PROYECTO FINAL DE CARRERA

Sistema Multi-Agente para la construcción semiautomática de Ontologías a partir de la Web.

Lorena Morales Padillo
Francisco Ruiz Cucharero
Ingeniería Técnica en Informática de Sistemas

Directores de proyecto:
Sr. David Sánchez Ruenes
Dr. Antonio Moreno Ribas

Escola Tècnica Superior d'Enginyeria (ETSE)
Universidad Rovira i Virgili (URV)

ÍNDICE

1- INTRODUCCIÓN

En la actualidad, Internet ofrece acceso a una cantidad enorme de información, convirtiéndose en una de las principales fuentes de conocimiento, 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.

Sería perfecto poder disponer de esta información de forma estructurada y clasificada, de tal forma que solo nos apareciera aquella información que buscásemos o que rápidamente pudiésemos separar lo que necesitamos de lo que no. Obviamente esta estructuración es imposible manualmente, hay millones de páginas Web y la mayoría tratan varios temas, por eso se ha de recurrir a métodos informáticos de clasificación. Existen buscadores (p.e. Google, Altavista) que revisan una gran cantidad de páginas, y nos muestran todas aquellas que contienen una combinación de palabras concreta. Esta solución es muy rápida y útil pero requiere un esfuerzo considerable, por parte del usuario, de análisis del contenido, ya que devuelve muchas páginas y en la mayoría de los casos se ha de repetir la búsqueda con patrones diferentes (combinación de palabras). Es decir esta es una solución puramente mecánica que simplemente busca en cada Web si aparece o no la palabra deseada. A pesar de esto, resaltar el alto rendimiento que ofrecen estos buscadores obteniendo resultados en décimas de segundo y destacar la gran utilidad que tienen hoy en día.

Una forma de mejorar el resultado de los sistemas de búsqueda actuales, es buscar el concepto y no como se representa éste. Esto se puede lograr gracias a técnicas de Inteligencia Artificial, con lo que ahora ya no estamos buscando una cadena de símbolos si no que buscamos conocimiento, que esta almacenado en páginas Web. Este cambio de razonamiento permite que Internet sea potencialmente más útil y abre nuevas posibilidades para la investigación.

Este proyecto, a partir de los resultados de estos sistemas actuales de búsqueda, en concreto los de Google, intenta mostrar al usuario, de una manera inteligente, únicamente la información que desee ver, no pretende ser un buscador, sino una herramienta de trabajo que sea útil a la hora de investigar o profundizar sobre un tema.
Pongamos por caso que un médico necesita investigar en profundidad un tipo concreto de cáncer, deseando obtener información de los síntomas. En los buscadores convencionales, este doctor introduciría "síntoma de cáncer de pulmón" obteniendo en menos de un segundo, millones de paginas que en el caso de que lo hubiese escrito sin comillas, contendrían "síntoma de cáncer de pulmón", "cáncer", "síntoma" o "pulmón", entre los 10 primeros resultados podría haber una pagina que trata del signo del zodiaco, otra que seria de la asociación de ayuda contra el cáncer, que si tiene relación pero no es lo que busca el doctor, y es su trabajo ir mirando una por una para averiguar de que trata esa Web. Probablemente tendrá que repetir la búsqueda quizás introduciendo "investigación sobre cáncer de pulmón" o algo parecido. Si lo hubiese escrito entre comillas, el buscador hubiese mostrado solo las páginas que contuviesen la frase escrita tal cual, y en este caso concreto, hubiese mostrado unas pocas páginas.
En sistemas de búsqueda inteligente, sería interesante, que este doctor al introducir "cáncer" en el dominio médico, obtuviese un árbol donde estuvieran clasificados los conceptos de las páginas Web encontradas, según la información analizada en ellas. De este resultado, el doctor seleccionaría lo que le interesara, en este caso "pulmón", y el

sistema le recomendaría una serie de conocimientos que pudiesen ser de su interés: “síntomas”, “causa”, “tratamiento”..., él buscaría “síntomas”, obteniendo otra estructura que albergaría los síntomas de los resultados anteriores (pero solo los elegidos por el usuario) y los nuevos. Así pues, el doctor ha encontrado un considerable número de webs clasificadas según su necesidad.
Este sistema sería útil para expertos, como en el caso del doctor, y también para quienes no tuviesen mucho conocimiento de un dominio, ya que el sistema los iría guiando de una manera sencilla, proponiéndole nuevos conceptos relacionados con el introducido por él.

Esto no es una tarea fácil, computacionalmente puede llegar a ser bastante difícil y costoso. Para poder alcanzar los objetivos, descritos en la siguiente sección, se ha de recurrir a técnicas de Inteligencia Artificial.
En primer lugar, el sistema usa una librería, desarrollada por David Sánchez Ruenes [Sánchez, 2005], capaz de construir taxonomías, clasificación jerárquica de un dominio concreto, según los parámetros introducidos por el usuario a partir de los recursos Web. Es decir, es la encargada de buscar en Internet, mediante Google, el concepto, introducido por el usuario, a partir del cual se construirá la Ontología, y generar una taxonomía del concepto, analizando (proceso descrito posteriormente) los recursos Web devueltos por Google. Así pues, hacemos uso de esta potente herramienta para construir, a partir de sus resultados parciales, la Ontología final, a base de reiteradas peticiones a ésta. Por lo que podríamos decir que este proyecto es una posible aplicación de este constructor de taxonomías. Pero esta librería es muy costosa de ejecutar, es decir, es inviable para una ejecución “centralizada”, en un solo computador, por lo tanto se usan agentes, para conseguir buena escalabilidad distribuyendo la carga del sistema entre varias máquinas. Un Agente es una unidad “Inteligente”, que reúne una serie de características (descritas con detalle posteriormente) que lo hacen muy interesante para la resolución de problemas distribuidos. Una de las características más interesantes es su capacidad de comunicación con otros Agentes, y juntos resolver un problema complejo, formando un sistema multi-Agente. Por lo tanto, estos sistemas nos ofrecen un entorno idóneo para este proyecto ya que proporcionan concurrencia, modularidad a nivel de problemas, escalabilidad, etc. En conclusión, basando el diseño del proyecto en estos sistemas podremos alcanzar los objetivos, con una cierta calidad, y pensado en un futuro podríamos, gracias a ellos, repartir la carga entre varios computadores.

Para representar estas Ontologías utilizamos el lenguaje OWL (Ontology Web Language), este ha sido creado precisamente para representar conocimiento con el fin de que una computadora lo pueda tratar, además es específico para representar Ontologías de recursos Web. Así pues, gracias a este lenguaje el sistema será capaz de tratar este conocimiento de un cierto dominio para posteriormente mostrárselo al usuario lo más claro posible.

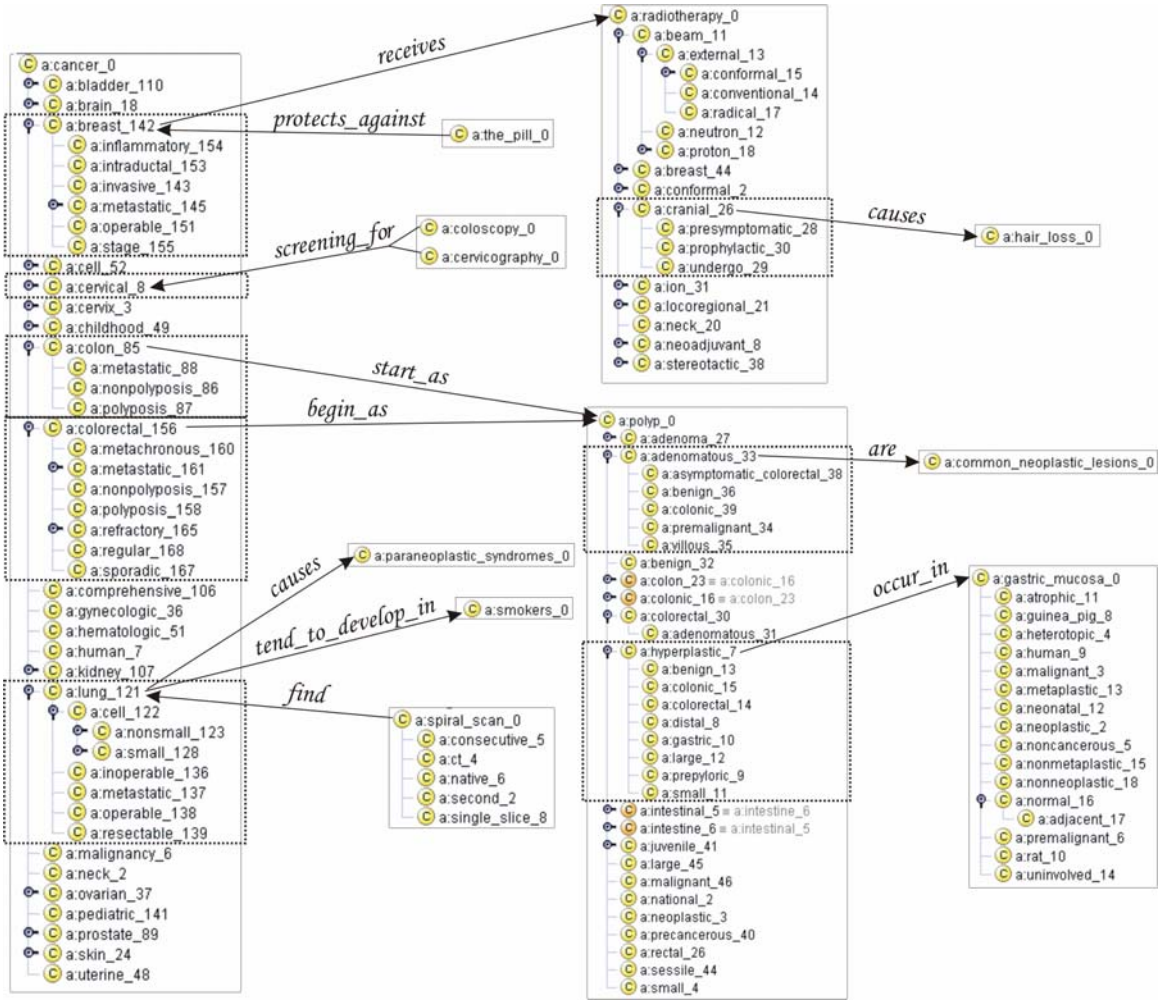
Concluyendo, se pretende diseñar una herramienta de búsqueda, que permita al usuario guiar la búsqueda y construir su Ontología (conceptualización de la información). A diferencia de los buscadores actuales, esta búsqueda no será en tiempo real, sino que puede llegar a ser un proceso largo, pero que una vez finalizada, la Ontología, el usuario podrá acceder a los resultados de forma inmediata. Es decir, es una filosofía diferente de búsqueda intentando proporcionar una alternativa más a los sistemas actuales.

