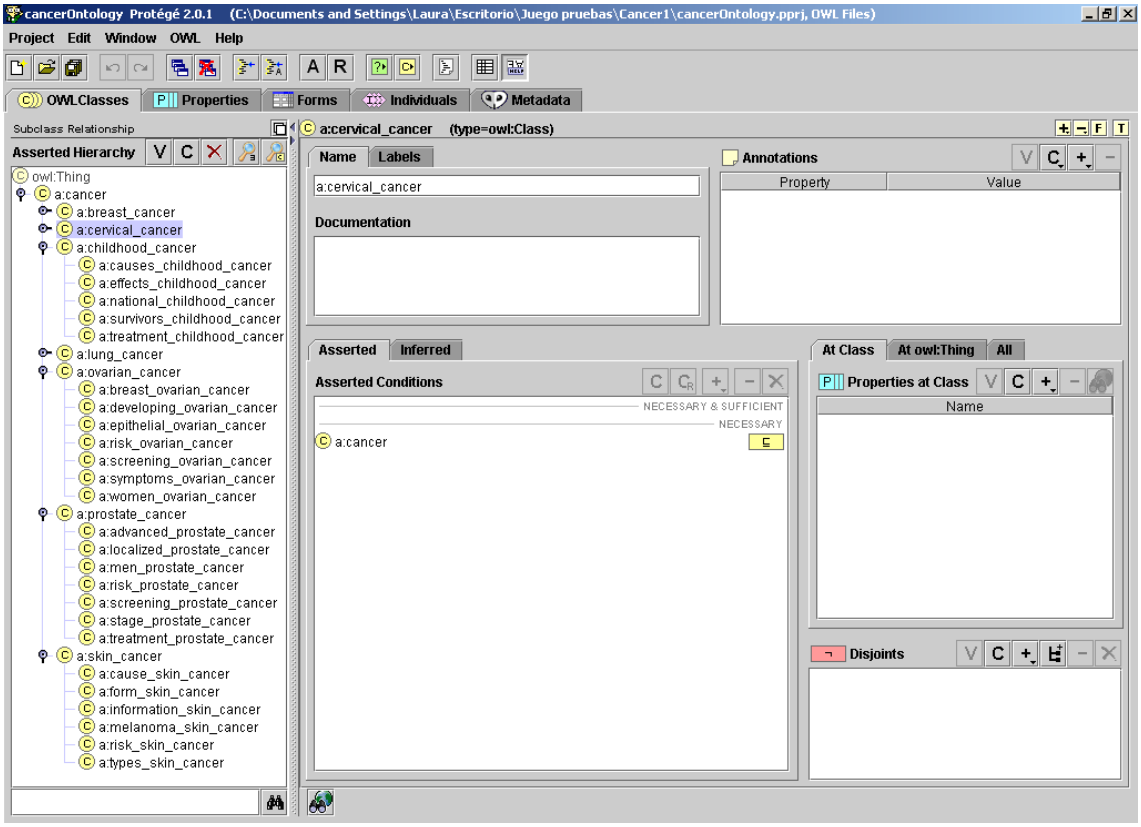
Prueba 1

- Parámetros de entrada:

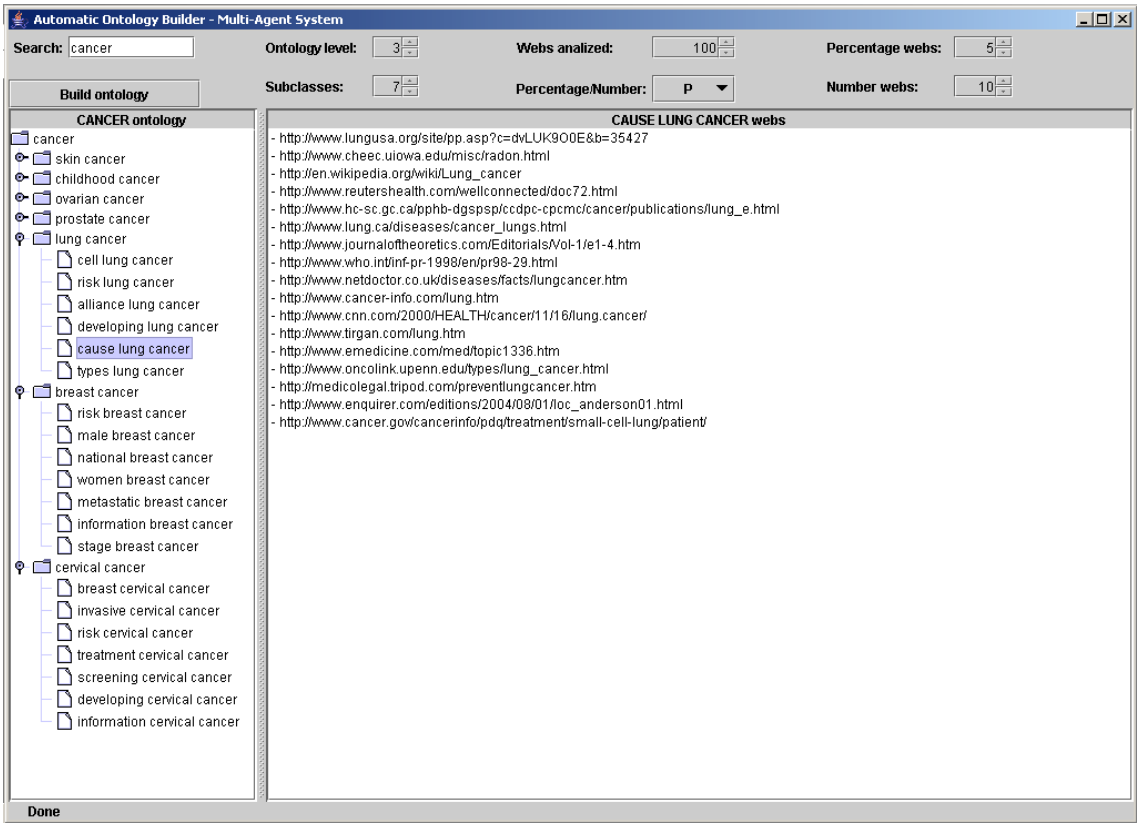
Concepto básico	**cancer**
Máximo nivel de la ontología	3
Número máximo de páginas a analizar por clase	100
Número máximo de subclases por clase	7
Porcentaje o número de páginas web	porcentaje
Porcentaje de páginas web mínimo para generar una nueva clase	5
Número de páginas web mínimo para generar una nueva clase	-------

Para poder ver todas las clases de la ontología mostramos cuatro de las subclases desplegadas en el Protégé y las otras tres en nuestra interfaz gráfica.

- Ontología en Protégé:



- Ontología en la interfaz gráfica del sistema:



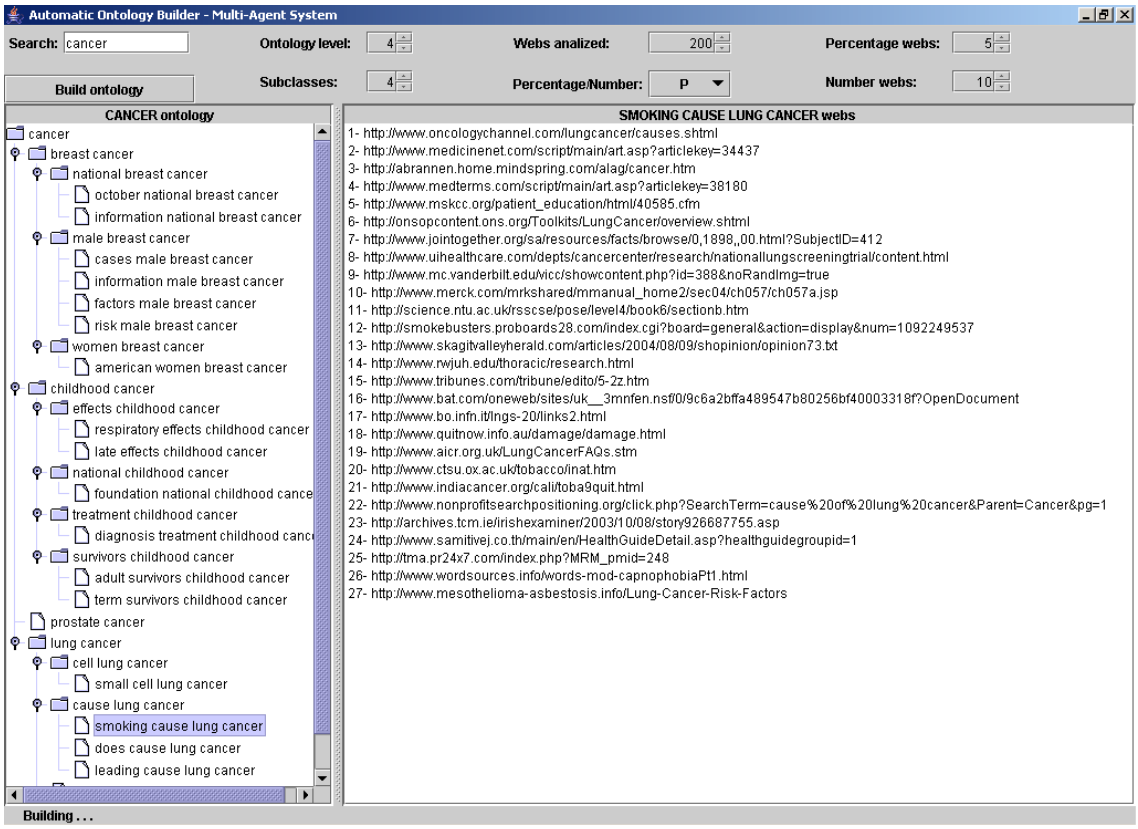
En esta prueba hemos obtenido un resultado muy bueno. Hemos puesto un valor bastante elevado de subclases por clase y, aún así, casi se han desarrollado todos los nodos posibles. Además, le hemos puesto que analizara sólo 100 páginas por clase, un valor algo pequeño, pero los subdominios generados son muy interesantes. Se ha divido el concepto 'cáncer' en siete tipos de cáncer y de cada uno de ellos se ofrece información relevante como causas, riesgo, tratamiento, efectos, etc..

Prueba 2

- Parámetros de entrada:

Concepto básico	**cancer**
Máximo nivel de la ontología	4
Número máximo de páginas a analizar por clase	200
Número máximo de subclases por clase	4
Porcentaje o número de páginas web	porcentaje
Porcentaje de páginas web mínimo para generar una nueva clase	5
Número de páginas web mínimo para generar una nueva clase	-------

- Ontología en la interfaz gráfica del sistema:



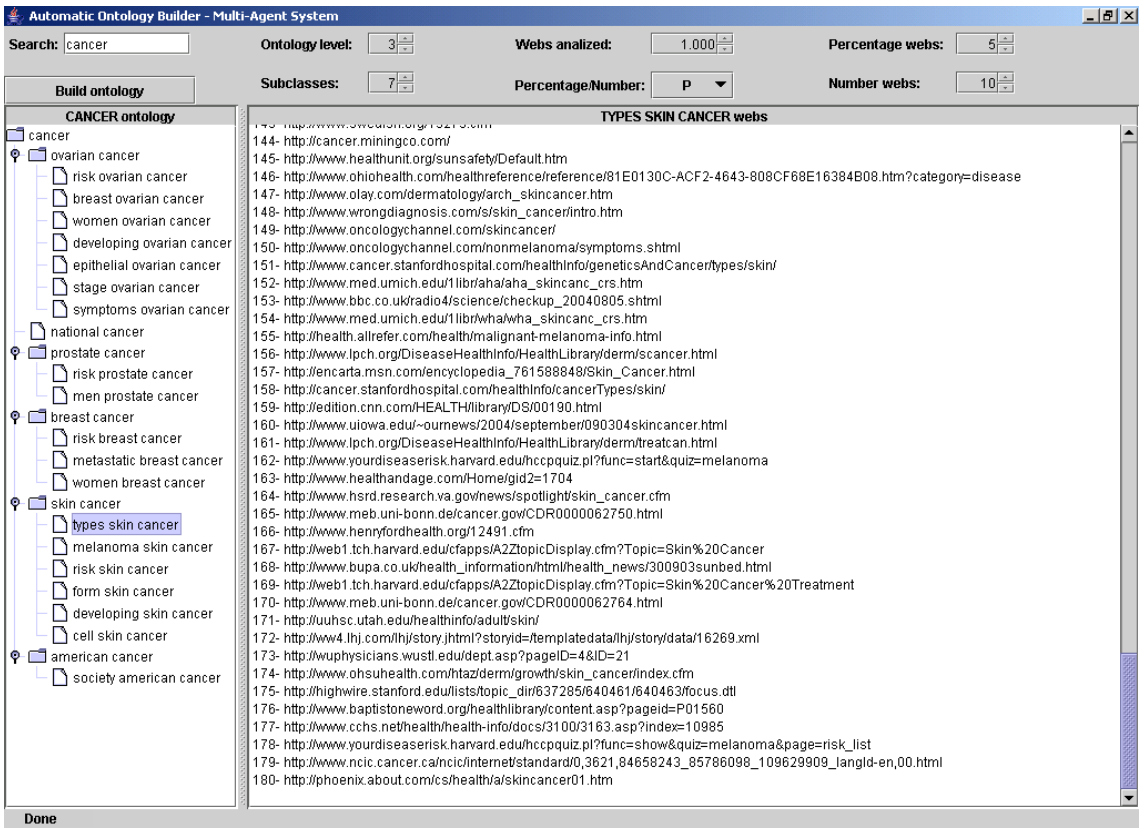
En esta prueba hemos podemos ver que, poniendo el nivel máximo a cuatro, también conseguimos buenos resultados para el término 'cáncer'. Es muy interesante comprobar que es posible realizar una división de la información hasta este punto (cuatro niveles es un valor bastante elevado) y encontrar clases interesantes como *factors male breast cancer*, *respiratory effects childhood cancer* o *smoking cause lung cancer*.

Prueba 3

- Parámetros de entrada:

Concepto básico	**cancer**
Máximo nivel de la ontología	3
Número máximo de páginas a analizar por clase	1000
Número máximo de subclases por clase	7
Porcentaje o número de páginas web	porcentaje
Porcentaje de páginas web mínimo para generar una nueva clase	5
Número de páginas web mínimo para generar una nueva clase	-------

- Ontología en la interfaz gráfica del sistema:



En este caso tenemos un resultado que no está mal del todo pero que si lo comparamos con la primera prueba que hemos hecho con 'cáncer' podemos ver que es bastante peor. Resulta difícil de creer porque en esta prueba se analiza un número 10 veces mayor de páginas por clase y el resto de parámetros son exactamente iguales. Pero la explicación más probable es que aunque en los dos casos el porcentaje mínimo sea del 5%, no es lo

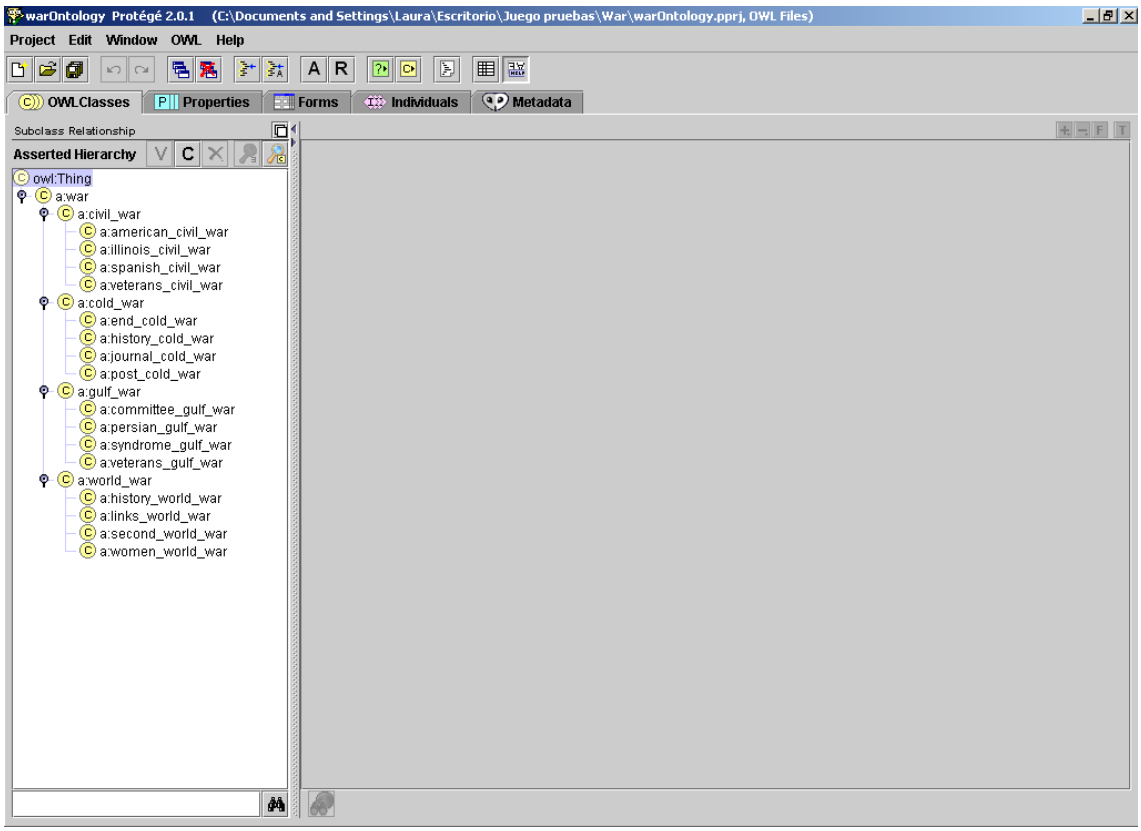
mismo el 5% de 100 que el 5% de 1000, es decir, que en la primera prueba una clase tenía que aparecer solamente en 5 páginas, y en esta tercera prueba, en cambio, una clase se tiene que encontrar en 50 páginas para ser generada, un valor muy alto, lo que seguro que hace que menos palabras pasen el 'filtro' y no se desarrolle más la ontología.

8.2- War Ontology

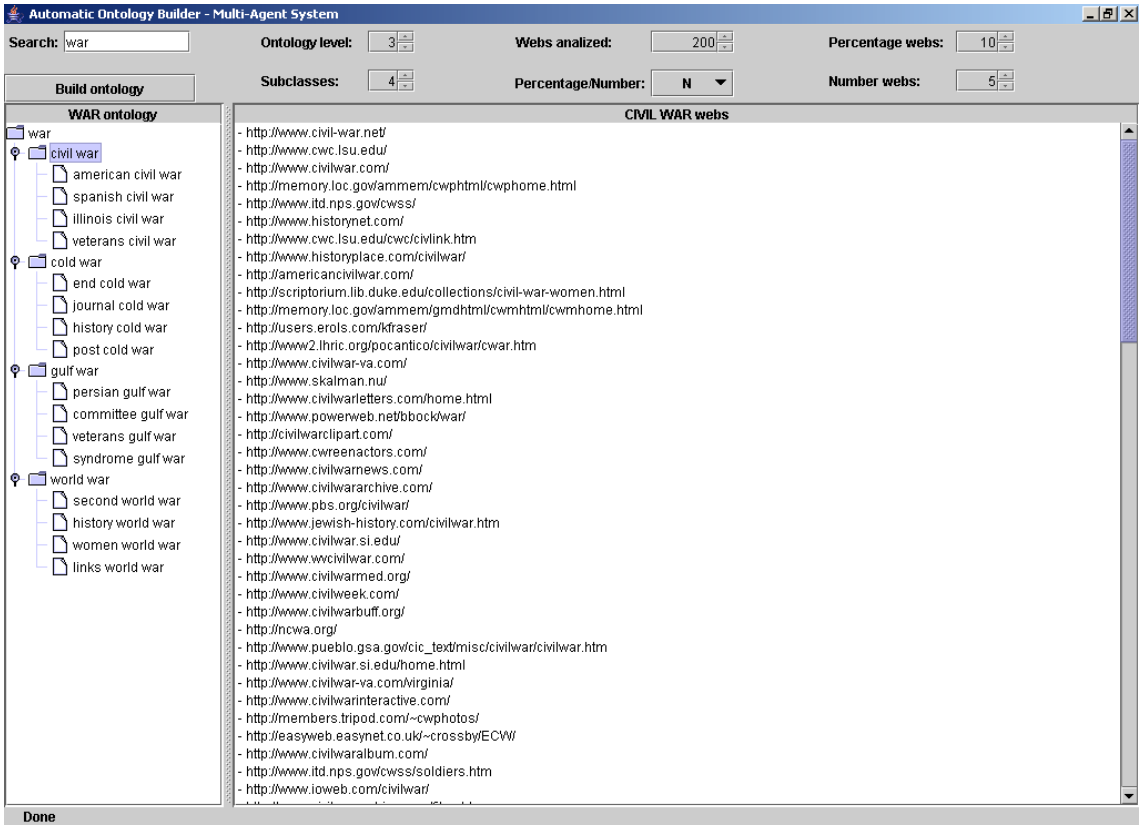
- Parámetros de entrada:

Concepto básico	**war**
Máximo nivel de la ontología	3
Número máximo de páginas a analizar por clase	200
Número máximo de subclases por clase	4
Porcentaje o número de páginas web	número
Porcentaje de páginas web mínimo para generar una nueva clase	-------
Número de páginas web mínimo para generar una nueva clase	5

- Ontología en Protégé:



- Ontología en la interfaz gráfica del sistema:



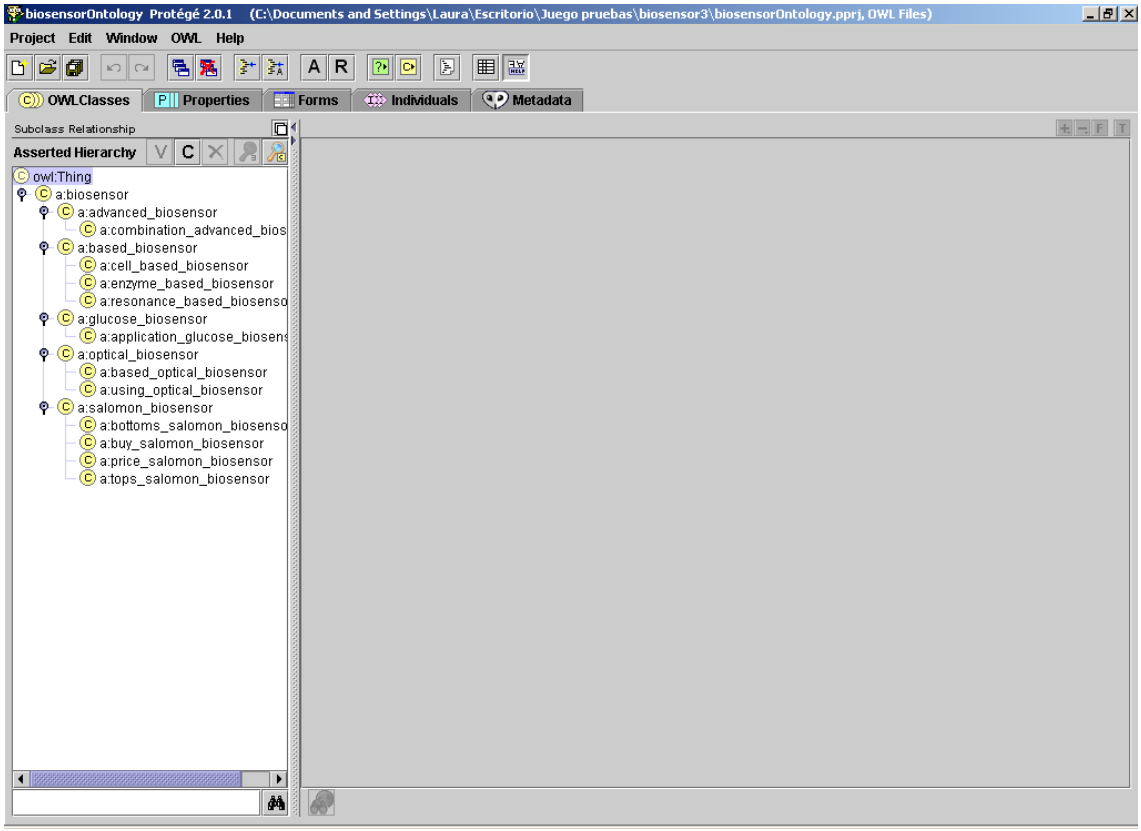
En esta prueba hemos obtenido un buen resultado, ya que se han desarrollado al máximo todos niveles y las subclases generadas estructuran bastante bien el dominio. Dividen el término ‘guerra’ en cuatro guerras concretas muy conocidas, y separan la información de cada una de ellas en diversas áreas (su historia, el lugar donde se produjeron, etc.). Y todo sin haber analizado un número muy elevado de páginas web.