1.1- Objetivos del proyecto

La finalidad de este proyecto es crear, con técnicas de Inteligencia Artificial, un constructor de Ontologías semiautomático a partir de la Web. Es decir, una aplicación Web que ayude a la construcción de estructuras semánticas y categorice los recursos Web (páginas Web, docs, pdf, etc), en función de los resultados obtenidos de una búsqueda en Internet.

Concretamente, se pretende construir una aplicación Web, donde el usuario introduce el dominio a buscar. Mediante el taxoBuilder (nombre de la librería constructora de taxonomías) se hace una primera evaluación de ese dominio y se devuelve unos resultados iniciales (clases, subclases, instancias, webs y frases que relacionan ese dominio con otros conceptos). El usuario puede evaluar esos resultados (mediante una interfaz Web), añadir nuevos conceptos, derivados de las frases obtenidas, a la Ontología (de cuyo manejo se encarga un módulo especialmente diseñado) que él va construyendo y pedir que se hagan nuevas búsquedas de los nuevos conceptos que le interesen, lo que conllevará sucesivas ejecuciones del taxobuilder de forma concurrente con diferentes parámetros, de ahí la conveniencia de usar una aproximación distribuida basada en agentes.

Un ejemplo del proceso de construcción podría ser el siguiente, el usuario introduce “cáncer”, como concepto a evaluar, obteniendo la correspondiente estructura de clases y subclases, webs asociadas, instancias, y las frases que el constructor de taxonomías consideró relevantes para el concepto inicial. En la estructura de clases, estructuración del conocimiento, aparecen conceptos como “breast”, “colon”, “colorectal”, etc. Haciendo click sobre cualquiera de ellas, podremos observar las frases relevantes, de esa clase, introduciendo nuevos conceptos. Por ejemplo “colon start as polyp” o “breast relieve radiotherapy”, así pues “polyp” y “radiotherapy” son nuevos conceptos que el sistema propone para ser evaluados e incluirlos en la Ontología de “cáncer”. Para hacer esto el usuario puede decidir si buscarlas para obtener mas información o simplemente incluir ese concepto en la estructura, si decide buscarlo se añadirá a la Ontología de “cáncer” la Ontología derivada del nuevo concepto. Este proceso se puede repetir tantas veces como el usuario desee para poder construir una Ontología tan completa como él necesite. En la siguiente figura se puede observar esquemáticamente este proceso y como se va construyendo la Ontología a partir de las frases propuestas.



Pero además, también devuelve, dentro de la estructura de clases, instancias, estas son clases de una entidad, por ejemplo “Asociación Nacional De Lucha Contra El Cáncer” seria una clase del concepto cáncer pero es, por si sola, una entidad, con lo cual no tiene subclases ni frases pero si su recursos Web (si tiene).

Para alcanzar esta meta de forma óptima, se han marcado los siguientes requisitos:

a) La aplicación esta basada en un sistema multi-Agente para construir, de forma incremental, una taxonomía (estructura jerárquica) de conceptos a partir de la Web. De esta forma obtendremos una aplicación distribuida, concurrente y escalable, ya que los Agentes se pueden ejecutar en diferentes máquinas y todos se ejecutan a la vez manteniendo una comunicación fluida.
Este sistema multi-Agente utiliza una herramienta, capaz de construir una taxonomía a partir de texto natural, sin estructurar, contenida en páginas Web y según los parámetros de la búsqueda. Esta librería utiliza muchos recursos (memoria y CPU) y no es recomendable ejecutar mas de dos taxoBuilder en una sola máquina, es por esto, que haría inviable su ejecución en un sistema centralizado, y que hace tan interesante basar el diseño en un sistema distribuido como es el caso de los sistemas multi-Agente.

b) Todo el procesado y composición de la Ontología esta basada en librerías OWL (Ontology Web Language). Precisamente OWL fue pesado y concebido, por W3C, para construir y tratar ontologías, hecho que se ajusta perfectamente a nuestras pretensiones. Pero además de los métodos proporcionados por OWL es necesario crear procesos que unan Ontologías, añadan propiedades (relaciones no taxonómicas) a estas, o que extraigan el contenido almacenado en ellas.

c) Finalmente, la aplicación ha de ser interactiva, mostrando gráficamente los resultados parciales y totales de las Ontologías. además, se ha decidido, que sea una aplicación Web, parecida a un buscador Web, con el fin de proporcionar las ventajas de dichas aplicaciones. Otra característica que ha de cumplir es que una Ontología no se finaliza hasta que el usuario desee, así pues la aplicación Web ha de guardar los resultados parciales e ir construyendo las nuevas Ontologías entrelazando los nuevos resultados con los anteriores, además de permitir buscar nuevos conceptos. Por último, debido a que todo el proceso puede se muy largo, el sistema permite al usuario desconectarse de él mientras trabaja y para cuando el usuario se vuelva a conectar directamente se le muestre los resultados si es que ya están disponibles.

Se ha considerado separar el diseño en dos partes bien diferenciadas considerando que se puede obtener un mejor resultado de cada una de ellas haciendo un conjunto mejor. Así pues Francisco Ruiz Cucharero es el encargado del estudio y diseño del sistema multi-Agente, además de integrar en el sistema el constructor de taxonomías, que como ya hemos mencionado ha sido desarrollado por David Sánchez Ruenes. Por otro lado Lorena Morales Padillo es la encargada de la implementación de la interficie Web junto con la representación de los resultados y la creación de la Ontología a partir de los resultados del constructor de taxonomías.

Un ejemplo de aplicación de este proyecto, es en el área de desarrollo e investigación, convirtiéndose en una potente herramienta para recopilar información sobre un dominio concreto en Internet, recibiendo resultados de gran provecho para la ciencia, gracias a la estructuración de los conceptos obtenidos y sobretodo de la facilidad que ofrece, la aplicación, para seguir construyendo la Ontología. Así pues, un experto en la materia puede ir construyendo la Ontología de una forma intencionada hacia el objetivo final de su investigación. Obteniendo no solo la información concreta de cada término en sus correspondientes webs sino que además visualizará de forma intuitiva la relación de los conceptos y las posibles lindas de investigación de cada uno.
Con relación a esto, otra posible aplicación de este trabajo es como **webs e-learning,** herramienta de aprendizaje para usuarios no expertos en un dominio concreto. En este sentido, un usuario puede explorar una determinada área del conocimiento, de forma interactiva, seleccionando nuevos conceptos, descubriendo importantes términos y como estos están relacionados, y finalmente acceder a las páginas Web que contienen esa especifica información.

Otro punto de vista, detectando instancias de conceptos a través de la extracción nombre-entidad, puede ser de gran utilidad para estudios de mercado, recuperando importantes compañías y organizaciones y sus asociados recursos Web. Además estas instancias son seleccionadas sin las clásicas restricciones (p.e. “organizaciones”, “personas”,..) permitiendo detectar todo tipo de entidades y eventos.

1.2- 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 detallaremos las tecnologías Web utilizadas .En el capítulo 6 analizaremos el diseño del sistema. En el capítulo 7 se detallan las pruebas realizadas para la evaluación del sistema. En el capítulo 8 se explican las conclusiones sobre el proyecto y el trabajo futuro. En el capítulo 9 se comentan los recursos utilizados. El capítulo 10 está destinado a los manuales de usuario.

2- AGENTES Y SISTEMAS MULTI-AGENTE

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- FIPA

La FIPA (*Foundation for Intelligent Physical Agents*) fue fundada en 1996 para producir software estándar para agentes heterogéneos e interactivos y sistemas basados en agentes.
Considerando la estandarización que FIPA ha producido (visitar http://www.fipa.org), esta claro que FIPA consiguió su objetivo. Sin embargo, ahora es el momento de trasladar éste estándar, para agentes y sistemas basados en agentes, hacia el más amplio contexto del desarrollo de software en general. En breve, la tecnología de agentes necesitará integrarse con el resto de tecnologías siguiendo estas normas. De ésta perspectiva la IEEE Computer Society a invitado FIPA tomar parte de la familia de comités estandarizadotes (http://www.computer.org/standards/). En Marzo del 2005, la Junta directiva de FIPA presentó ésta oportunidad al resto de los miembros de FIPA, quienes unánimemente votaron a favor de la adhesión de FIPA al IEEE Computer Society. FIPA pronto será el undécimo comité estandarizador del IEEE, y será conocido como FIPA Standard Committee (comité estandarizador). Esta transición coincidirá con la clausura de FIPA como organización suiza.
El IEEE FIPA Standard Committee actuará como una organización propia, con sus propias políticas, procedimientos, estructuras de deudas y cuentas bancarias sin la IEEE. La IEEE Computer Society le proporcionará la fachada de la organización, mantenimiento de los sitios Web, y todas las otras ventajas que una gran organización estandarizadora proporciona.

2.2- 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.

2.3- 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.

2.4- 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.

2.5- 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.

2.6- 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.