8.3- Biosensor Ontology

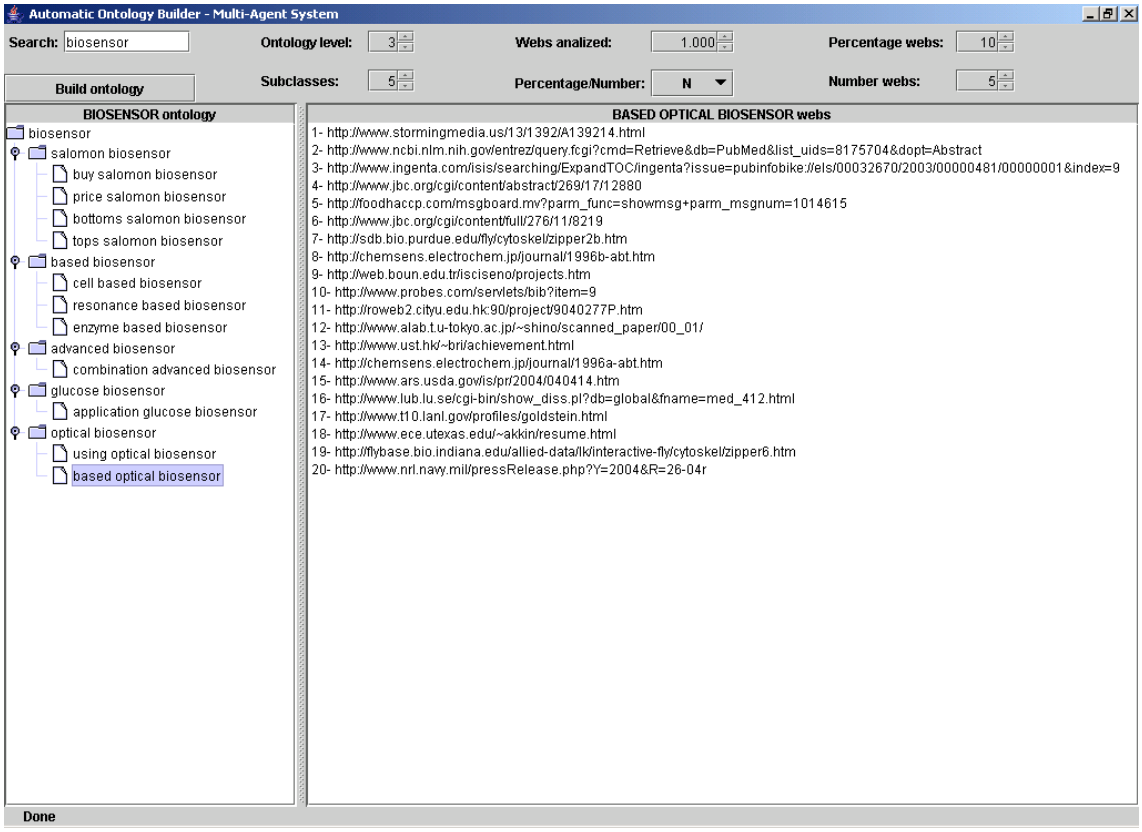
- Parámetros de entrada:

Concepto básico	**biosensor**
Máximo nivel de la ontología	3
Número máximo de páginas a analizar por clase	1000
Número máximo de subclases por clase	5
Porcentaje o número de páginas web	número
Porcentaje de páginas web mínimo para generar una nueva clase	-------
Número de páginas web mínimo para generar una nueva clase	5

- Ontología en Protégé:



- Ontología en la interfaz gráfica del sistema:



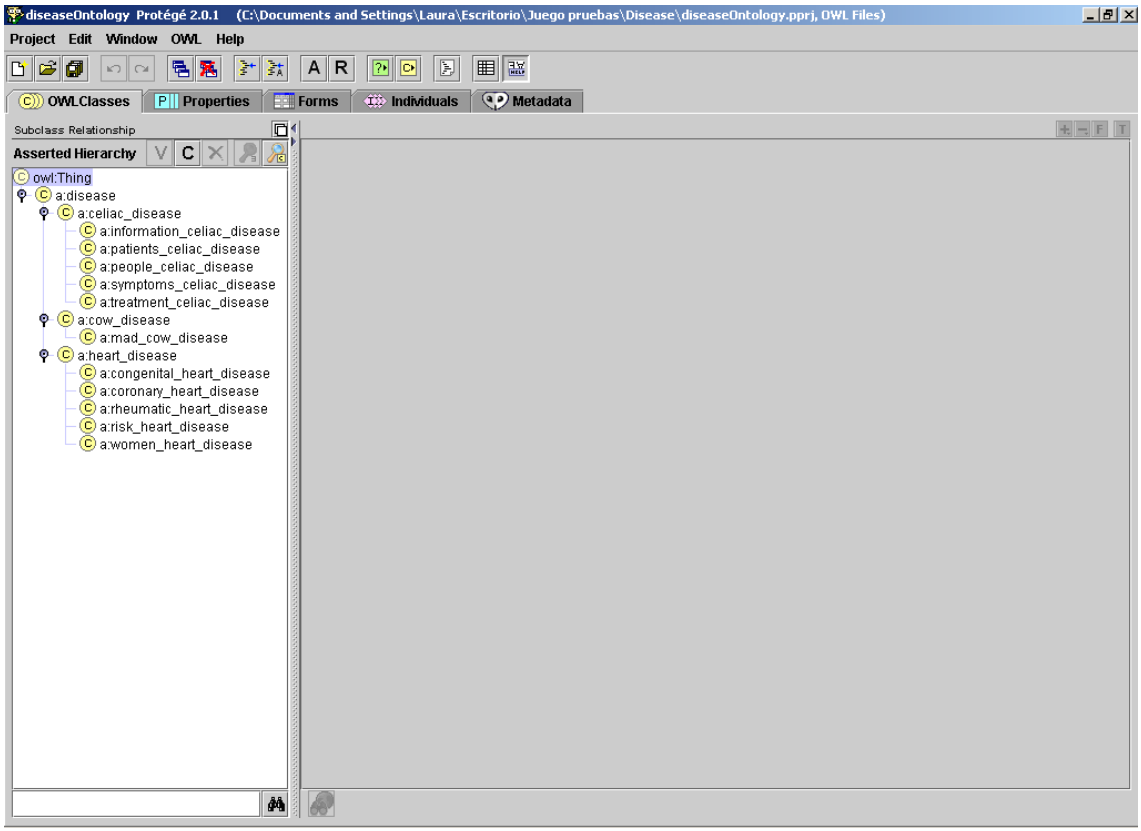
En esta prueba no hemos obtenido un resultado demasiado bueno, ya que no se ha desarrollado al máximo el último nivel y algunas subclases generadas contienen verbos (*using optical biosensor*), cosa intentamos evitar en este sistema. La causa es que el WordNet detecta muchos verbos como nombres también, aunque sólo correspondan a esa categoría en un significado muy concreto. Si comparamos el resultado con el anterior podemos ver que analizando cinco veces más páginas web por clase, obtenemos un resultado peor con este concepto. Por lo tanto, el resultado no depende solamente a los parámetros de control, sino que el tema escogido es un factor determinante.

8.4- Disease Ontology

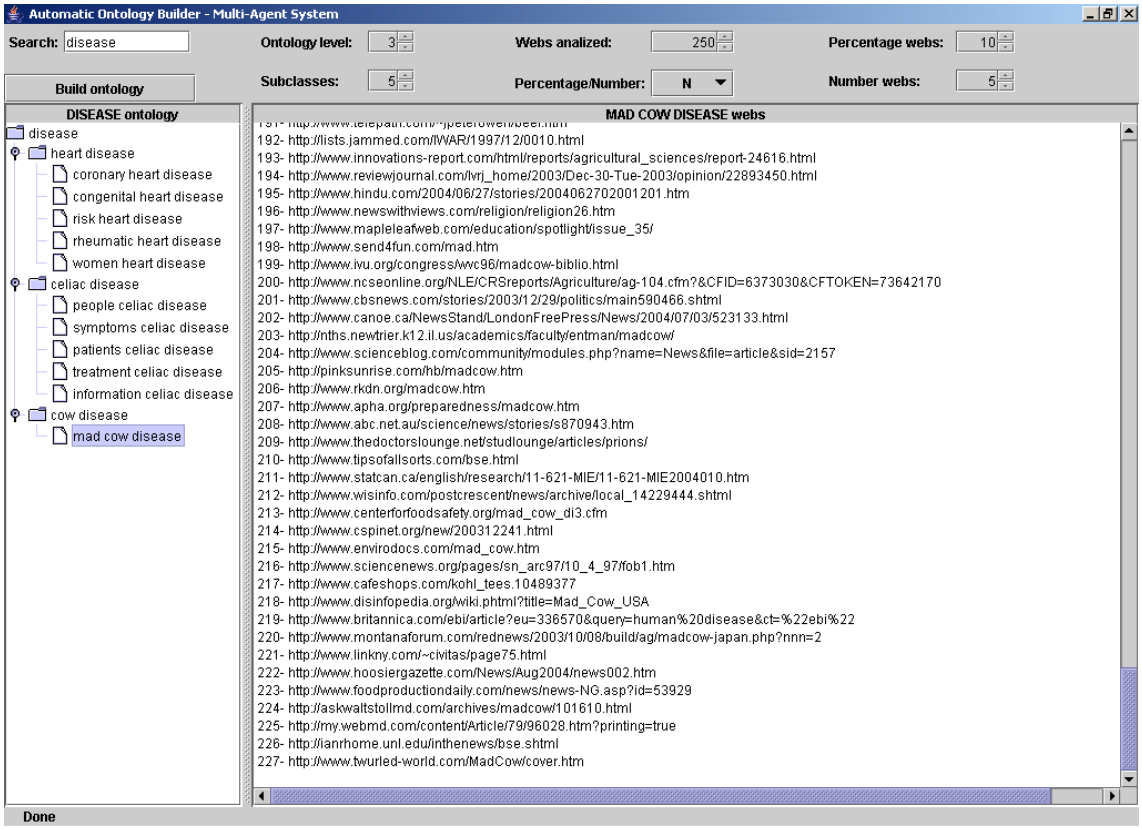
- Parámetros de entrada:

Concepto básico	**disease**
Máximo nivel de la ontología	3
Número máximo de páginas a analizar por clase	250
Número máximo de subclases por clase	5
Porcentaje o número de páginas web	número
Porcentaje de páginas web mínimo para generar una nueva clase	-------
Número de páginas web mínimo para generar una nueva clase	5

- Ontología en Protégé:



- Ontología en la interfaz gráfica del sistema:



En esta prueba no se han desarrollado al máximo los niveles, pero la información se ha estructurado bastante bien. Se ha separado el concepto ‘enfermedad’ en tres tipos de enfermedades y de las dos que más se han desarrollado se ha seleccionado información bastante interesante (riesgos, tratamiento, síntomas, etc.).

9- CONCLUSIONES Y TRABAJO FUTURO

Después de realizar un gran número de pruebas y de obtener los correspondientes resultados, considero que se ha alcanzado el objetivo inicial de este proyecto, y que es posible estructurar el conocimiento de forma 'inteligente' y así poder sacar mucho más beneficio de él.

En general los resultados obtenidos han sido buenos, y han ido mejorando notablemente a medida que se utilizaban nuevas herramientas. Todas ellas han supuesto un gran apoyo para realizar el proyecto. A continuación comentaré el grado de ayuda que ha aportado cada una de ellas.

El *HTMLParser* ha sido imprescindible, ya que extraer el texto de las páginas web para analizarlo era algo vital en el proceso, y sobretodo era importante obtenerlo nodo a nodo, olvidándonos de etiquetas que hubiesen entorpecido el proceso.

Los *Analyzers* de *Lucene* proporcionan un servicio de gran utilidad, y en nuestro caso ha sido especialmente útil el *StopAnalyzer*, ya que antes de utilizar esta herramienta se pasó por una fase en la que el proceso aceptaba las *stop words* y los resultados eran bastante malos. Por ejemplo, obteníamos infinidad de clases del tipo 'with cancer' o 'and biosensor'. Luego se optó por rechazar estas palabras pero sin ayuda del *StopAnalyzer* la única posibilidad era tener una lista de ellas e ir comparándolas una a una. Así que disponer de este 'filtro' para evitar dichas palabras ha sido una ventaja muy importante.

La herramienta *Snowball* realiza un trabajo muy interesante y complicado, pero en nuestro proceso no ha sido de tanta utilidad como otras herramientas, ya que no se daban muchos casos de palabras que pertenecieran a la misma familia y hubiera que eliminar. Pero aunque sean pocas, mejor evitarlas y perfeccionar, en la medida que sea, el proceso.

De las dos funciones para las que se ha utilizado *WordNet* una ha sido mucho más útil que la otra. Nos referimos a la categorización de palabras, ya que en este proceso era sumamente importante seleccionar los nombres y los adjetivos del resto de las palabras. En cambio con la otra función, el tratamiento de sinónimos, pasa lo mismo que con *Snowball*, han sido pocos los casos en que se han encontrado parejas de palabras sinónimas, pero como he dicho antes, cuanta más precisión se aporte al proceso, por poca que sea, mejor.

Finalmente, *Protégé* también ha sido de gran utilidad, ya que la interfaz gráfica del sistema no se ha implementado hasta el final, y hasta entonces era la única manera cómoda de visualizar los resultados que íbamos obteniendo.

Para terminar me gustaría añadir que un posible trabajo para el futuro sería ampliar el ámbito de este proyecto a otras lenguas diferentes de la inglesa. Existe una herramienta, *EuroWordNet* que extiende las utilidades que ofrece *WordNet* para el inglés, a varias lenguas europeas. El problema es que si se quisiera realizar un proceso similar para la lengua española, por ejemplo, nos encontraríamos seguramente con más dificultades que con la inglesa. Un ejemplo de este hecho es que si en inglés queremos encontrar una variante del concepto 'cancer', como hemos podido comprobar, basta con mirar las palabras anteriores para obtener 'skin cancer' o 'women cancer'. Aplicando este mismo

ejemplo para el español, las variantes serían del tipo ‘cáncer de piel’ o ‘cáncer en mujeres’, y aquí ya encontramos una dificultad añadida y es que entre el concepto y su ‘palabra relacionada’ encontramos otras palabras como ‘de’ o ‘en’. Pero bueno, todo es cuestión de buscar nuevas fórmulas que resuelvan los problemas que vayan surgiendo.

10- RECURSOS UTILIZADOS

10.1- Software

A continuación se enumeran las librerías y programas que hemos utilizado para la realización del proyecto, se explica brevemente para qué lo hemos utilizado y se proporciona una URL para descargarlo.

10.1.1- Librerías

- JADE: para poder utilizar todos los componentes de JADE (agente, ontologías, protocolos, etc.) → Jade.jar, iiop.jar y jadeTools.jar

http://jade.cselt.it/

- Google: para buscar en el google y obtener las URLs de las páginas web encontradas → googleapi.jar

http://www.google.com/apis/

- OWL: para crear la ontología en el lenguaje OWL → Api.jar , Impl.jar y Log4j.jar

http://sourceforge.net/project/showfiles.php?group_id=90989&package_id=95853

- RDF *parser*: para almacenar la ontología creada en un archivo '.owl', formato que puede leer Protégé. → rdfparser.jar

http://cvs.sourceforge.net/viewcvs.py/protege-owl/protege-owl/lib/

- HTML *parser*: para extraer el texto de las páginas web obtenidas. → htmlparser.jar

http://sourceforge.net/project/showfiles.php?group_id=24399&package_id=47712&release_id=256601

- Lucene: para analizar palabra a palabra el texto de las páginas web olvidándonos de los símbolos y de las *stop words*. → lucene-1.4-final.jar

http://apache.rediris.es/jakarta/lucene/binaries/lucene-1.4-final.zip

- Snowball: para comprobar si dos palabras son de la misma familia (tienen la misma raíz). → snowball-1.0.jar

http://jakarta.apache.org/lucene/docs/lucene-sandbox/snowball/

- WordNet: para saber la categoría gramatical de las palabras y así seleccionar los nombres y adjetivos, y para comprobar si dos palabras son sinónimas. → jwnl.jar y commons-logging.jar

http://sourceforge.net/projects/jwordnet

10.1.2- Programas

- WordNet: tiene que estar instalado para poder utilizar su librería.

http://www.cogsci.princeton.edu/~wn/wn2.0.shtml

- Protégé: para visualizar ontologías.

http://protege.stanford.edu/download.html

- OWL *plug-in:* para visualizar ontologías en OWL en Protégé.

http://protege.stanford.edu/plugins/owl/download.html

10.2- Páginas web

A continuación se enumeran las URLs de algunas de las páginas web que han servido de ayuda para realizar este proyecto.

Java

- **http://java.sun.com/j2se/1.4.2/docs/api/**
- **http://www.onjava.com/pub/a/onjava/2001/05/30/optimization.html**
- **http://www.javaworld.com/javaworld/javaqa/2001-06/03-qa-0622-vector.html**

Java Swing

- **http://java.sun.com/docs/books/tutorial/uiswing/components/**
- **http://www.rgagnon.com/bigindex.html**
- **http://forum.java.sun.com/thread.jsp?thread=531478&forum=31&message=2562219**
- **http://java.sun.com/j2se/1.4.2/docs/guide/swing/1.4/spinner.html**
- **http://www.tutorialized.com/tutorials/Java/Swing/1**
- **http://www.csie.nctu.edu.tw/document/java/tutorial/uiswing/components/spinner.html**
- **http://www-106.ibm.com/developerworks/java/library/j-mer0703/?open&l=766,t=grj,p=MwMjSpinner**
- **http://www.enode.com/x/markup/tutorial/spinner.html**
- **http://cblinux.fhs-hagenberg.ac.at/~aparamyt/ws_02/Readings/Books/Swing/2ndEdition/Chapter17.doc**
- **http://gate.ac.uk/gate/doc/java2html/com/ontotext/gate/vr/MappingTreeView.java.html**

JADE

- **http://sharon.cselt.it/projects/jade/community-faq.htm#q20**
- **http://mia.ece.uic.edu/~papers/MediaBot/jade/doc/examples/protocols.html**

OWL

- **http://owl.man.ac.uk/api.shtml**
- **http://www.mindswap.org/~aditkal/SEKE04.pdf.**
- **http://cvs.ecoinformatics.org/cvs/cvsweb.cgi/seek/projects/kr-sms/OwlIOTest/OwlIOTest.java?rev=1.2**
- **http://ontoweb.aifb.uni-karlsruhe.de/Members/hartmann/Bibliography_Folder.2004-01-22.5248/137**
- **http://www.w3.org/2004/OWL/**
- **http://www.cs.man.ac.uk/~horrocks/ISWC2003/Tutorial/**

Protégé

- **http://protege.stanford.edu/**

HtmlParser

- **http://htmlparser.sourceforge.net/**
- **http://javaboutique.internet.com/tutorials/HTMLParser/**
- **http://sourceforge.net/mailarchive/forum.php?thread_id=4059906&forum_id=2023**

Lucene

- **http://javaboutique.internet.com/tutorials/HTMLParser/**
- **http://jakarta.apache.org/lucene/docs/queryparsersyntax.html**
- **http://today.java.net/pub/a/today/2003/07/30/LuceneIntro.html**
- **http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words**
- **http://libpolaris.myclearwater.com/polaris/help/PWbasicsearch6.html**
- **http://europa.eu.int/eurodicautom/Controller?ACTION=stop_words**

Snowball

- **http://snowball.tartarus.org/**
- **http://www.tartarus.org/~martin/PorterStemmer/**
- **http://jakarta.apache.org/lucene/docs/lucene-sandbox/snowball/**
- **http://www.comp.lancs.ac.uk/computing/research/stemming/general/**
- **http://www.ils.unc.edu/keyes/java/porter/**
- **http://exist.sourceforge.net/api/org/exist/util/PorterStemmer.html**

WordNet

- **http://www.cogsci.princeton.edu/~wn/**

10.3- Bibliografía

A continuación se enumeran algunos de los libros, tutoriales y proyectos que han servido de ayuda para realizar este proyecto.

- *Ontological Engineering with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web*
Asunción Gómez-Pérez (PhD, MSc, MBA), Mariano Fernández-López (PhD, MSc), Óscar Corcho (MSc)
Facultad de Informática, Universidad Politécnica de Madrid

- *Tutorial: Creating Semantic Web (OWL) Ontologies with Protégé*
2nd International Semantic Web Conference (ISWC2003)
Holger Knublauch, Mark A. Musen, Natasha F. Noy
Sanibel Island, Florida, USA, October 20-23th, 2003

- *JADE Programmer's Guide*
Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB, formely CSELT), Giovanni Rimassa (University of Parma)
21st February 2003. JADE 3.0b1

- *Learning Java*
Pat Niemeyer, Jonathan Knudsen
Publisher: O'Reilly
First Edition May 2000
ISBN: 1-56592-718-4, 722 pages

- *Java Swing*
Robert Eckstein, Marc Loy & Dave Wood
Published by O'Reilly & Associates, Inc.

- *Basic Swing, GUI Controls in Java2*
Core Web Programming
http://www.corewebprogramming.com

- *Estudio de la implementación de agentes en dispositivos móviles*
Alexandre Viejo Galicia, Dr. Antonio Moreno Ribas, Dra. Aïda Valls Mateu
Proyecto Final de Carrera, Escola Técnica Superior d'Enginyeria (ETSE)
Universitat Rovira i Virgili (URV), 2003

- *Provisió de serveis sanitaris a través d'una plataforma d'agents*
David Sánchez Ruenes, Dr. Antonio Moreno Ribas, Dra. Aïda Valls Mateu
Proyecto Final de Carrera, Escola Técnica Superior d'Enginyeria (ETSE)
Universitat Rovira i Virgili (URV), 2001

11- MANUALES

11.1- Instalación de WordNet

Para poder ejecutar el sistema es necesario tener instalado el programa *WordNet*. Hay que descargar el archivo 'WordNet-2.0.exe' (ver URL en el apartado 10.1 → Programas), y el archivo 'jwnl13rc3.zip' (ver URL en el apartado 10.1 → Librerías). A continuación instalamos *WordNet*. Descomprimimos el archivo 'jwnl13rc3.zip' y modificamos, si es necesario, las siguientes líneas poniendo el *path* donde hemos instalado *WordNet*:

- En el archivo 'file_properties.xml' la línea:
<param name="dictionary_path" value="c:\program files\wordnet\2.0\dict"/>

- En el archivo 'map_properties.xml' la línea:
<param name="dictionary_path" value="c:\program files\wordnet\2.0\dict\serialized"/>

11.2- Instalación y uso de Protégé

Este programa lo podemos descargar de su página web (ver URL en el apartado 10.1 → Programas). Para ello es necesario registrarse. Para poder visualizar archivos en OWL es necesario tener el *plug-in* de OWL, que obtendremos si descargamos la versión completa (*full*) o, si no nos interesa tener el resto de *plug-ins*, podemos descargar la versión básica (*basic*) y descargar por separado el *plug-in* de OWL (ver URL en el apartado 10.1 → Programas). Instalamos Protégé y guardamos el *plug-in* en la carpeta de *plug-ins* del directorio donde hayamos instalado el programa.

Si queremos visualizar la ontología que hemos obtenido como resultado de la ejecución del sistema, simplemente tenemos que importar como un *OWL file* el archivo '.owl' que contiene la ontología.

En Protégé 2.0.1: Project → Import... → OWL Files → OWL file name
En Protégé 2.1.1: Project → Import to Standard Interface → OWL File → OWL file name

11.3- Introducción de datos en el sistema

A continuación se dan unos consejos mínimos para obtener un buen resultado del proceso. Recordemos los parámetros que debe introducir el usuario:

- El concepto básico a partir del cual desea construir la ontología.
- El máximo nivel de profundidad que podrá tener la ontología.
- El número máximo de páginas web a considerar (analizar) de cada clase.
- El número máximo de subclases que tendrá una clase.
- La forma de decidir cuando se genera una nueva clase. Las opciones son: el porcentaje de páginas web o el número de páginas web en que aparezca.
- El porcentaje de páginas web a partir del cual se genera una nueva clase.
- El número de páginas web a partir del cual se genera una nueva clase.

Como hemos comentado anteriormente, el único parámetro que debe introducir el usuario obligatoriamente es el concepto básico. Para este parámetro no se puede dar ningún consejo, como es lógico. Dependerá del dominio cuya información interesa estructurar al usuario.

El nivel de la ontología no debería ser muy elevado (hasta 4 ó 5 está bien), puesto que tardaría mucho tiempo en realizarse el proceso y con tanto nivel de descomposición del dominio quizá no se obtendrían muy buenos resultados.

Cuanto más elevado sea el número de páginas a analizar, mejor será el resultado, pero también aumentará bastante el tiempo de ejecución. Un valor recomendable estaría entre 1000 y 2000 páginas, aunque con menos (sin bajar de 100) también se obtienen buenos resultados.

El número de subclases no afecta tanto al tiempo de ejecución, ya que una de las ventajas que nos proporciona trabajar con un SMA es que varios agentes pueden trabajar concurrentemente, y en nuestro caso cada agente se encargará de una subclase. Por lo tanto, se puede decidir en función de las necesidades del usuario (obviamente, cuantas más subclases se creen, más posibilidades existen de encontrar el subdominio que se busca, si se busca uno en concreto).

En cuanto al porcentaje y número mínimo de páginas para generar una ontología, un número elevado filtrará mucho la búsqueda, aceptando sólo subclases muy repetidas en las páginas.

Y la decisión entre porcentaje o número mínimo de páginas no es relevante, ya que si analizamos 1000 páginas, por ejemplo, nos da igual elegir un porcentaje del 30% que un número mínimo de 300 páginas.

Por último, destacar que todo depende, sobretodo, de las necesidades del usuario, ya que no es lo mismo que esté buscando información general sobre un tema (en cuyo caso no le interesará hacer una descomposición muy grande) o que busque información sobre una parte concreta de un tema (en este caso sí le interesará descomponer mucho, para tener más posibilidades de encontrar lo que busca).