2.6.1- 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

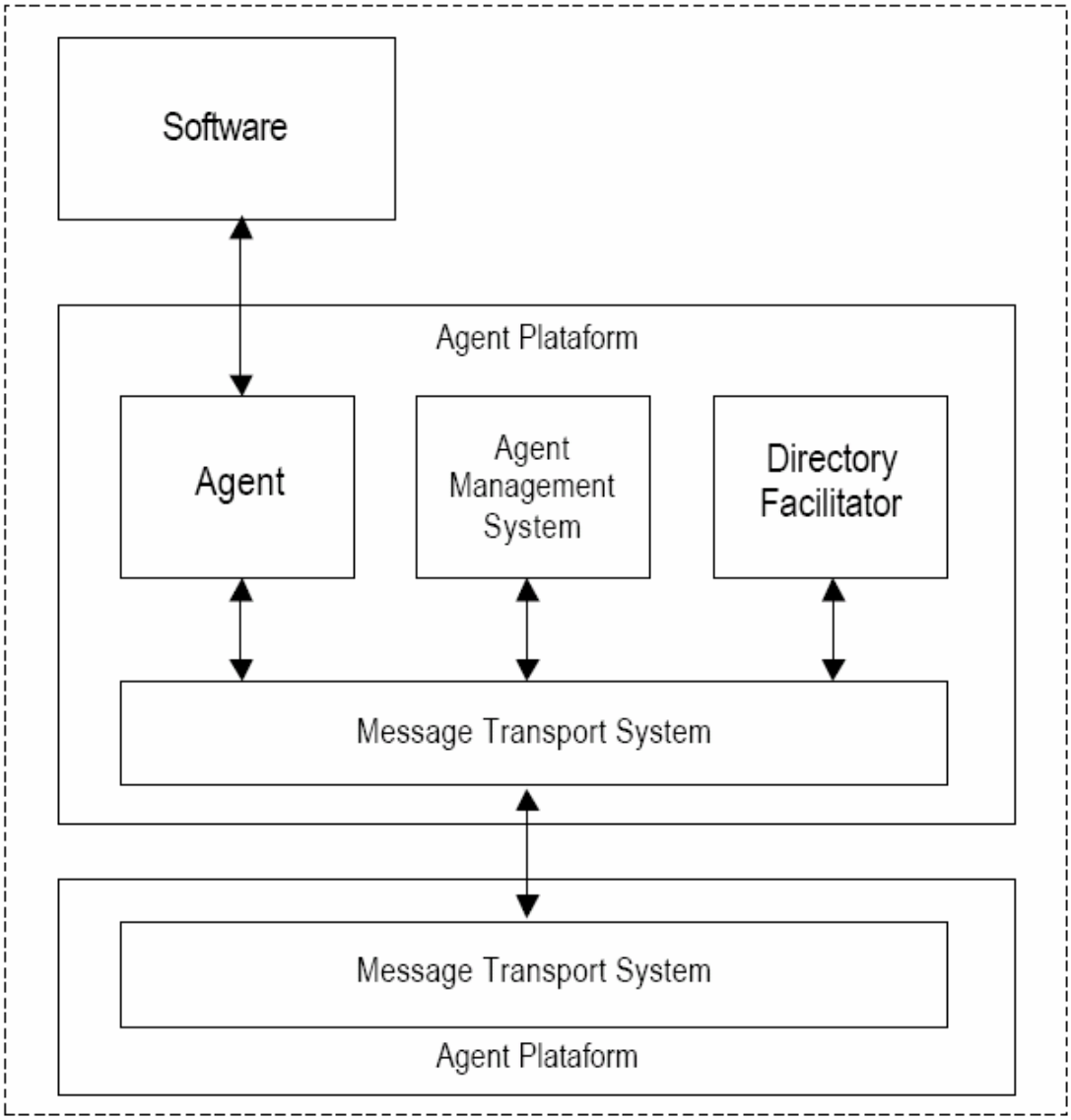
- **Eficiencia**: la programación distribuida permite repartir las tareas entre los agentes, consiguiendo paralelismo (agentes trabajando en diferentes máquinas).

- **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.

2.6.2- 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.

2.6.3- 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).

2.6.3.1- 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.

2.6.3.2- 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*) (Bellifemine,2003) 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 3.3.

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)

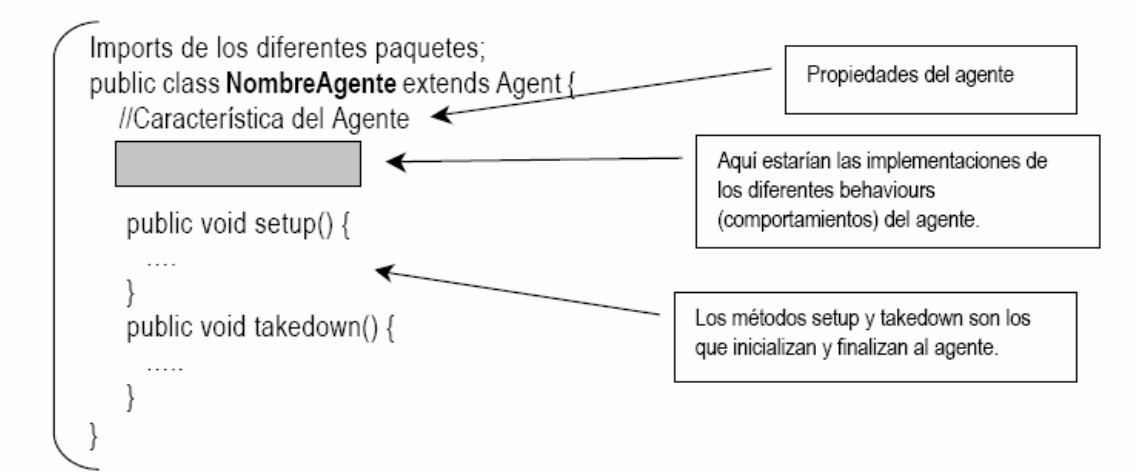
Para poder crear agentes en JADE, debemos extender la clase *jade.core.Agent*, que ya está definida, y rellenar los campos y las funciones básicas que nos proporciona la interfaz. Dos métodos muy importantes que se tienen que implementar son **setup()** y **takedown()** que inicializan y finalizan el 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 [Gómez-Pérez, Facultad de Inf. de Madrid] 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 u 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:

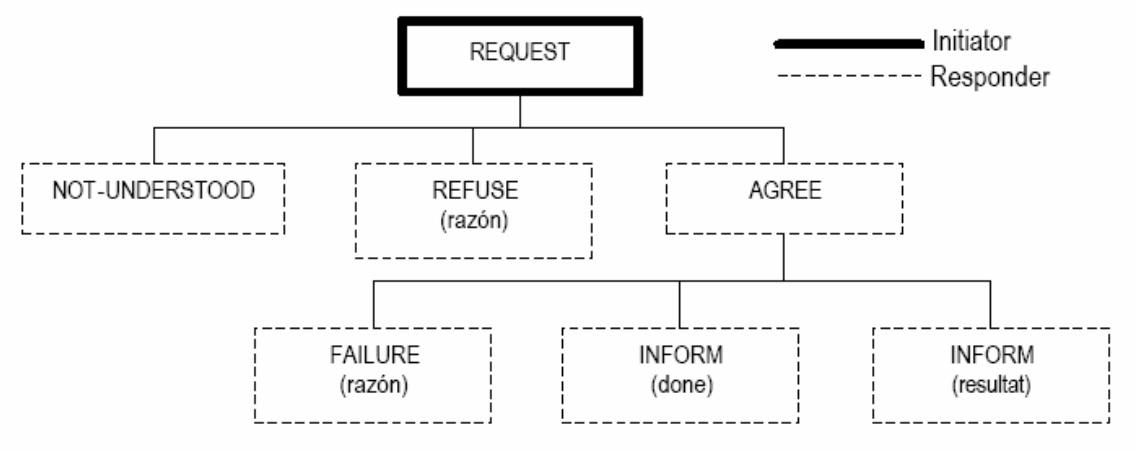
- El agente iniciador envía un mensaje de tipo *Request* al agente receptor para que realice una acción determinada.

- 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

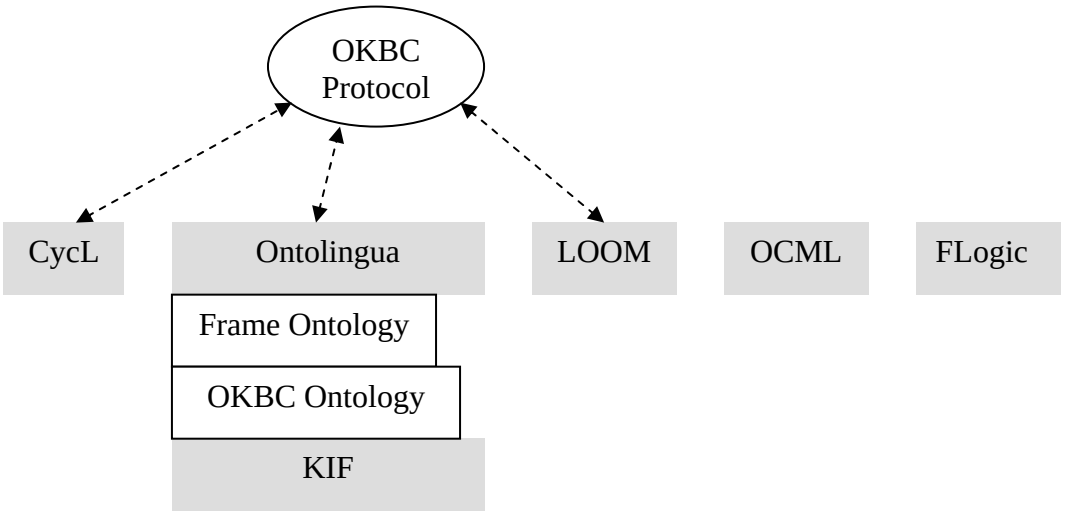
4.2.1- Introducción

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

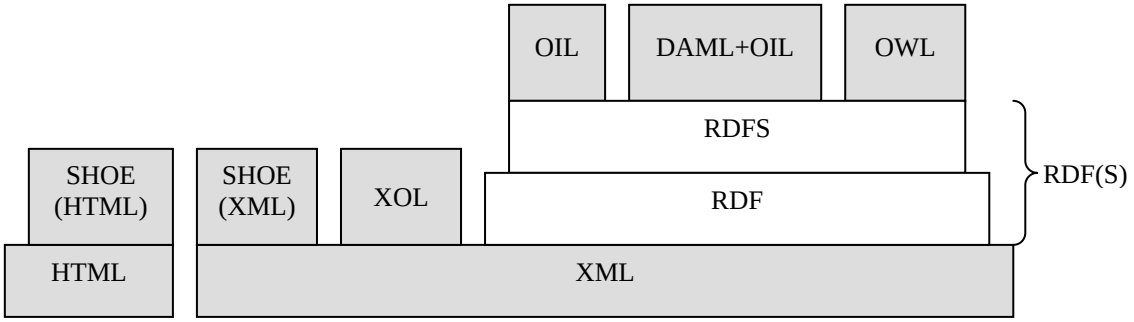
Del anterior conjunto de lenguajes, CycL (Lenat y Guha, 1990) fue el primero en ser creado. CycL está basado en marcos y primer orden lógico, y se utilizó para construir la Ontología Cyc.