A.- CÓDIGO FUENTE

A.1- Paquete Agents

A.1.1- BuilderAgent.java

```
package Agents;

import Ontology.*;

import java.io.*;
import java.util.*;
import java.net.*;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import jade.proto.*;
import jade.content.*;
import jade.content.onto.basic.*;
import jade.content.onto.*;
import jade.content.lang.*;
import jade.content.lang.sl.*;
import jade.domain.JADEAgentManagement.*;

import org.semanticweb.owl.util.*;
import org.semanticweb.owl.model.*;
import org.semanticweb.owl.model.change.*;
import org.semanticweb.owl.io.owl_rdf.Renderer;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.snowball.SnowballAnalyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.Token;

import net.didion.jwnl.JWNLException;
import net.didion.jwnl.data.*;
import net.didion.jwnl.dictionary.Dictionary;

public class BuilderAgent extends Agent {

   BuilderAgent myBuilder = this;

   Request r = new Request();
   int maxLevel;
   String mainSearch;
   ArrayList level = new ArrayList();
   ArrayList searches = new ArrayList();
   ArrayList classes = new ArrayList();
   SearchAndLevel search = new SearchAndLevel();
   String agentName;
   ArrayList agentNames = new ArrayList();
   int numAgents;
   int agents;
   String ontology_name;
   OWLConnection connection_ant;
   OWLDataFactory factory_ant;
   ChangeVisitor visitor_ant;
   String word;
   OWLOntology ontology_ant;
   ArrayList URIs = new ArrayList();
   OWLClass onto_keyword_ant;
   OWLClass superClass;
   int currentLevel = 1;
```

```
   OntologyBuilderGUI gui;

   Behaviour b;

   private ContentManager manager = (ContentManager) getContentManager();
   private Codec codec = new SLCodec();
   private Ontology ontology = WebOntology.getInstance();

   private static BuilderAgent theInstance = null;
   public static BuilderAgent getInstance() {
      return theInstance;
   }

   public void setup() {

      manager.registerLanguage(codec, codec.getName());
      manager.registerOntology(ontology, ontology.getName());
      manager.registerOntology(JADEManagementOntology.getInstance(),
JADEManagementOntology.NAME);

      theInstance = this;

      gui = new OntologyBuilderGUI();
   }

   public void startBuilding(){

      r.setSearch(gui.searchField.getText());
      mainSearch = r.getSearch();
      r.setLevel(((Integer)gui.option1F.getValue()).intValue());
      maxLevel = r.getLevel();
      r.setNumberMaxWebs(((Integer)gui.option2F.getValue()).intValue());
      r.setPercentageOfWebs(((Integer)gui.option3F.getValue()).intValue());
      r.setNumberOfSubclasses(((Integer)gui.option4F.getValue()).intValue());
      if (gui.option5F.getSelectedItem().equals("        P     ")){
         r.setPercentageOrNumber(true);
      }
      else{
         r.setPercentageOrNumber(false);
      }
      r.setNumberOfWebs(((Integer)gui.option6F.getValue()).intValue());

      b = new BuilderFirstBehaviour(this);
      addBehaviour(b);
   }

   public void takeDown() {
   }

   class BuilderFirstBehaviour extends SimpleBehaviour {

      public BuilderFirstBehaviour(Agent a) {
         super(a);
      }

      public void action() {

         ///////// OWL ///////////
         try{
            word = mainSearch.toLowerCase() + "Ontology";
            ontology_name = "http://grusma.etse.urv.es/ontologies/" + word +
"/#";
            Map parameters_ant = new HashMap();

parameters_ant.put(OWLManager.OWL_CONNECTION,"org.semanticweb.owl.impl.model.O
WLConnectionImpl");
            connection_ant = OWLManager.getOWLConnection(parameters_ant);
```

```
            factory_ant = connection_ant.getDataFactory();
            ontology_ant = connection_ant.createOntology(new
URI("file:./Resultats/" + word + ".owl"), new
URI("http://grusma.etse.urv.es/ontologies/" + word + "/#"));
            visitor_ant = connection_ant.getChangeVisitor(ontology_ant);
         }
         catch (Exception e) {
            System.err.println("ERROR creando la ontologia: " + e);
            myBuilder.takeDown();
         }
         ////////////////////////

         level.add(new Integer(1));
         search.setLevel(1);
         search.setSearch(mainSearch.toLowerCase());
         word = mainSearch.toLowerCase();

         ///////// OWL ///////////
         try{
            onto_keyword_ant =  factory_ant.getOWLClass( new
URI(ontology_name+word));
            URIs.add(onto_keyword_ant);
            AddEntity ae1 = new AddEntity( ontology_ant, onto_keyword_ant,
null );
            ae1.accept( visitor_ant );
         }
         catch (Exception e) {
            System.err.println("ERROR creando la clase " + word + ": " + e);
            myBuilder.takeDown();
         }
         ////////////////////////

         searches.add(search);
         numAgents = 1;
         agents = 1;
         agentName = "Searcher" + numAgents;
         agentNames.add(agentName);
         newAgent(agentName , "Agents.SearcherAgent");
      }

      public boolean done() {
         return true;
      }
   }

   public void newAgent(String agentName, String className){

      CreateAgent ca = new CreateAgent();
      String containerName = AgentManager.MAIN_CONTAINER_NAME;
      ca.setAgentName(agentName);
      ca.setClassName(className);
      ca.setContainer(new ContainerID(containerName, null));
      try {
         jade.content.onto.basic.Action a = new
jade.content.onto.basic.Action();
         a.setAction(ca);
         a.setActor(getAMS());
         ACLMessage requestMsg = new ACLMessage(ACLMessage.REQUEST);
         requestMsg.setOntology(JADEManagementOntology.NAME);
         requestMsg.setProtocol("FIPA-Request");
         requestMsg.setLanguage(codec.getName());
         requestMsg.setSender((AID)getAID());
         requestMsg.clearAllReceiver();
         requestMsg.addReceiver((AID)getAMS());
         getContentManager().fillContent(requestMsg, (ContentElement)a);
         addBehaviour(new AMSClientBehaviour(this, "CreateAgent",
requestMsg));
      }
```

```
      catch (Exception fe) {
         fe.printStackTrace();
      }
   }

   // Sends requests to the AMS
   private class AMSClientBehaviour extends AchieveREInitiator {

      private String actionName;

      public AMSClientBehaviour(Agent parent,   String an, ACLMessage request)
{
         super(parent, request);
         actionName = an;
      }

      protected void handleNotUnderstood(ACLMessage reply) {
         System.err.println("NOT-UNDERSTOOD received by RMA during " +
actionName);
      }

      protected void handleRefuse(ACLMessage reply) {
         System.err.println("REFUSE received during " + actionName);
      }

      protected void handleAgree(ACLMessage reply) {
         System.err.println("AGREE received during " + actionName);
      }

      protected void handleFailure(ACLMessage reply) {
         System.err.println("FAILURE received during " + actionName);
      }

      protected void handleInform(ACLMessage reply) {
         System.err.println("INFORM received during " + actionName);
         if (actionName == "CreateAgent"){
            createRequest();
         }
      }
   }

   public void createRequest() {

      ACLMessage request = createMessage();
      b = new RequestInitiator(this, request);
      addBehaviour(b);
   }

   //   Método que crea un nuevo mensaje (un request)
   public ACLMessage createMessage(){
      //   Creamos el mensaje y rellenamos los campos claves
      ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
      AID receiver = new AID(agentNames.get(0) + "@" + getHap(), false);
      agentNames.remove(0);
      msg.setSender(getAID());
      msg.addReceiver(receiver);
      msg.setLanguage(codec.getName());
      msg.setOntology(ontology.getName());
      msg.setProtocol(jade.domain.FIPANames.InteractionProtocol.FIPA_REQUEST);
      // Rellenamos la acción SearchWord
      search = ((SearchAndLevel)searches.get(0));
      searches.remove(0);
      r.setSearch(search.getSearch());
      r.setLevel(search.getLevel());
      SearchWord sw = new SearchWord();
      sw.setRequest(r);
      try{
```

```
         // Rellenamos el contenido (SL requires actions to be included into
the action construct)
         Action a = new Action(getAID(), sw);
         manager.fillContent(msg, a);
      }
      catch(Exception e){
         e.printStackTrace();
      }
      // Devolvemos el ACLMessage
      return msg;
   }

   //   Clase que inicia el Request
   class RequestInitiator extends AchieveREInitiator {

      public RequestInitiator(Agent a, ACLMessage req) {
         super(a, req);
      }

      protected void handleAgree(ACLMessage agree){
         System.out.println("["+getLocalName()+"]:"+" Ha recibido un
'agree'");
      }

      protected void handleFailure(ACLMessage failure) {
         System.out.println("["+getLocalName()+"]:"+" Ha recibido un
'failure'");
         createRequest();
      }

      protected void handleInform(ACLMessage inform) {
         System.out.println("["+getLocalName()+"]:"+" Ha recibido un
'inform'");
         try{
            Equals e = (Equals)manager.extractContent(inform);
            if (e.getRight() instanceof Inform){
               destroyAgent(inform.getSender());
               agents --;
               Inform i = (Inform)e.getRight();
               jade.util.leap.ArrayList result = new
jade.util.leap.ArrayList();
               result = i.getSubClasses();
               ClassAndWebs caw;
               caw = (ClassAndWebs)result.get(0);
               result.remove(0);
               // Añadir nodo al árbol de la interfaz gráfica
               if (caw.getWord().equalsIgnoreCase(mainSearch)){
                  gui.addNewNode(caw, null);
               }
               else{
                  String s;
                  StringTokenizer st = new StringTokenizer(caw.getWord());
                  st.nextToken();
                  s = st.nextToken();
                  while (st.hasMoreTokens()){
                     s = s.concat(" ".concat(st.nextToken()));
                  }
                  gui.addNewNode(caw, s);
               }

               if (result.isEmpty()){
                  b = new BuilderThirdBehaviour(myBuilder);
                  addBehaviour(b);
               }
               else{

               Iterator it = result.iterator();
               while (it.hasNext()){
```

```
                    boolean repeatedClass = false;
                    caw = (ClassAndWebs)it.next();
                    currentLevel = ((Integer)level.get(0)).intValue() + 1;
                    Iterator itt = classes.iterator();
                    SearchAndLevel sal;
                    while (itt.hasNext()) {
                       sal = (SearchAndLevel)itt.next();
                       if (sal.getLevel() == currentLevel){
                          if (compareClasses(caw.getWord(), sal.getSearch(),
currentLevel)){
                             repeatedClass = true;
                             break;
                          }
                          else{
                          }
                       }
                    }
                    if (!repeatedClass){
                    classes.add(new SearchAndLevel(caw.getWord(),
currentLevel));
                    searches.add(new SearchAndLevel(caw.getWord(),
currentLevel));
                    level.add(new Integer(currentLevel));

                    ///////// OWL ///////////
                    try{
                       word = caw.getWord().replace(' ', '_');
                       onto_keyword_ant =  factory_ant.getOWLClass( new
URI(ontology_name + word));
                       URIs.add(onto_keyword_ant);
                       AddEntity ae1 = new AddEntity( ontology_ant,
onto_keyword_ant, null );
                       ae1.accept( visitor_ant );
                       boolean found = false;
                       Iterator ittt = URIs.iterator();
                       while ((!found) && (ittt.hasNext())){
                          superClass = (OWLClass) ittt.next();
                          if (word.endsWith(superClass.getURI().getFragment())){
                             found = true;
                          }
                       }
                       if (found){
                          OntologyChange oc1 = new AddSuperClass(ontology_ant,
onto_keyword_ant, superClass, null);
                          oc1.accept(visitor_ant);
                       }
                       else{
                          System.out.println("ERROR: Esta clase no tiene super
clase");
                       }
                       OWLAnnotationProperty oap =
factory_ant.getOWLAnnotationProperty(new URI(ontology_name + "Urls"));
                       OntologyChange  oc = new
AddAnnotationInstance(ontology_ant, onto_keyword_ant, oap, caw.getUrl(),
null);
                       oc.accept(visitor_ant);
                    }
                    catch (Exception ex) {
                       System.err.println("ERROR creando la clase " + word + ":
" + ex);
                       myBuilder.takeDown();
                    }
                    ////////////////////////

                    if (currentLevel == maxLevel){
                       // Si es una hoja, lo añado al árbol
                       String s;
                       StringTokenizer st = new StringTokenizer(caw.getWord());
```

```
                    st.nextToken();
                    s = st.nextToken();
                    while (st.hasMoreTokens()){
                       s = s.concat(" ".concat(st.nextToken()));
                    }
                    gui.addNewNode(caw, s);
                 }

                 Behaviour b = new BuilderSecondBehaviour(myBuilder);
                 addBehaviour(b);
                 }
              }
              }
              URIs.remove(superClass);
              level.remove(0);
           }
        }
        catch(Exception e){
           System.out.println("ERROR en la extracción del inform");
        }
     }
  }

   public boolean compareClasses(String class1, String class2, int level)
throws IOException{

      // Comprobar si hay dos palabras de la misma familia
      Analyzer sa = new SnowballAnalyzer("English",
StopAnalyzer.ENGLISH_STOP_WORDS);
      TokenStream stream1 = sa.tokenStream("contents", new
StringReader(class1));
      Token token1 = stream1.next();
      TokenStream stream2;
      Token token2;
      int equals = 0;
      while (token1 != null){
         stream2 = sa.tokenStream("contents", new StringReader(class2));
         token2 = stream2.next();
         while (token2 != null){
            if (token1.termText().equalsIgnoreCase(token2.termText())){
               equals ++;
               break;
            }
            token2 = stream2.next();
         }
         token1 = stream1.next();
      }

      // Comprobar si hay dos palabras sinonimas
      if (equals != level){
         equals = 0;

         Dictionary dictionary = Dictionary.getInstance();
         IndexWord w1 = null;
         IndexWord w2 = null;
         boolean found;

         try {
            StringTokenizer st1 = new StringTokenizer(class1);
            StringTokenizer st2;
            String s1;
            String s2;
            Synset[] senses;
            Word[] words;
            while (st1.hasMoreTokens()){
               found = false;
               s1 = st1.nextToken();
               w1 = (dictionary.lookupIndexWord(POS.ADJECTIVE, s1));
```

```
                if (w1 != null){
                   senses = w1.getSenses();
                   if (senses != null){
                      for (int i = 0; i < senses.length; i++){
                         words = senses[i].getWords();
                         for (int j = 0; j < words.length; j++){
                            st2 = new StringTokenizer(class2);
                            while (st2.hasMoreTokens()){
                               s2 = st2.nextToken();
                               if (words[j].getLemma().equalsIgnoreCase(s2)){
                                  equals++;
                                  found = true;
                                  break;
                               }
                            }
                            if (found){
                               break;
                            }
                         }
                         if (found){
                            break;
                         }
                      }
                   }
                }
                if (!found){
                   w2 = (dictionary.lookupIndexWord(POS.NOUN, s1));
                   if (w2 != null){
                      senses = w2.getSenses();
                      if (senses != null){
                         for (int i = 0; i < senses.length; i++){
                            words = senses[i].getWords();
                            for (int j = 0; j < words.length; j++){
                               st2 = new StringTokenizer(class2);
                               while (st2.hasMoreTokens()){
                                  s2 = st2.nextToken();
                                  if
(words[j].getLemma().equalsIgnoreCase(s2)){
                                     equals++;
                                     found = true;
                                     break;
                                  }
                               }
                               if (found){
                                  break;
                               }
                            }
                            if (found){
                               break;
                            }
                         }
                      }
                   }
                }
             }
          } catch (JWNLException e) {
             e.printStackTrace();
          }
       }

       return (equals == level);
    }

    public void destroyAgent(AID agentAID) {

       KillAgent ka = new KillAgent();
       ka.setAgent(agentAID);
       try {
```

```
         jade.content.onto.basic.Action a = new
jade.content.onto.basic.Action();
         a.setAction(ka);
         a.setActor(getAMS());
         ACLMessage requestMsg = new ACLMessage(ACLMessage.REQUEST);
         requestMsg.setOntology(JADEManagementOntology.NAME);
         requestMsg.setProtocol("FIPA-Request");
         requestMsg.setLanguage(codec.getName());
         requestMsg.setSender((AID)getAID());
         requestMsg.clearAllReceiver();
         requestMsg.addReceiver((AID)getAMS());
         getContentManager().fillContent(requestMsg, (ContentElement)a);
         addBehaviour(new AMSClientBehaviour(myBuilder, "KillAgent",
requestMsg));
      }
      catch (Exception fe) {
         fe.printStackTrace();
      }
   }

   class BuilderSecondBehaviour extends SimpleBehaviour {

      public BuilderSecondBehaviour(Agent a) {
         super(a);
      }

      public void action() {

            search = ((SearchAndLevel)searches.get(0));
            if (search.getLevel() < maxLevel){
               numAgents ++;
               agents ++;
               agentName = "Searcher" + numAgents;
               agentNames.add(agentName);
               newAgent(agentName , "Agents.SearcherAgent");
            }
            else{
               searches.remove(0);
               if ((agents == 0) && (searches.isEmpty())){
                  try {
                     File results1 = new File(mainSearch + "Ontology.owl");
                     FileWriter w1 = new FileWriter(results1);
                     Renderer r1 = new Renderer();
                     r1.renderOntology(ontology_ant, w1);
                     w1.flush();
                     System.out.println ("**** FIN ****");
                     gui.progressLabel.setText("      Done");
                  }
                  catch (Exception e) {
                     System.err.println("ERROR en la creación del archivo OWL"
+ e);
                     myBuilder.takeDown();
                  }
               }
            }
      }

      public boolean done() {
         return true;
      }
   }

   class BuilderThirdBehaviour extends SimpleBehaviour {

      public BuilderThirdBehaviour(Agent a) {
         super(a);
      }
```

```
      public void action() {

         if ((agents == 0) && (searches.isEmpty())){
            try {
               File results1 = new File(mainSearch + "Ontology.owl");
               FileWriter w1 = new FileWriter(results1);
               Renderer r1 = new Renderer();
               r1.renderOntology(ontology_ant, w1);
               w1.flush();
               System.out.println ("**** FIN ****");
               gui.progressLabel.setText("      Done");
            }
            catch (Exception e) {
               System.err.println("ERROR en la creación del archivo OWL" + e);
               myBuilder.takeDown();
            }
         }
      }

      public boolean done() {
         return true;
      }
   }
}
```