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’.

Son varios los motivos que nos han llevado a utilizar este lenguaje para representar la Ontología. En primer lugar la librería taxoBuilder (constructor de ontologías) nos devuelve el resultado de sus taxonomías descritas en este lenguaje, así pues, con tal de ser compatibles con ella hemos preferido no cambiar de lenguaje. Por otro lado, la filosofía de este lenguaje es ideal para nuestro propósito, ya que esta pensado para representar conocimiento de recursos Web.

así pues, utilizamos este lenguaje para tratar con los resultados de taxoBuilder, ya sea para mostrarlos, teniendo en cuenta todos los parámetros que definen esa Ontología (clases, subclases, frases, instancias, paginas Web, etc), como también para el proceso de construcción de de la nueva Ontología a través de unir resultados de taxoBuilder (proceso descrito con detalle posteriormente).

A pesar de que a priori parece un lenguaje ideal para nuestro sistema, tiene algunas limitaciones derivadas de su corta evolución y de su diseño. El problema viene cuando queremos almacenar datos y estos contienen ciertos caracteres (&, <, >) entre las etiquetas, como por ejemplo las paginas Web relacionadas con cada clase. Esto es debido a que OWL no esta pensado para realizar esta tarea.
Por otro, lado también se desea conseguir interoperabilidad pudiendo mostrar los resultados en otros sistemas similares, como es el caso de taxoBuilder y este proyecto.

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.

El uso de esta herramienta dentro del proyecto es puramente de verificación de las ontologías creadas por nuestro sistema, una Ontología en OWL esta bien creada solo si Protégé-2000 es capaz de leerlo, así permite detectar los errores derivados de almacenar datos dentro la Ontología.

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- TECNOLOGÍA WEB

La aplicación Web está formada por distintas tecnologías: html, páginas dinámicas, javaScript, applet,…, explicadas a continuación.

5.1 – Páginas dinámicas

Las páginas dinámicas son páginas HTML generadas a partir de lenguajes de programación (scripts) que son ejecutados en el propio servidor Web. A diferencia de otros scripts, como el JavaScript, que se ejecutan en el propio navegador del usuario, los 'Server Side' scripts generan un código HTML desde el propio servidor Web.

Este código HTML puede ser modificado, por ejemplo, en función de una petición realizada por el usuario en una Base de Datos. Dependiendo de los resultados en la Base de Datos, se generará un código HTML u otro, mostrando diferentes contenidos.
En nuestro caso, utilizamos JSP (*Java Server Pages*) para el manejo de las cookies, para arrancar y controlar los agentes, como parte de la interficie gráfica y para arrancar el applet.

JSP y Servlets

Los *servlets* y *Java Server Pages (JSPs)* son dos métodos de creación de páginas Web dinámicas en servidor usando el lenguaje Java. En ese sentido son similares a otros métodos o lenguajes tales como el PHP, los CGIs (common gateway interface), programas que generan páginas Web en el servidor, o los ASP (Active Server Pages), un método específico de Microsoft. Sin embargo, se diferencian de ellos en otras cosas. Para empezar, los JSPs y servlets se ejecutan en una máquina virtual Java, lo cual permite que, en principio, se puedan usar en cualquier tipo de ordenador, siempre que exista una máquina virtual Java para él. Cada servlet (o JSP, a partir de ahora lo usaremos de forma indistinta) se ejecuta en su propia hebra, es decir, en su propio contexto, pero no se comienza a ejecutar cada vez que recibe una petición, sino que persiste de una petición a la siguiente, de forma que no se pierde tiempo en invocarlo (cargar programa + intérprete). Su persistencia le permite también hacer una serie de cosas de forma más eficiente: conexión a bases de datos y manejo de sesiones, por ejemplo.

Los JSPs son en realidad servlets: un JSP se compila a un programa en Java la primera vez que se invoca, y del programa en Java se crea una clase que se empieza a ejecutar en el servidor como un servlet. La principal diferencia entre los servlets y los JSPs es el enfoque de la programación: un JSP es una página Web con etiquetas especiales y código Java incrustado, mientras que un servlet es un programa que recibe peticiones y genera a partir de ellas una página Web

Ambos necesitan un programa que los contenga, y sea el que envíe efectivamente páginas Web al servidor, y reciba las peticiones, las distribuya entre los servlets, y lleve a cabo todas las tareas de gestión propias de un servidor Web. Mientras que servidores como el Apache están especialmente pensados para páginas Web estáticas CGIs, y programas ejecutados por el servidor, tales como el PHP, hay otros servidores

específicos para servlets y JSPs llamados *contenedores de servlets (servlet containers)* o *servlet engines*. Los principales son los siguientes:

- Resin, de Caucho Technologies, un motor especialmente enfocado al servicio de páginas XML, con una licencia libre para desarrolladores. Dice ser bastante rápido. Incluye soporte para Javascript además de Java. Incluye también un lenguaje de templates llamado XTP. Es bastante fácil de instalar, y en dos minutos, se pueden empezar a servir páginas JSP.

- BEA Weblogic es un servidor de aplicaciones de alto nivel, y también de alto precio. Está escrito íntegramente en Java, y se combina con otra serie de productos, tales como Tuxedo, un servidor de bases de datos para XML.

- JRun, de Macromedia, un servidor de aplicaciones de Java, de precio medio y probablemente prestaciones medias. Se puede bajar una versión de evaluación gratuita

- Lutris Enhydra, otro servidor gratuito y Open Source, aunque tiene una versión de pago. También enfocado a servir XML, y para plataformas móviles. Las versiones más actualizadas son de pago, como es natural

- El más popular, Open Source, y continuamente en desarrollo, es el Jakarta Tomcat, del consorcio Apache, un contenedor de servlets con muchos desarrollos adicionales alrededor; por ejemplo, Cocoon para servir páginas XML. Puede servir páginas sólo o bien como un añadido al servidor Apache. Es Open Source, relativamente rápido, y fácil de instalar. La versión actual es la 5.5.9.

5.2- Cookies

Las Cookies son una forma para que un servidor (o un servlet, como parte de un servidor) envíe información al cliente para almacenarla, y para que el servidor pueda posteriormente recuperar esos datos desde el cliente. Los servlet envían cookies al cliente añadiendo campos a las cabeceras de respuesta HTTP. Los clientes devuelven las cookies automáticamente añadiendo campos a las cabeceras de peticiones HTTP.

Cada cabecera de petición o respuesta HTTP es nombrada como un sólo valor. Por ejemplo, una cookie podría tener un nombre de cabecera **BookToBuy** con un valor **304qty1**, indicando a la aplicación llamante que el usuario quiere comprar una copia del libro con el número 304 en el inventario. (Las cookies y sus valores son específicos de la aplicación).

Varias cookies pueden tener el mismo nombre. Por ejemplo, un servlet podría enviar dos cabeceras llamadas **BookToBuy**; una podría tener el valor anterior, **304qty1**, mientras que la otra podría tener el valor **301qty3**. Estas cookies podrían indicar que el usuario quiere comprar una copia del libro con el número 304 en el inventario y tres copias del libro con el número 301del inventario.

Además de un nombre y un valor, también se pueden proporcionar atributos opcionales como comentarios. Los navegadores actuales no siempre tratan correctamente a los atributos opcionales.

Un servidor puede proporcionar una o más cookies a un cliente. El software del cliente, como un navegador, se espera que pueda soportar veinte cookies por host de al menos 4 kb cada una.

Las cookies que un cliente almacena para un servidor sólo pueden ser devueltas a ese mismo servidor. Un servidor puede contener múltiples servlets; el ejemplo **Duke's Bookstore** está compuesto por varios servlets ejecutándose en un sólo servidor. Como las cookies son devueltas al servidor, los servlets que se ejecutan dentro de un servidor comparten las cookies.

Las cookies son de gran importancia en el proyecto ya que permiten identificar una conexión, conseguir que el buscador siga su tarea y de este modo que el usuario se pueda desconectar del sistema, ya que en su PC tendrá una cookie "permanente", que cuando se vuelva a conectar, indicará al sistema todo lo relacionado con su búsqueda.

Persistencia

Todas las cookies tienen un período de caducidad. Cuando este tiempo se cumple, la cookie desaparece. Teniendo en cuenta este comportamiento cabe distinguir cuatro tipos de cookies:

- Cookies de sesión: desaparecen cuando se acaba la sesión, entendiendo por sesión el período de tiempo que un usuario está en un sitio Web de manera continuada con la misma ventana del navegador.
- Cookies permanentes: si la fecha de caducidad de la cookie se corresponde con un futuro lejano, se puede decir que la cookie que es permanente. Esto no es del todo cierto, como se ha observado, ya que los navegadores cuentan con un espacio limitado para almacenar cookies de manera que las más nuevas hacen desaparecer a las más viejas cuando dicho espacio está totalmente ocupado.
- Cookies con fecha de caducidad: a veces los sitios Web establecen cookies con una fecha de caducidad concreta. Por ejemplo, se puede utilizar una cookie para recordar a un cliente que existe una oferta. Esta cookie tendrá necesariamente la fecha de caducidad de la propia oferta, más allá de la cual no tienen ningún sentido que siga existiendo.
- Cookies para borrar: borrar una cookie existente es como volver a escribirla con una fecha de caducidad anterior a la fecha actual.