A.1.2- SearcherAgent.java

```
package Agents;

import Ontology.*;

import java.io.*;
import java.util.*;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import jade.proto.*;
import jade.content.*;
import jade.content.onto.basic.*;
import jade.content.onto.*;
import jade.content.lang.*;
import jade.content.lang.sl.*;

// API to exchange information with Google (www.google.com)
import com.google.soap.search.*;

import org.htmlparser.Parser;
import org.htmlparser.visitors.TextExtractingVisitor;
//import org.htmlparser.util.ParserException;

import net.didion.jwnl.JWNL;
import net.didion.jwnl.JWNLException;
//import net.didion.jwnl.JWNLException;
import net.didion.jwnl.data.*;
import net.didion.jwnl.dictionary.Dictionary;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.snowball.SnowballAnalyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.Token;

public class SearcherAgent extends Agent {

   SearcherAgent mySearcher = this;

   Request r = new Request();
   Inform i = new Inform();
   String search; // Nombre de la clase (palabra que el agente buscador tendrá
que buscar en el google)
   int level; // Nivel del árbol y número de palabras a buscar
   int numberMaxWebs; // Número máximo de páginas a considerar de cada clase
   int numberOfSubclasses; // Número máximo de subclases de una clase
   int percentageOfWebs; // Porcentaje de páginas a partir del cual se genera
una nueva clase
   int numberOfWebs; // Número de páginas a partir del cual se genera una
nueva clase
   boolean percentageOrNumber;

   private ContentManager manager = (ContentManager) getContentManager();
   private Codec codec = new SLCodec();
   private Ontology ontology = WebOntology.getInstance();

   public void setup() {

      manager.registerLanguage(codec, codec.getName());
      manager.registerOntology(ontology, ontology.getName());

      Behaviour rq = new RequestResponder(this,
AchieveREResponder.createMessageTemplate
```

```
(jade.domain.FIPANames.InteractionProtocol.FIPA_REQUEST));
      addBehaviour(rq);
   }

   public void takeDown(){
   }

   class RequestResponder extends AchieveREResponder {

      public RequestResponder(Agent a, MessageTemplate mt) {
         super(a, mt);
      }

      // Método que se ejecuta automáticamente cuando recibe un REQUEST
      protected ACLMessage prepareResponse(ACLMessage request) {
         ACLMessage response = request.createReply();
         try{
            Action a = (Action)manager.extractContent(request);
            if (a.getAction() instanceof SearchWord){
               System.out.println("["+getLocalName()+"]:"+" Ha recibido un
request de SearchWord");
               response.setPerformative(ACLMessage.AGREE);
            }
         }
         catch(Exception e) {
            System.out.println("ERROR en la introducción de información");
            e.printStackTrace();
            mySearcher.takeDown();
         }
         return response;
      }

      // Método que se ejecuta automáticamente cuando recibe un REQUEST
      protected ACLMessage prepareResultNotification(ACLMessage
request,ACLMessage response) {
         response.setPerformative(ACLMessage.INFORM);
         try{
            Action a = (Action)manager.extractContent(request);
            if (a.getAction() instanceof SearchWord){
               SearchWord sw = (SearchWord)a.getAction();
               r = sw.getRequest();
               search = r.getSearch();
               level = r.getLevel();
               numberMaxWebs = r.getNumberMaxWebs();
               numberOfSubclasses = r.getNumberOfSubclasses();
               percentageOrNumber = r.getPercentageOrNumber();
               if (percentageOrNumber) {
                  percentageOfWebs = r.getPercentageOfWebs();
               }
               else {
                  numberOfWebs = r.getNumberOfWebs();
               }
               Vector v = searchGoogle("\"".concat(search.concat("\"")),
numberMaxWebs);
               System.out.println("RESULTS: " + v.size());
               ArrayList result = analysisWebs(v);
               i = new Inform();
               Iterator itt = result.iterator();
               while (itt.hasNext()) {
                  i.addSubClass((ClassAndWebs) itt.next());
               }
            }
            // Rellenamos el contenido (SL requires actions to be included
into the action construct)
            Equals e = new Equals(a, i);
            manager.fillContent(response, e);
         }
```

```
         catch(Exception e){
            e.printStackTrace();
         }
         // Devolvemos el ACLMessage
         return response;
      }
   }

   static public Vector searchGoogle(String name, int maxPages) throws
GoogleSearchFault {

      final String[] keys = {
         "gxCKbLH6QBjPSLK0FdKa8UN8SKyliVJK",
           "AARe7PZQFHI3V5k+mN3hXftuZLm7PTNq",
           "2oOyrvFQFHLapwVtBEBhzVvRdnNTJE1J",
           "Bboid2xQFHLQWGDqIevWdknQlDGO/LQM",
           "RNDNoWpQFHKEBGTHSuK2qec2pAA3uqUx",
           "pKqLM4JQFHKSFngrpaUP253MjVT58lat",
      };
      int current_key = 0;
      Vector pagesRetrieved = new Vector();
      int i = 0;

        while (pagesRetrieved.size()<maxPages) {
         try {
            GoogleSearch s = new GoogleSearch();  //instance of Google class
              s.setKey((String)keys[current_key]);
              s.setQueryString(name);
              s.setStartResult(i);
              s.setMaxResults(Math.min(10,maxPages - pagesRetrieved.size()));
            GoogleSearchResult r = s.doSearch(); //first search
              // Si no hay resultados, retornamos
              if (r.getEstimatedTotalResultsCount()<1) {
               return pagesRetrieved;
              }
            // Vector con los nuevos links pasados a InformationURL
            Vector new_links = loadPagesRetrieved(r);
              for (int j=0; j<new_links.size(); j++) {
               if (pagesRetrieved.size() < r.getEstimatedTotalResultsCount())
{
               // Añadimos un links si necesitamos más, si aún había
resultados
                 // y si no se trata de un pdf o doc
                 String str = (String) new_links.get(j);
                 if (str.length() != 0)
                  if ((!str.endsWith(".pdf")) && (!str.endsWith(".doc")))
                       pagesRetrieved.add(str);
               }
              }
         }
         catch (com.google.soap.search.GoogleSearchFault e) {
            System.err.println(e.toString());
            if (e.toString().indexOf("Unsupported response")!=-1) {
               System.err.println("["+name+"] Google exception -> CONNECTION
ERROR");
               i -= 10;
              }
              if (e.toString().indexOf("Error opening socket")!=-1) {
                 System.err.println("["+name+"] Google exception -> SOCKET
ERROR");
                 i -= 10;
              }
              if (e.toString().indexOf("1000") != -1) {
               System.err.println("["+name+"] Google exception -> KEY ERROR");
               // Hay que cambiar de clave
               if (current_key < keys.length - 1) { // Si aún quedan claves
                    current_key++; // Se cambia
```

```
                    System.err.println("[" + name + "] Google exception -> New
KEY: " + (String) keys[current_key]);
                    i -= 10;
                }
                else { // Si no, se acaba la búsqueda y se informa
                    System.err.println("[" + name + "] Google exception -> The
are no more KEYS!!!");
                    return pagesRetrieved;
                  }
             }
         }
         i+=10;
        }
        return pagesRetrieved;
   }

   private static Vector loadPagesRetrieved(GoogleSearchResult r) {
        Vector newPages = new Vector();
        GoogleSearchResultElement[] re = r.getResultElements();
        for(int i=0; i<re.length; i++) {
         String webaddress = re[i].getURL();
         newPages.add( webaddress );
        }
        return newPages;
   }

   public ArrayList analysisWebs (Vector webs) {

      int numWords = r.getLevel();
      String word = "";
      String prevWord;
      String listedWord;
      String contingut1 = new String();
      String contingut2 = new String();
      boolean badWord = false;
      Hashtable allWords = new Hashtable();
      String currentWeb;
      String tokens [] = new String [level];

      for (int i=0; i<tokens.length; i++) {
         tokens[i] = "";
      }

      try {
         JWNL.initialize(new
FileInputStream("./lib/jwnl13rc3/file_properties.xml"));
      } catch (FileNotFoundException e2) {
         e2.printStackTrace();
      } catch (JWNLException e2) {
         e2.printStackTrace();
      }

      Dictionary dictionary = Dictionary.getInstance();
      IndexWord w1 = null;
      IndexWord w2 = null;

      ArrayList stopWords = new ArrayList();
      try {
         BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream("PFC/stopWords.txt")));
         String line = "";
         while((line = br.readLine()) != null) {
            stopWords.add(line);
         }
      } catch (FileNotFoundException e1) {
         e1.printStackTrace();
      } catch (IOException e1) {
         e1.printStackTrace();
```

```
      }
      //  converting an ArrayList to an array
      String[] sw = (String[])stopWords.toArray(new String[stopWords.size()]);
      Analyzer analyzer = new StopAnalyzer(sw);

      int numWeb = 0;
      Iterator ite = webs.iterator();

      while (ite.hasNext()) {
         currentWeb = (String) ite.next();
      numWeb ++;
      System.out.print(numWeb + " ");
         try {
            Parser parser1 = new Parser(currentWeb);
            TextExtractingVisitor visitor1 = new TextExtractingVisitor();
            parser1.visitAllNodesWith(visitor1);
            contingut1 = visitor1.getExtractedText();
            TokenStream stream = analyzer.tokenStream("contents", new
StringReader(contingut1));
            Token token = stream.next();
            while (token != null){
               prevWord = tokens[0];
               for (int i=0; i<(tokens.length - 1); i++) {
                  tokens[i] = tokens[i + 1];
               }
               tokens[(tokens.length) - 1] = token.termText();
               token = stream.next();
               word = tokens[0];
               for (int i=1; i<tokens.length; i++) {
                  word = word.concat(" ".concat(tokens[i]));
               }
               if (word.equalsIgnoreCase(search)) {
                  // Compruebo que la palabra no sea demasiado corta
                  if (prevWord.length() < 3){
                     badWord = true;
                  }
                  // Compruebo si la palabra está 'parecida' en la clase
                  if (!badWord){
                     SnowballAnalyzer sa = new SnowballAnalyzer("English",
StopAnalyzer.ENGLISH_STOP_WORDS);
                     TokenStream stream1 = sa.tokenStream("contents", new
StringReader(prevWord));
                     Token token1 = stream1.next();
                     if (token1 != null){
                        for (int i = 0; i < (tokens.length); i++){
                           TokenStream stream2 = sa.tokenStream("contents",
new StringReader(tokens[i]));
                           Token token2 = stream2.next();
                           if (token2 != null){
                              if
(token1.termText().equalsIgnoreCase(token2.termText())){
                                 badWord = true;
                                 break;
                              }
                           }
                        }
                     }
                  }
                  // Compruebo si la palabra es nombre o adjetivo
                  if (!badWord){
                     w1 = (dictionary.lookupIndexWord(POS.ADJECTIVE,
prevWord));
                     w2 = (dictionary.lookupIndexWord(POS.NOUN, prevWord));
                     if ((w1 == null) && (w2 == null)){
                        badWord = true;
                     }
                  }
```

```
                  // Compruebo si la palabra tiene un sinónimo en la clase
entre sus significados como adjetivo
                  if ((!badWord) && (w1 != null)){
                     Synset[] s1 = w1.getSenses();
                     if (s1 != null){
                        for (int i = 0; i < s1.length; i++){
                           Word[] ww1 = s1[i].getWords();
                           for (int j = 0; j < ww1.length; j++){
                              for (int k = 0; k < (tokens.length); k++){
                                 if
(ww1[j].getLemma().equalsIgnoreCase(tokens[k])){
                                    badWord = true;
                                 }
                              }
                           }
                        }
                     }
                  }
                  // Compruebo si la palabra tiene un sinónimo en la clase
entre sus significados como nombre
                  if ((!badWord) && (w2 != null)){
                     Synset[] s2 = w2.getSenses();
                     if (s2 != null){
                        for (int i = 0; i < s2.length; i++){
                           Word[] ww2 = s2[i].getWords();
                           for (int j = 0; j < ww2.length; j++){
                              for (int k = 0; k < (tokens.length); k++){
                                 if
(ww2[j].getLemma().equalsIgnoreCase(tokens[k])){
                                    badWord = true;
                                 }
                              }
                           }
                        }
                     }
                  }
                  // Si finalmente la palabra es 'buena', la guardo
                  if (!badWord) {
                     if (allWords.get(prevWord) == null) {
                        ClassAndWebs s = new ClassAndWebs();
                        s.setWord(prevWord);
                        s.setCount(1);
                        jade.util.leap.ArrayList se = s.getUrl();
                        se.add(currentWeb);
                        s.setUrl(se);
                        allWords.put(prevWord, s);
                     }
                     else {
                        ClassAndWebs s = (ClassAndWebs)
allWords.get(prevWord);
                        s.setCount(s.getCount() + 1);
                        jade.util.leap.ArrayList se = s.getUrl();
                        if (!se.contains(currentWeb)) {
                           se.add(currentWeb);
                        }
                        s.setUrl(se);
                        allWords.put(prevWord, s);
                     }
                  }
                  else {
                     badWord = false;
                  }
               }
            }
         }
         catch (Exception e) {
            //System.err.println("ERROR en el parsing de una web o tratamiento
de palabra: " + e.toString());
```

```
         }
      }

      Collection col = allWords.values();
      ArrayList al = new ArrayList();
      al.addAll(col);
      IntegerComparator ic = new IntegerComparator();
      Collections.sort (al, ic);

      ArrayList subClasses = new ArrayList(numberOfSubclasses);
      Iterator it = al.iterator();
      boolean stop = false;
      ClassAndWebs newClass;

      while ((it.hasNext()) && (! stop) && (subClasses.size() <
numberOfSubclasses)) {
         newClass = (ClassAndWebs) it.next();
         if (percentageOrNumber) {
            if (((newClass.getUrl().size()*100)/webs.size()) <
percentageOfWebs) {
               stop = true;
            }
            else {
               newClass.setWord(newClass.getWord().concat("
".concat(search)));
               subClasses.add(newClass);
            }
         }
         else {
            if ((newClass.getUrl().size()) < numberOfWebs) {
               stop = true;
            }
            else {
               newClass.setWord(newClass.getWord().concat("
".concat(search)));
               subClasses.add(newClass);
            }
         }
      }

      Iterator itSubClasses = subClasses.iterator();
      while (itSubClasses.hasNext()){
         Iterator itUrls =
((ClassAndWebs)itSubClasses.next()).getUrl().iterator();
         while (itUrls.hasNext()){
            String s = (String)itUrls.next();
            if (webs.contains(s)){
               webs.remove(s);
            }
         }
      }

      ClassAndWebs caw = new ClassAndWebs();
      caw.setWord(search);
      jade.util.leap.ArrayList w = new jade.util.leap.ArrayList(new
ArrayList(webs));
      caw.setUrl(w);
      subClasses.add(0, caw);

      return subClasses;
   }
}
```

A.1.3- SearchAndLevel.java

```
package Agents;

public class SearchAndLevel {

	private String search;
	private int level;

	public SearchAndLevel() {
	}

	public SearchAndLevel(String s, int l) {
	   search = s;
	   level = l;
	}

	public String getSearch(){
	   return search;
	}

	public void setSearch(String s){
	   search = s;
	}

	public int getLevel() {
	   return level;
	}

	public void setLevel(int l){
	   level = l;
	}
}
```

A.1.4- OntologyBuilderGUI.java

```
package Agents;

import Ontology.*;

import java.util.*;
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import javax.swing.text.*;

public class OntologyBuilderGUI extends JFrame implements ActionListener,
DocumentListener, MouseListener {

   private BuilderAgent agent = BuilderAgent.getInstance();

    // JFrame >
    JPanel formPanel = new JPanel();
    JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
    JPanel statusBar = new JPanel();

    // JFrame > formPanel >
    JPanel searchPanel = new JPanel();
    JPanel searchOptionsPanel = new JPanel();

    // JFrame > splitPane >
   JScrollPane treePanel = new JScrollPane();
   JScrollPane webPanel = new JScrollPane();

    // JFrame > statusbar >
   JLabel progressLabel = new JLabel(" ", JLabel.RIGHT);

    // JFrame > formPanel > searchPanel >
   JPanel northSearchPanel = new JPanel();
    JButton ok = new JButton("Build ontology");

   // JFrame > formPanel > searchPanel > northSearchPanel >
   JLabel searchLabel = new JLabel("Search:", JLabel.RIGHT);
   JTextField searchField = new JTextField(10);
   Document document;

    // JFrame > formPanel > searchOptionsPanel >
   JPanel searchOptionsPanel1 = new JPanel();
   JPanel searchOptionsPanel11 = new JPanel();
   JPanel searchOptionsPanel2 = new JPanel();
   JPanel searchOptionsPanel22 = new JPanel();
   JPanel searchOptionsPanel3 = new JPanel();
   JPanel searchOptionsPanel33 = new JPanel();
   JPanel searchOptionsPanel4 = new JPanel();
   JPanel searchOptionsPanel44 = new JPanel();
   JPanel searchOptionsPanel5 = new JPanel();
   JPanel searchOptionsPanel55 = new JPanel();
   JPanel searchOptionsPanel6 = new JPanel();
   JPanel searchOptionsPanel66 = new JPanel();
   JLabel option1L = new JLabel("                  Ontology level:   ",
JLabel.RIGHT);
   JSpinner option1F = new JSpinner(new SpinnerNumberModel(3, 2, 10, 1));
   JLabel option2L = new JLabel("Webs analized:          ", JLabel.RIGHT);
   JSpinner option2F = new JSpinner(new SpinnerNumberModel(100, 1, 1000, 10));
   JLabel option3L = new JLabel("Percentage webs:  ", JLabel.RIGHT);
   JSpinner option3F = new JSpinner(new SpinnerNumberModel(10, 0, 100, 5));
```

```
   JLabel option4L = new JLabel("                  Subclasses:       ",
JLabel.RIGHT);
   JSpinner option4F = new JSpinner(new SpinnerNumberModel(5, 1, 50, 1));
   JLabel option5L = new JLabel("Percentage/Number:", JLabel.RIGHT);
   JComboBox option5F = new JComboBox(new String[] { "        P     ", "
N     " });
   JLabel option6L = new JLabel("Number webs:         ", JLabel.RIGHT);
   JSpinner option6F = new JSpinner(new SpinnerNumberModel(10, 0, 100, 5));

   // JFrame > splitPane > treePanel >
   JLabel treeLabel = new JLabel("Ontology", JLabel.CENTER);
   JTree ontologyTree;
   DefaultTreeModel treeModel;
   ArrayList nodes = new ArrayList();

    // JFrame > splitPane > webPanel >
   JLabel tableLabel = new JLabel("Webs", JLabel.CENTER);
   JTextArea webArea = new JTextArea();

    ArrayList classes = new ArrayList();

    public OntologyBuilderGUI() {
        super(" Automatic Ontology Builder - Multi-Agent System ");
        this.setSize(994,758);
        this.setExtendedState(JFrame.MAXIMIZED_BOTH);

      // JFrame >
          Container content = this.getContentPane();
          content.setLayout(new BorderLayout());
          content.add(formPanel, BorderLayout.NORTH);
      content.add(splitPane, BorderLayout.CENTER);
      content.add(statusBar, BorderLayout.SOUTH);

          // JFrame > formPanel
      formPanel.setLayout(new BorderLayout());
          formPanel.add(searchPanel, BorderLayout.WEST);
          formPanel.add(searchOptionsPanel, BorderLayout.CENTER);

      // JFrame > splitPane >
      splitPane.setLeftComponent(treePanel);
      splitPane.setRightComponent(webPanel);
      splitPane.setDividerLocation(0.25);

      // JFrame > statusbar >
      statusBar.setLayout(new BorderLayout());
      statusBar.add(progressLabel, BorderLayout.WEST);

      // JFrame > formPanel > searchPanel >
      searchPanel.setLayout(new BorderLayout());
      searchPanel.add(northSearchPanel, BorderLayout.NORTH);
      searchPanel.add(ok, BorderLayout.SOUTH);
      ok.setEnabled(false);
      ok.addActionListener(this);
      ok.setActionCommand("search");

      // JFrame > formPanel > searchPanel > northSearchPanel >
      northSearchPanel.setLayout(new FlowLayout());
      northSearchPanel.add(searchLabel);
      northSearchPanel.add(searchField);
      document = searchField.getDocument();
      document.addDocumentListener(this);

      // JFrame > formPanel > searchOptionsPanel >
      searchOptionsPanel.setLayout(new GridLayout(2, 3));
      searchOptionsPanel.add(searchOptionsPanel11);
      searchOptionsPanel11.setLayout(new BorderLayout());
      searchOptionsPanel11.add(searchOptionsPanel1, BorderLayout.WEST);
      searchOptionsPanel1.setLayout(new FlowLayout());
```

```
      searchOptionsPanel1.add(option1L);
      searchOptionsPanel1.add(option1F);

((JSpinner.DefaultEditor)option1F.getEditor()).getTextField().setEditable(fals
e);
      option1L.setToolTipText("Maximum ontology depth level / " +
                              "Nivel de profundidad máximo de la ontología");

((JSpinner.DefaultEditor)option1F.getEditor()).getTextField().setToolTipText("
Maximum ontology depth level / " +
                                                                  "Nivel de
profundidad máximo de la ontología");
      searchOptionsPanel.add(searchOptionsPanel22);
      searchOptionsPanel22.setLayout(new BorderLayout());
      searchOptionsPanel22.add(searchOptionsPanel2, BorderLayout.WEST);
      searchOptionsPanel2.setLayout(new FlowLayout());
      searchOptionsPanel2.add(option2L);
      searchOptionsPanel2.add(option2F);

((JSpinner.DefaultEditor)option2F.getEditor()).getTextField().setEditable(fals
e);
      option2L.setToolTipText("Maximum number of analized webs per class / " +
                          "Número máximo de páginas analizadas por clase");

((JSpinner.DefaultEditor)option2F.getEditor()).getTextField().setToolTipText("
Maximum number of analized webs per class / " +
                                                                   "Número
máximo de páginas analizadas por clase");
      searchOptionsPanel.add(searchOptionsPanel33);
      searchOptionsPanel33.setLayout(new BorderLayout());
      searchOptionsPanel33.add(searchOptionsPanel3, BorderLayout.WEST);
      searchOptionsPanel3.setLayout(new FlowLayout());
      searchOptionsPanel3.add(option3L);
      searchOptionsPanel3.add(option3F);

((JSpinner.DefaultEditor)option3F.getEditor()).getTextField().setEditable(fals
e);
      option3L.setToolTipText("Minimum percentage of webs to generate a new
class / " +
                          "Porcentaje mínimo de páginas para que se genere una
nueva clase");

((JSpinner.DefaultEditor)option3F.getEditor()).getTextField().setToolTipText("
Minimum percentage of webs to generate a new class / " +
                                                                  "Porcentaje
mínimo de páginas para que se genere una nueva clase");
      searchOptionsPanel.add(searchOptionsPanel44);
      searchOptionsPanel44.setLayout(new BorderLayout());
      searchOptionsPanel44.add(searchOptionsPanel4, BorderLayout.WEST);
      searchOptionsPanel4.setLayout(new FlowLayout());
      searchOptionsPanel4.add(option4L);
      searchOptionsPanel4.add(option4F);

((JSpinner.DefaultEditor)option4F.getEditor()).getTextField().setEditable(fals
e);
      option4L.setToolTipText("Maximum number of subclasses per class / " +
                          "Número máximo de subclases por clase");

((JSpinner.DefaultEditor)option4F.getEditor()).getTextField().setToolTipText("
Maximum number of subclasses per class / " +
                                                                  "Número máximo
de subclases por clase");
      searchOptionsPanel.add(searchOptionsPanel55);
      searchOptionsPanel55.setLayout(new BorderLayout());
      searchOptionsPanel55.add(searchOptionsPanel5, BorderLayout.WEST);
      searchOptionsPanel5.setLayout(new FlowLayout());
      searchOptionsPanel5.add(option5L);
      searchOptionsPanel5.add(option5F);
```

```
      option5L.setToolTipText("Choice between minimum percentage or number of
webs to generate a new class / " +
                        "Elección entre porcentaje o número mínimo de páginas
para que se genere una nueva clase");
      option5F.setToolTipText("Choice between minimum percentage or number of
webs to generate a new class / " +
                        "Elección entre porcentaje o número mínimo de páginas
para que se genere una nueva clase");
      searchOptionsPanel.add(searchOptionsPanel66);
      searchOptionsPanel66.setLayout(new BorderLayout());
      searchOptionsPanel66.add(searchOptionsPanel6, BorderLayout.WEST);
      searchOptionsPanel6.setLayout(new FlowLayout());
      searchOptionsPanel6.add(option6L);
      searchOptionsPanel6.add(option6F);

((JSpinner.DefaultEditor)option6F.getEditor()).getTextField().setEditable(fals
e);
      option6L.setToolTipText("Minimum number of webs to generate a new class
/ " +
                        "Número mínimo de páginas para que se genere una nueva
clase");

((JSpinner.DefaultEditor)option6F.getEditor()).getTextField().setToolTipText("
Minimum number of webs to generate a new class / " +
                                                                "Número mínimo