La persistencia permanente es la que más se ajusta a nuestros objetivos ya que no sabemos cuando el usuario se volverá a conectar. Quedará definir una política de destrucción de Agentes ya que puede darse el caso de que el usuario no se vuelva a conectar.

5.3 – Sesiones

Las sesiones almacenan variables propias de cada usuario en el servidor, son parecidas a las cookies, pero:

- Se almacenan en el servidor, no en el cliente
- Pueden almacenar cualquier tipo de datos (no sólo cadenas)
- No tienen límite de tamaño (una cookie como máx. 4K)

Objeto implícito **session**: en él se pueden meter objetos Java asignándoles un nombre y recuperarlos con ese nombre

–session.setAttribute(*nombre*,*objeto*)
–session.getAttribute(*nombre*)

Los objetos guardados en la sesión se mantienen mientras que el cliente siga navegando por el sitio web.En realidad, las sesiones usan cookies, el servidor asigna automáticamente a cada usuario una cookie que hace de identificador único, este identificador sirve como “clave”para acceder a la información propia del usuario.

El objeto application funciona de manera parecida pero cuando se almacena un objeto y todos los clientes acceden al mismo.

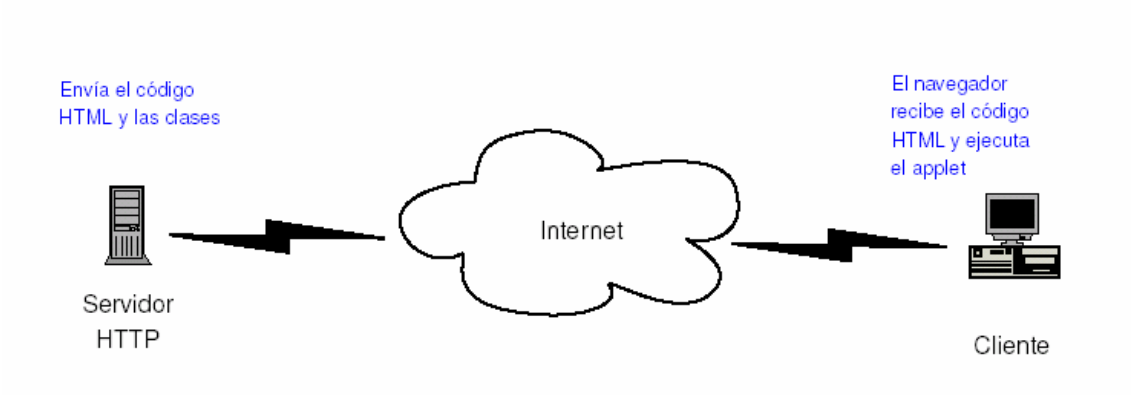
–application.getAttribute(nombre)
–application.setAttribute(nombre, objeto)

Lo almacenado en el objeto application se conserva en memoria mientras el servidor esté en marcha. Este objeto es de gran utilidad en nuestro proyecto, por que gracias a el y a la cookie podemos mantener la persistencia del proceso de búsqueda de los usuarios en el sistema. La cookie nos da el identificador del usuario, que buscamos en la lista que tenemos guardada dentro del objeto application, para obtener la información relacionada con la búsqueda.

5.4 – Applets

Un applet es una pequeña aplicación, formada por una o varias clases escritas en Java, que está insertada en una página Web.

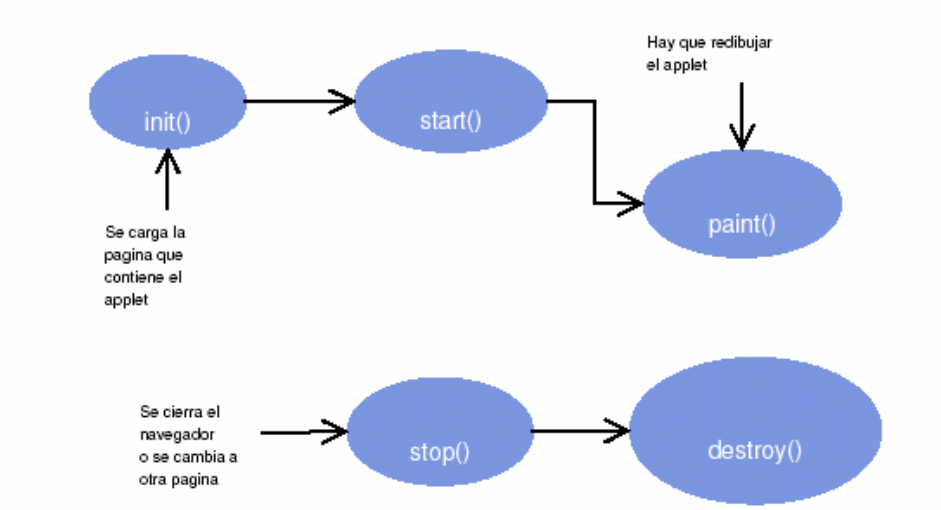
Cuando un usuario carga la página en la que está el applet, éste se ejecuta localmente (en la máquina cliente donde se está ejecutando el navegador Web) y no remotamente (sobre el sistema que está ejecutando el servidor HTTP)



Los applets deben ser subclases de la clase Applet (si se van a utilizar clases de AWT) o de Japplet (si se van a utilizar clases de Swing [Robert Eckstein]).

A diferencia de otras aplicaciones, la ejecución de un applet no comienza en el método main(). Hay una serie de métodos que son llamados cuando ocurren determinadas circunstancias. Estos métodos de Applet o de Japplet son los que hay que sobrescribir.

La siguiente figura muestra cuando llama el navegador a estos métodos:



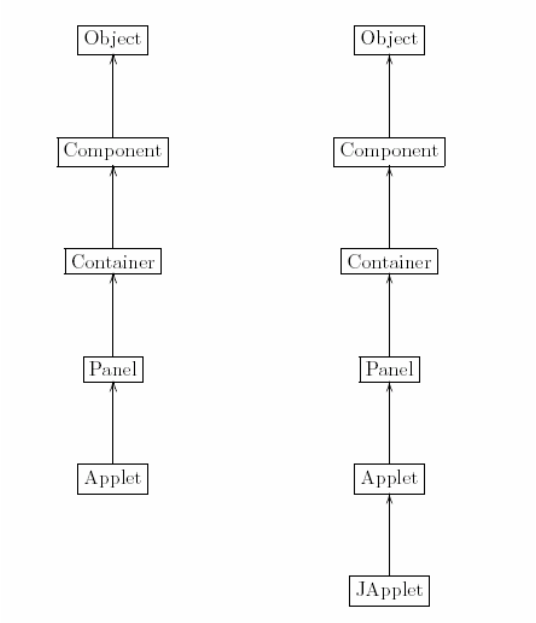
El esqueleto de un applet basado en la clase Applet se muestra a continuación:

```
import java.applet.Applet;
import java.awt.* ;

class unApplet extends Applet {
     // Declaración de atributos

      public void Init() {

    // Este método se llama al crearse el applet y sólo se ejecuta esa vez. Se suele utilizar para inicializar variables
    // globales y para construir la GUI.

      }
```

```
public void Start () {

// Se llama cuando hay que activar el applet. Esto sucede cuando el applet se ve en la pantalla. Se suele utilizar
// para comenzar las actividades de applet, ya que en general éste debe estar visible en la pantalla para poder
// realizar su función.

        }

 public void stop() {

// Es el inverso del anterior. Es llamado cuando el applet deja de estar visible. Se suele utilizar para interrumpir las
// actividades activadas en start().

 }

 public void destroy () {

 // Método llamado al destruirse el applet. Normalmente no se suele sobreescribir, ya que la destrucción del applet
 // conlleva la destrucción de todo lo que se ha creado en él. Puede ser útil para mostrar una ventana de
 // confirmación que pregunte al usuario si de verdad debe cargárselo.

 }

 public void paint (Graphics g) {

 // Método en el que se dibuja el applet y el cual deberemos sobreescribir para poder dibujar lo que nosotros
 // deseemos..

 }

 public void repaint () {

 // Este método no debería ser sobrecargado y es llamado cuando se necesita redibujar el applet. Podemos llamarlo
 // nosotros desde nuestro código. Llama a update(Graphics).

 }

 public void update (Graphics g) {

 // Llamado por repaint() cuando se necesita actualizar el applet. Su implementación borra el rectángulo y llama a
 // paint() para volver a pintarlo.

 }

 public void print (Graphics g) {

 // Equivalente a paint() pero es llamado cuando se va a imprimir el contenido del applet.

 }}
```

Un applet concreto puede sobrescribir todos estos métodos o solamente alguno de ellos en función de la tarea que deba realizar. En nuestro caso se ha modificado la función Init() que se encarga de representar directamente los resultados

La jerarquía de las clases Applet y Japplet es la siguiente:



Además de los métodos que controlan el ciclo de vida del applet hay otros métodos disponibles, entre los cuales destacan:

```
Public Image getImage (URL urlImages)
Public Image getImage (URL url, String fichero)
```

Estos métodos obtienen una imagen desde una URL y crean un objeto del tipo Image.

```
Public void showStatus (String mensaje)
```

Este método muestra la cadena pasada como argumento en la barra de estado (en la parte inferior del navegador).

```
Public void showDocument (URL htmlDoc)
Public void showDocument (URL htmlDoc, String frame)
```

Estos métodos se pueden utilizar para indicar al navegador que muestre una determinada página Web. En realidad están definidos en la clase AppletContext por lo que para utilizarlos hay que hacer lo siguiente: getAppletContext().showDocument(...).

Como ya se ha comentado antes, el applet debe estar incluido en una página Web. El lenguaje de marcado HTML contiene etiquetas para indicar que un elemento es un applet.

El esqueleto de código de HTML de una página que contenga un applet se muestra a continuación:

```
<HTML>
        <HEAD>
                <TITLE>
                        Applet
                </TITLE>
        <HEAD>
        <BODY>
                <APPLET
                        ARCHIVE=fichero.jar
                        CODE=ejemplo.class
                        CODEBASE=http://....
                        WIDTH=400
                        HEIGHT=400
                        ALIGN=MIDDLE>
                        <PARAM NAME="FICHERO" VALUE="c:/tmp/prueba.txt"/>
                </APPLET>
        </BODY>
</HTML>
```

ARCHIVE: Especifica el archivo jar donde se encuentran las clases y otros recursos que utilice el applet.

CODE: Especifica la clase que extiende a Applet o a Japplet. Se asume que la clase se puede encontrar en el mismo sitio que la página Web a no ser que se proporcione CODEBASE.

CODEBASE: Designa la URL donde encontrar la clase o el fichero jar.

WIDTH: Especifica la anchura que tendrá el applet.

HEIGHT: Especifica la altura que tendrá el applet.

ALIGN: Específica cómo se debe colocar el applet dentro del espacio disponible (LEFT, RIGHT, TOP, BOTTOM, MIDDLE).

PARAM: Se puede utilizar para pasar parámetros al applet. En NAME se Especifica el nombre del parámetro y en VALUE se especifica el valor.

Un applet se puede visualizar en un navegador Web (Internet Explorer, Mozilla,...) o utilizando la utilidad appletviewer que se distribuye en el JSDK.

Para visualizar un applet utilizando appletviewer hay que pasarle a esta utilidad el nombre de la página HTML que lo contiene, por ejemplo:

```
Appletviewer Ejemplo.html
```

Esta utilidad muestra solamente los applets descartando el resto del código que puede haber en la página. Si hay más de un applet en la página, cada uno de ellos se mostraría en una ventana diferente. También es posible visualizar con appletviewer applets que estén en páginas remotas.

Seguridad Applets

El hecho de que el código se ejecute localmente implica que la seguridad sea crucial.

Una forma en que la Plataforma Java proporciona protección contra ataques de un virus, por ejemplo, es a través del uso del **administrador de seguridad**. Actualmente los códigos del sistema del JDK llaman a los métodos del **administrador de seguridad** para realizar chequeos del control de accesos a recursos.

La mayoría de los navegadores instalan un controlador de seguridad, por eso los applets se ejecutan para el escrutinio de un **administrador de seguridad**. Ningún applet tiene permitido el acceso a recursos a menos que explícitamente se lo concedamos, mediante un permiso en la política de seguridad. En las plataformas Java que son compatibles con el JDK 1.2, los permisos deben ser concedidos mediante una entrada en un fichero de política.

El **administrador de seguridad** es una clase que controla si el código realiza operaciones que no están permitidas (en tal caso se lanza una excepción).

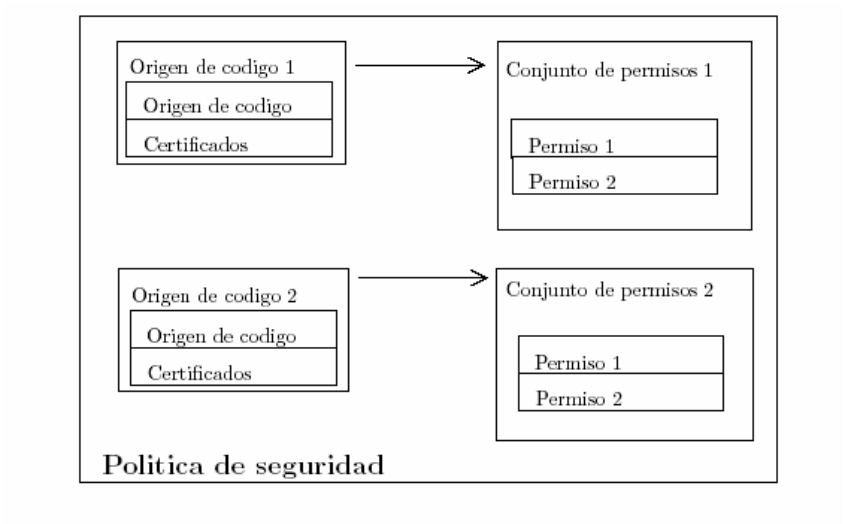
Entre las operaciones que comprueba están las siguientes:

- Si el hilo actual puede crear un nuevo cargador de clases.
- Si el hilo actual puede crear un subproceso.
- Si el hilo actual puede detener la máquina virtual.
- Si una clase puede acceder a un miembro de otra clase.
- Si el hilo actual puede acceder a un paquete específico.

- Si el hilo actual puede acceder o modificar las propiedades del sistema.
- Si el hilo actual puede leer desde o escribir en un determinado archivo.
- Si el hilo actual puede aceptar una conexión socket desde un host y número de puerto específicos.
- Si el hilo actual puede esperar una solicitud de conexión en un número de puerto local específico.
- Si el hilo actual puede llamar a los métodos Sto.(), suspend(), resume(), destroy(), setPriority()/setMaxPriority(), sentame(), o setDaemon() de un hilo dado o de un grupo de hilos.

- Si el hilo actual puede iniciar un trabajo de impresión.
- Si una clase puede acceder al portapapeles del sistema.
- Si una clase puede acceder a la cola de eventos de AWT.

Existe una política de seguridad que asigna un conjunto de permisos a orígenes de código.



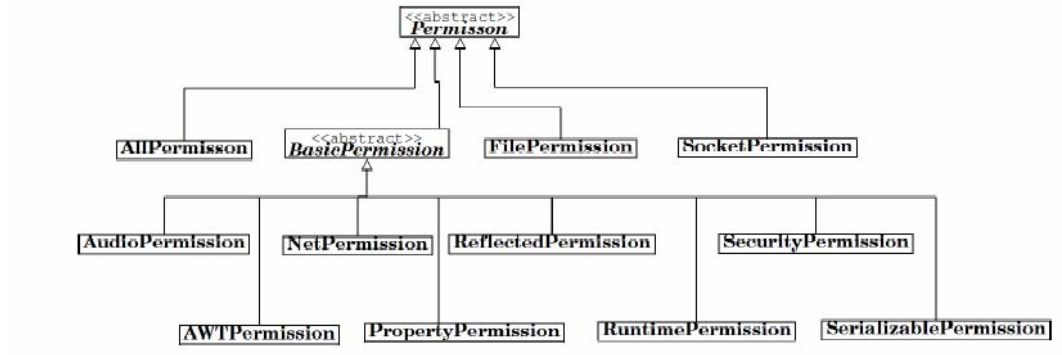
El origen de código consta de:

1. La ubicación del código que puede ser una URL o un fichero JAR, y
2. Certificados (que veremos más adelante).

El administrador de seguridad comprueba estos permisos en tiempo de ejecución.

Hay varias clases de permisos cada una de las cuales encapsula los detalles de un permiso particular.

La jerarquía de permisos se muestra en el siguiente diagrama:



Archivos de política de seguridad

Cuando se utiliza un administrador de seguridad, en el instante en el que se cargan las clases, se asignan permisos solicitando a un objeto del tipo Policy que asigne los permisos para el origen de código de cada clase.

La clase de política está establecida en el archivo java.security del subdirectorio jre/lib/security del directorio donde está instalado JRE. Por defecto dentro de este archivo está la siguiente línea:

Policy.provider=sun.security.provider.PolicyFile

Es decir, los permisos se deben especificar en un fichero. La siguiente línea muestra un ejemplo de permiso en un fichero de política de seguridad:

```
Permission java.io.FilePermission "/tmp/*", "read, write";
```

Se pueden crear archivos de políticas en localizaciones prefijadas. Por defecto existen dos localizaciones :

- El archivo java.policy en el directorio jre/lib/security.
- El archivo .java.policy en el directorio raíz del usuario.

Estas localizaciones están establecidas por defecto en el fichero java.security de la siguiente forma:

```
Policy.url.1=file:${java.home}/lib/security/java.policy
Policy.url.2=file:${user.home}/.java.policy
```

Durante la fase de desarrollo es mejor no modificar estos ficheros estándar java.policy o .java.policy (ya que los podríamos dejar en un estado peligroso). Es mejor especificar de forma explícita el archivo de políticas necesario para cada aplicación. Para ello hay que colocar los permisos en un determinado archivo (por ejemplo aplicación.policy) e iniciar la máquina virtual del siguiente modo:

```
Java –Djava.security.policy=aplicación.policy Aplicacion
```

Para applets cargados con appletviewer se utiliza (si el fichero de política se llama applet.policy).

```
Appletviewer –J-Djava.security.policy=applet.policy applet.html
```

El archivo de políticas especificado se añade a las políticas especificadas en los ficheros java.policy en el directorio jre/lib/security y .java.policy en el directorio raíz del usuario. Si se añade un segundo signo igual entonces solo se utiliza el archivo de política especificado, ignorando los otros.

```
Appletviewer –J-Djava.security.policy==applet.policy applet.html
```

Un archivo de política contiene una secuencia de entradas que comienzan con grant. Cada una de ellas tiene la forma:

```
grant codigoFuente {
        permiso_1;
        permiso_2;
                ...}
```

codigoFuente contiene un código base (que puede omitirse en el caso en que los permisos se apliquen a todas las fuentes) y los nombres de las entidades certificadoras seguras (que pueden omitirse si no son necesarias las firmas para esta entrada).