de páginas para que se genere una nueva clase");

      // JFrame > splitPane > treePanel >
      treePanel.setColumnHeaderView(treeLabel);

      // JFrame > splitPane > webPanel >
      webPanel.setColumnHeaderView(tableLabel);

      this.setVisible(true);
    }

   public void addNewNode(ClassAndWebs subClass, String superClass){

      DefaultMutableTreeNode subClassNode, superClassNode;

      subClassNode = new DefaultMutableTreeNode(subClass.getWord());
      nodes.add(subClassNode);
      classes.add(subClass);
      if (superClass == null){
         // Aquí se empieza a mostrar el árbol
         treeModel = new DefaultTreeModel(subClassNode);
         ontologyTree = new JTree(treeModel);
         treePanel.setViewportView(ontologyTree);
         ontologyTree.setToggleClickCount(10);
         ontologyTree.addMouseListener(this);

         webPanel.setViewportView(webArea);
         webArea.setEditable(false);
      }
      else{
         Iterator i = nodes.iterator();
         while (i.hasNext()){
            superClassNode = (DefaultMutableTreeNode)i.next();
            if
(((String)superClassNode.getUserObject()).equalsIgnoreCase(superClass)){
               treeModel.insertNodeInto(subClassNode, superClassNode,
treeModel.getChildCount(superClassNode));
               break;
            }
         }
      }
   }
```

```
   public void changedUpdate(DocumentEvent e ) {

     if (e.getDocument()==document)
      disableIfEmpty(document);
   }

   public void insertUpdate( DocumentEvent e ){

     if (e.getDocument()==document)
      disableIfEmpty(document);
   }

   public void removeUpdate( DocumentEvent e ){

     if (e.getDocument()==document)
      disableIfEmpty(document);
   }

   public void disableIfEmpty(Document d) {

      ok.setEnabled(d.getLength() > 0);
   }

    public void actionPerformed(ActionEvent ae) {
        if (ae.getActionCommand().equals("search")) {
             treeLabel.setText(searchField.getText().toUpperCase() + "
ontology");
         agent.startBuilding();
         progressLabel.setText("      Building . . .");
        }
    }

   public void mouseClicked(MouseEvent e){

      TreePath path = ((JTree)e.getComponent()).getPathForLocation(e.getX(),
e.getY());//.leadSelectionPath();
      DefaultMutableTreeNode node = null;
      ClassAndWebs caw;
      Vector url = null;
      Vector col = new Vector();

      if (SwingUtilities.isLeftMouseButton(e)) {
         if (e.getClickCount() == 2) {
            if (path != null){
               node = (DefaultMutableTreeNode)path.getLastPathComponent();
               Iterator it = classes.iterator();
               while (it.hasNext()){
                  caw = (ClassAndWebs)it.next();
                  if
(caw.getWord().equalsIgnoreCase((String)node.getUserObject())){
                     url = new Vector(caw.getUrl().toList());
                     break;
                  }
               }

tableLabel.setText(((String)node.getUserObject()).toUpperCase() + " webs");
               webArea.setText("");
               int webNumber = 1;
               Iterator itt = url.iterator();
               while (itt.hasNext()){
                  webArea.append(" " + webNumber + "- " + (String)itt.next() +
"\n");
                  webNumber ++;
               }
            }
         }
         else if (e.getClickCount() == 1) {
            if (path != null){
```

```
                if (ontologyTree.isCollapsed(path)){
                   ontologyTree.expandPath(path);
                }
                else{
                   ontologyTree.collapsePath(path);
                }
             }
          }
       }
   }

   public void mouseEntered(MouseEvent e){}

   public void mouseExited(MouseEvent e){}

   public void mousePressed(MouseEvent e){ }

   public void mouseReleased(MouseEvent e){}
}
```

A.2- Paquete Ontology

A.2.1- WebOntology.java

```
package Ontology;

import jade.content.onto.*;
import jade.content.schema.*;

public class WebOntology extends Ontology{

   public static final String ONTOLOGY_NAME = "Web-Ontology";
   public static final String CLASSANDWEBS = "ClassAndWebs";
   public static final String REQUEST = "Request";
   public static final String INFORM = "Inform";
   public static final String SEARCHWORD = "SearchWord";

   private static Ontology theInstance = new WebOntology();
   public static Ontology getInstance() {
      return theInstance;
   }

   public String getName() {
      return ONTOLOGY_NAME;
   }

   private WebOntology(){
      super(ONTOLOGY_NAME, BasicOntology.getInstance());

        try {
         PrimitiveSchema stringSchema =
(PrimitiveSchema)getSchema(BasicOntology.STRING);
         PrimitiveSchema integerSchema =
(PrimitiveSchema)getSchema(BasicOntology.INTEGER);
         PrimitiveSchema booleanSchema =
(PrimitiveSchema)getSchema(BasicOntology.BOOLEAN);

         add(new ConceptSchema(CLASSANDWEBS), ClassAndWebs.class);
         add(new ConceptSchema(REQUEST), Request.class);
         add(new ConceptSchema(INFORM), Inform.class);
         add(new AgentActionSchema(SEARCHWORD), SearchWord.class);

         ConceptSchema cs = (ConceptSchema)getSchema(CLASSANDWEBS);
         cs.add("word", stringSchema);
         cs.add("count", integerSchema);
         cs.add("url", stringSchema, 0, ObjectSchema.UNLIMITED);

         cs = (ConceptSchema)getSchema(REQUEST);
         cs.add("search", stringSchema);
         cs.add("level", integerSchema);
         cs.add("numberMaxWebs", integerSchema);
         cs.add("numberOfSubclasses", integerSchema);
         cs.add("percentageOfWebs", integerSchema);
         cs.add("numberOfWebs", integerSchema);
         cs.add("percentageOrNumber", booleanSchema);

         cs = (ConceptSchema)getSchema(INFORM);
         cs.add("subClasses", (ConceptSchema)getSchema(CLASSANDWEBS), 0,
ObjectSchema.UNLIMITED);
         AgentActionSchema as = (AgentActionSchema)getSchema(SEARCHWORD);
         as.add(REQUEST, (ConceptSchema)getSchema(REQUEST));
      }
        catch(OntologyException oe) {
           oe.printStackTrace();
      }
   }
}
```

A.2.2- Request.java

```
package Ontology;

import jade.content.*;

public class Request implements Concept {

   private String search; // Nombre de la clase (palabra que el agente
buscador tendrá que buscar en el google)
   private int level; // Nivel del árbol y número de palabras a buscar
   private int numberMaxWebs; // Número máximo de páginas a considerar de cada
clase
   private int numberOfSubclasses; // Número máximo de subclases de una clase
   private int percentageOfWebs; // Porcentaje de páginas a partir del cual se
genera una nueva clase
   private int numberOfWebs; // Número de páginas a partir del cual se genera
una nueva clase
   private boolean percentageOrNumber; // Elección entre porcentaje o número
de páginas a partir del cual se genera una nueva clase
                              // true: percentage / false: number

   public Request() {
      search = null;
   }

   public String getSearch(){
      return search;
   }

   public void setSearch (String s){
      search = s;
   }

   public int getLevel(){
      return level;
   }

   public void setLevel (int l){
      level = l;
   }

   public int getNumberMaxWebs(){
      return numberMaxWebs;
   }

   public void setNumberMaxWebs (int n){
      numberMaxWebs = n;
   }

   public int getNumberOfSubclasses(){
      return numberOfSubclasses;
   }

   public void setNumberOfSubclasses (int n){
      numberOfSubclasses = n;
   }

   public int getPercentageOfWebs(){
      return percentageOfWebs;
   }

   public void setPercentageOfWebs (int p){
      percentageOfWebs = p;
   }
```

```
   public int getNumberOfWebs(){
      return numberOfWebs;
   }

   public void setNumberOfWebs (int n){
      numberOfWebs = n;
   }

   public boolean getPercentageOrNumber(){
      return percentageOrNumber;
   }

   public void setPercentageOrNumber (boolean p){
      percentageOrNumber = p;
   }
}
```

A.2.3- Inform.java

```
package Ontology;

import jade.content.*;
import jade.util.leap.*;

public class Inform implements Concept {

   // Nombres de las nuevas clases (palabras seleccionadas) y webs asociadas
   private ArrayList subClasses = new ArrayList();

   public Inform() {
   }

   public ArrayList getSubClasses()  {
      return subClasses;
   }

   public void setSubClasses(ArrayList l) {
      subClasses = l;
   }

   public void addSubClass (ClassAndWebs s){
      subClasses.add(s);
   }
}
```

A.2.4- SearchWord.java

```
package Ontology;

import jade.content.*;

public class SearchWord implements AgentAction {

   private Request req;

   public SearchWord() {
   }

   public Request getRequest() {
     return req;
   }

   public void setRequest(Request r) {
      this.req = r;
   }
}
```

A.2.5- ClassAndWebs.java

```
package Ontology;

import jade.content.*;
import jade.util.leap.*;

public class ClassAndWebs implements Concept {

   private String word;
   private int count;
   private ArrayList url = new ArrayList();

   public ClassAndWebs() {
      count = 0;
   }

   public String getWord(){
      return word;
   }

   public void setWord (String w){
      word = w;
   }

   public int getCount() {
      return count;
   }

   public void setCount (int c){
      count = c;
   }

   public ArrayList getUrl()  {
      return url;
   }

   public void setUrl(ArrayList s) {
      url = s;
   }

   public void addUrl (String u){
      url.add(u);
   }
}
```

A.2.6- IntegerComparator.java

```
package Ontology;

import java.util.*;

public class IntegerComparator implements Comparator {

      public int compare (Object obj1, Object obj2) {
         int s1 = ((ClassAndWebs) obj1).getCount();
         int s2 = ((ClassAndWebs) obj2).getCount();

         if (s1 < s2) return 1;
         else if (s1 > s2)
            return -1;
         else return 0;
      }
}
```