A continuación se muestran varios ejemplos de código base:

```
Grant codeBase « http://ejemplo.es/ej/ » {…}
Grant codeBase « http://ejemplo.es/ej/fichero.jar » {…}
Grant codeBase « file :c:\\ejemplo\\ » {…}
```

En la primera línea se conceden los permisos al código que esté en el directorio /ej/ en la máquina http://ejemplo.es

En la segunda línea se conceden los permisos al código que se encuentre en el fichero /ej/fichero.jar en la máquina http://ejemplo.es pero no a ningún otro fichero en el directorio /ej/

En la tercera línea se conceden los permisos al código que se encuentre en el directorio c:/ejemplo/ en la máquina local.

En cuanto a los permisos, estos tienen la siguiente estructura:

```
Permission nombreClass nombreDestino, listaAcciones
```

- **nombreClase** es el nombre completo (incluyendo la información del paquete al que pertenece) de una clase de permisos (por ejemplo java.net.SocketPermission).
- **nombreDestino** es una especificación del permiso, en el ejemplo anterior sería especificar el servidor y quizá un número de puerto.
- **listaAcciones** es también una especificación de permiso, se trata de una lista de acciones como read o connect separadas por comas.

Algunas clases de permisos no necesitan que se especifique nombreDestino o listaAcciones.

La siguiente tabla muestra algunas clases de permisos:

java.io.FilePermission	Para asignar permisos de acceso, creación, borrado o ejecución a ficheros.
java.net.SocketPermission	Para asignar permisos para aceptar conexiones, conectarse, escuchar o resolve de sockets.
java.util.PropertyPermission	Para establecer que propiedades se pueden leer o escribir.
java.lang.RuntimePermission	Para establecer si se puede crear un cargador de clases, si se puede obtener el que se está utilizando, si se puede finalizar la máquina virtual, si se puede imprimir, si se puede parar un hilo,...
java.awt.AWTPermission	Para establecer si se puede acceder al portapapeles, si se puede acceder a la cola de eventos,...
java.net.NetPermission	
java.security.SecurityPermission	Para establecer si se puede tener acceso a objetos relativos con la seguridad como por ejemplo Security, Policy, Provider ...
java.security.AllPermision	Otorga todos los permisos, por lo tanto debe ser utilizada sólo durante pruebas.

El JSDK proporciona una herramienta para crear o editar ficheros de políticas de seguridad llamada policytool.

Certificados

La certificación digital consiste en que una autoridad de certificación (CA) firma un mensaje especial m que contiene la identificación de un usuario y su clave pública de forma que cualquiera puede verificar que un mensaje fue firmado por una autoridad de certificación y pueda por tanto confiar en la clave pública del usuario.

Veamos un ejemplo:

- Alicia envía una petición de certificado que contiene su identificación y su clave pública a una CA.
- La CA forma un mensaje especial m a partir de la petición y lo firma con su clave privada obteniendo una firma X. La CA envía el mensaje m y la firma X a Alicia. Las dos partes forman un certificado.
- Alicia envía a Roberto el certificado para que éste confíe en su clave pública.
- Roberto verifica la firma & utilizando la clave pública de la CA. Si todo es correcto, acepta la clave pública de Alicia.

El certificado más extendido es el del formato X.509. Este estándar es parte de las series X.500 del CCITT.

En el formato más sencillo, un certificado X.509 contiene los siguientes datos:

- Versión del formato de certificado.
- Número de serie del certificado
- Identificador del algoritmo de firma
- Nombre del firmante del certificado.
- Periodo de validez (inicio y final)
- Nombre de loa identidad que se certifica
- Clave pública de la identidad que se certifica(Algoritmo, parámetros del algoritmo y clave pública).
- Firma (de todo lo anterior codificados con la clave privada del firmante)

Sun proporciona una herramienta para la generación de claves y almacenes: **keytool**:
Algunas de las opciones que admite son:

-genkey	genera un par de claves y las guarda en un almacén
-list	lista el contenido de un almacén
-export	para exportar la clave pública a un archivo de certificado. El certificado resultante está autofirmado.
-printcert	Para visualizar la información de un certificado
-import	Si se confía en un certificado se puede añadir al almacén
-certreq	Genera una petición de firma de certificado (*Certificate Signing Request*) en formato PKCS#10.
-delete	Elimina una entrada del almacén.

Una vez que se dispone de las claves, se puede firmar el código utilizando jarsigner.

```
Jar –cvf applet.jar *.class
```

El fichero JAR resultante se puede firmar del siguiente modo:

```
Jarsigner –keystore LP.almacen –signedjar sApplet.jar applet.jar LP
```

Supongamos ahora que este applet lo desea ejecutar alguien en su máquina y que ha obtenido el certificado y que confía en él.

Por defecto, los applets firmados que se ejecutan desde el plug-in del navegador, una vez que se acepta que se ejecuten, tienen todos los permisos, por lo tanto no habría que aplicar ningún fichero de política de seguridad. Pero este comportamiento se puede modificar si en una de las localizaciones donde java busca los ficheros de políticas se han añadido las siguientes líneas:

```
grant {
        permission java.lang.RuntimePermission “usePolicy”;
}
```

donde se indica al plugin-in que para los applets firmados utilice también los ficheros de políticas de seguridad.

Una vez añadidas estas líneas ya no aparece el mensaje inicial indicando si deseamos ejecutar el applet, sino que intentará ejecutarlo y si no tiene los permisos asociados fallará en la ejecución y si tiene los permisos necesarios lo ejecutará.

En este caso vamos a asumir que se decide modificar (o crear) .java.policy del directorio $user.home.

Este fichero con la modificación comentada anteriormente y con las líneas para asignar permisos al código firmado por LP2 sería.

```
grant {
        permission java.lang.RuntimePermission “usePolicy”;
};
keystore “certificados.almacen”;
grant signedBy “LP2” {
    permission java.io.FilePermission “user.home“, “read”;
};
```

Además colocaremos el fichero certificados.almacen en este mismo directorio.

Ahora ya se puede cargar el applet y se ejecutará correctamente.

6- DISEÑO DEL SISTEMA

Se trata de una aplicación Web, basada en un sistema multi-Agente, capaz de guiar, a un usuario, en la construcción de una Ontología sobre un dominio concreto, a partir de la Web. Concretamente es una aplicación Web que interactúa con el usuario para pedirle los datos iniciales, concepto a partir del cual se crea la Ontología, profundidad de la búsqueda (numero máximo de subclases de una clase) y tamaño (numero máximo de paginas Web a evaluar) estos parámetros son pasados al sistema multi-Agente para que gestione el trabajo de construcción y obtenga resultados del constructor de taxonomías y luego el SMA se los devuelva a la aplicación para que se lo muestre al usuario. Estos resultados están formados por la estructura jerárquica de conceptos (conceptos, instancias y relaciones no taxonómicas), los recursos Web asociados a estas clases y las frases relevantes de cada concepto, seleccionando esta frases con conceptos nuevos el usuario puede ir aumentando la Ontología, haciendo uso de nuevo del SMA que mediante el constructor de taxonomías recibirá una nueva Ontología (una para cada frase seleccionada) que el SMA irá interlazando.
Otra cosa que permite el sistema es guardar los resultados para futuras consultas, así pues, se puede recuperar una Ontología guardada para revisar sus estructura de clases o para acceder a las paginas Web, pero no podrá seguir construyendo esa Ontología. El sistema también permite, debido a que el proceso puede llegar a ser largo, que el usuario comience una búsqueda y se desconecte del sistema, y éste es capaz de continuarla para que cuando el usuario vuelva a conectarse pueda interactuar con los resultados.
Así pues, el sistema esta formado por tres módulos:

- Constructor de taxonomías (taxoBuilder), librería que busca en Internet el concepto deseado y analizando un conjunto de páginas Web es capaz de construir una taxonomía. Este proceso es el mas pesado para el sistema, requiere de una gran cantidad de recursos y exige que el sistema este diseñado en base a esta limitación, por esto la decisión de implementar un SMA.

- SMA, Sistema multi-Agente, encargada del procesado del trabajo a realizar, es decir soporta la carga de gran parte del sistema. Por ejemplo ejecuta el constructor de taxonomías, construye la Ontología mediante los resultados del constructor de taxonomías interlazando resultados del constructor, interpreta ontologías, etc. Es decir, bajo el sistema multi-Agente se ejecutan todos los procesos y tareas del sistema, exceptuando los propios procesos de la interficie Web y la representación grafica. Utilizar un SMA permite que la aplicación pueda aceptar un número mucho mayor de clientes simultáneamente y aumentar su escalabilidad, ya que estos sistemas permiten ejecutar agentes en diferentes máquinas.

- Aplicación Web que realiza tareas de representación de ontologías, además de interface entre el sistema y el usuario controlando el sistema según los parámetros introducidos. Es decir, es la fachada del sistema, además según las características de las aplicaciones Web, permite el acceso concurrente al sistema de múltiple usuarios y que éste sea remoto.

A continuación explicaremos cada uno de estos módulos, así como las partes que los forman, los recursos que utilizan, los problemas que han causado y sus soluciones, etc.

6.1. Constructor de taxonomías.

Con el fin de comprender mejor que es y para que se usa taxoBuilder en este proyecto, hace falta conocer antes que es una taxonomía (concepto descrito en el capitulo 4 ontologías) y como se usa en el proyecto. La *taxonomía* es la ciencia de la clasificación, estructurando un dominio por *taxones* que describen jerárquicamente las relaciones, de similitud, parentesco u otro criterio, entre las entidades de ese dominio. Por lo tanto, en este proyecto, el constructor de taxonomías creara una estructura jerárquica de los conceptos obtenidos de todas las webs analizadas, es decir clasifica la información.
En el proceso de análisis de las webs encontradas, taxoBuilder además de conceptos clasificables, también encuentra candidatos a relaciones no taxonómicas, que son difícilmente clasificables. Además clasifica *instancias* [Sánchez, 2005] de conceptos, clase-entidad. Es decir, la concretización de un concepto en una entidad propia (entidades, personas, compañías, eventos, etc), como por ejemplo "Asociación Nacional De Lucha Contra El cáncer" es una instancia de "cáncer".

El constructor de taxonomía, taxoBuilder [Sánchez, 2005] es la librería, escrita en Java 1.5, capaz de generar una taxonomía con la información contenida en la Web, a partir de los parámetros de búsqueda. Esta librería ha sido desarrollada por David Sánchez Ruenes, profesor de la Universidad Rovira y Virgili, miembro de Grusma y director de este proyecto.

Esta herramienta, es sin duda, una de las partes más importantes del proyecto, ya que le ofrece una taxonomía, del dominio del conocimiento deseado por el usuario a partir de la Web, y haciendo sucesivas ejecuciones de ella, según desee el usuario, el sistema construirá una Ontología que abarque los resultados de dichas ejecuciones. Pero de igual manera, este proyecto hace de taxoBuilder una herramienta mas funcional ya que por si sola no es aplicable para su uso directo y es este proyecto quien le permite interactuar con el usuario, representar sus resultados de forma que cualquier persona pueda comprender, utilizar sus resultados para crear ontologías mas amplias, etc. Por lo que podemos decir que este proyecto es una de las muchas aplicaciones que podría tener taxoBuilder.

El constructor de taxonomías utiliza la API de Google con tal de poder encontrar las páginas Web que debe analizar. De este modo Google le proporciona un motor de búsqueda potente, sirviendo a taxoBuilder un rico material para analizar.
Además de las páginas Web, que contienen la información a clasificar, también le ofrece información relativa a la búsqueda, como por ejemplo, tiempo transcurrido, numero de páginas Web encontradas, etc. Toda esta información es recopilada por taxoBuilder para poder construir la taxonomía. Una vez que ya dispone de información para clasificar y procede a analizarla extrayendo relaciones entre conceptos, instancias, relaciones no taxonómicas y las webs asociadas a partir de texto natural. Hay que tener en cuenta que el contenido de las webs no esta estructurado computacionalmente, es solo texto que únicamente los humanos pueden entender.

El algoritmo de construcción de taxonomías esta basado en el análisis de una considerable cantidad de sitios Web para encontrar candidatos de conceptos para un dominio definido por la palabra (o combinación que el usuario desea evaluar) inicial (*keyword)*. La detección se realiza usando el modelo que envuelve a esa *keyword*. En

este punto, se puede centrar el análisis para el estudio del *vecindario* de la palabra inicial, es decir, las palabras que la acompañan, cuando ésta se presenta en las *oraciones sustantivas* (frases que se pueden sustituir por un sintagma nominal o un sustantivo). En Ingles la palabra inmediatamente anterior a la *keyword* es clasificada frecuentemente como candidata, mientras que la inmediatamente posterior representa el dominio donde esta aplicado (Grefenstette, 1997).

El sistema realiza peticiones al motor de búsqueda (Google) para obtener las paginas webs donde extraer el conocimiento, estas las realiza dinámicamente y así cada vez ara la petición mas adecuada obteniendo el recurso Web mas apropiado en cada momento. Por lo tanto, se extrae el conocimiento mas exacto para un dominio y la evaluación mas especifica para el cuerpo de un documento Web. Además, mediante una consulta especialmente realizada, se puede obtener del motor de búsqueda información relativa a la ésta y así crear una escala estadística según el número de veces que aparece la *keyword.* De esta estadística se chequea la relevancia de los términos extraídos y se evalúa la fuerza de la relación entre los conceptos.

Con más detalles, el algoritmo para descubrir conceptos representativos y construir una taxonomía de los recursos Web tienen las siguientes fases.

- El proceso empieza con la *keyword* inicial representante de un dominio (p.e cáncer), entonces el algoritmo usa el motor de búsqueda (Google) para obtener la más amplia representación de webs que contengan esa *keyword* (p.e. 100 páginas para el dominio de cáncer). En principio, cuantas más páginas Web evalúe más completos serán los resultados. Pero, al evaluar muchas páginas la mayoría de los términos detectados ya se habían detectado anteriormente así pues con un reducido número de páginas (p.e. 200) se puede obtener buenos resultados.

- Para cada sitio Web obtenido se realiza un análisis exhaustivo del contenido. Gracias a un analizador sintáctico diseñado para la lengua inglesa (OpenNLP) se pueden detectar categorías gramaticales. La palabra anterior al *keyword* es seleccionada como concepto candidato (p.e. prostate cáncer) y la posterior también es seleccionada para su posterior evaluación (p.e. cáncer tratament).

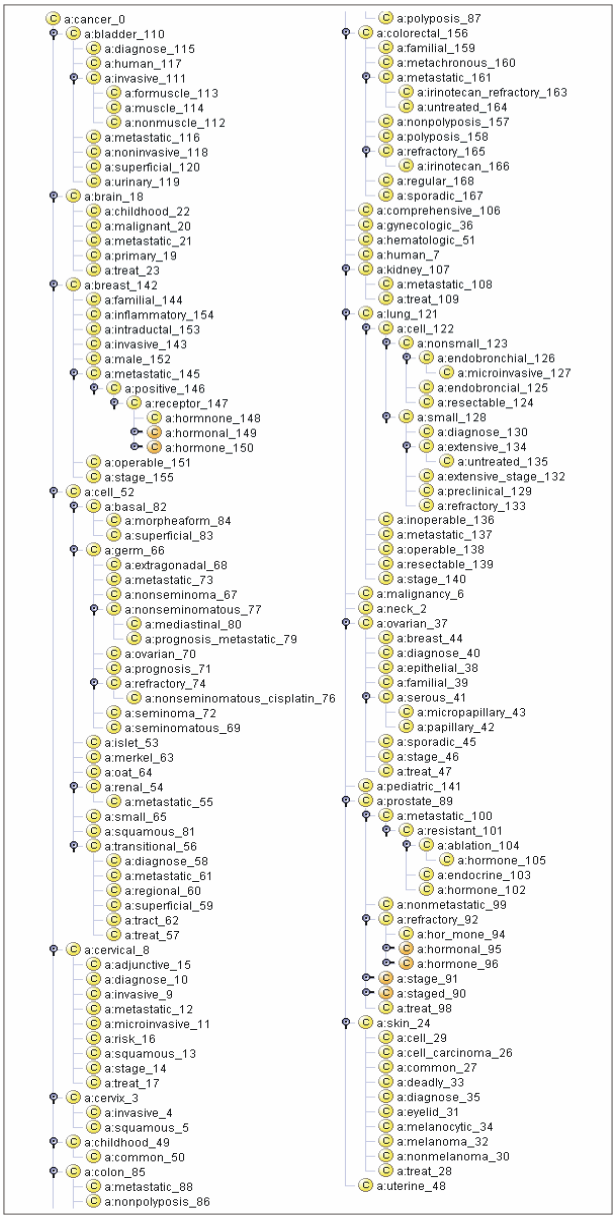
- Para cada clase candidata un algoritmo decide si será considerada como una candidata a subclase de una inicial (p.e. prostate cáncer es tipo de cáncer), o como una candidata a instancia de una clase (p.e. lucemia es un ejemplo de cáncer). Estas frases serán las que se muestren al usuario para continuar su construcción evaluando el nuevo término. Por ejemplo, (en castellano pero el sistema es en inglés): “cáncer recibe radioterapia” posible frase que se le mostraría al usuario para seguir su búsqueda explorando radioterapia, “cáncer de colon empieza como un pólipo” el usuario exploraría pólipo.

- Gracias a un estudio estadístico se pueden seleccionar las candidatas más relevantes. Según el número de veces que aparece esa *keyword*, junto con el número de páginas encontradas con esa *keyword*, y el tiempo que se ha tardado, (más otros parámetros estadísticos).

- Ahora para cada candidato se le calcula su relevancia dentro de la taxonomía, es decir, la importancia que tiene ese termino dentro de la estructura. Para ello se calcula con métodos estadísticos usando de nuevo el tiempo de la búsqueda, el número de apariciones de la *keyword*, y el número de webs analizadas.

- Para cada concepto, una nueva y mas compleja *keyword* se construye uniendo éste con el *keyword* inicial (p.e. “breast cáncer”), y el algoritmo se ejecuta de nuevo. Este proceso se ejecuta recursivamente hasta alcanzar la profundidad deseada.

- El resultado es una estructura almacenada como una Ontología. Cada clase e instancia almacena las URLs (paginas Web) de donde fueron seleccionadas. La siguiente figura muestra la estructura.



Además como ya hemos mencionado anteriormente, esta librería también reconoce instancias, resulta bastante complicado decidir si una clase es una instancia de otra o bien es una clase-entidad, por ejemplo esta claro que "lucemia" es una instancia de la clase cáncer ya que es un ejemplo concreto de cáncer y que "Asociación De Lucha Contra El cáncer" es una instancia pero en este caso es una clase-entidad. Pues una forma de determinar esto es realizar cálculos estadísticos según el número de mayúsculas y minúsculas dentro de la *keyword.*

Esta librería devuelve al sistema dos ficheros con la taxonomía deseada expresada como una Ontología en lenguaje OWL. Uno de los ficheros contiene la taxonomía completa (fichero full) junto con todos sus atributos, es decir, elementos que aportan información de la clase. Los atributos que se utilizan en este proyecto son:

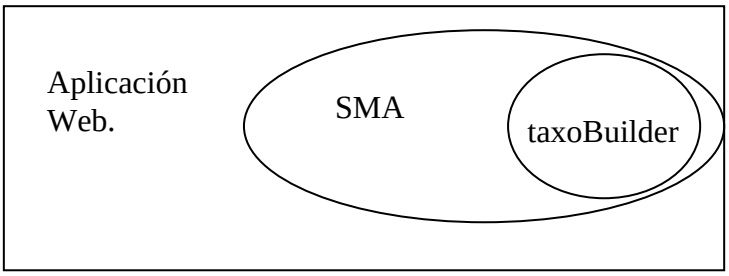
- *Confidence:* indica la relevancia de una clase dentro de la Ontología.
- *URL_totales:* contiene todos lo recursos Web asociados a la clase.
- *frase_clean_suj:* la clase a la que esta relacionada es el sujeto.
- *frase_clean_pred:* la clase a la que esta relacionada es el predicado.

El otro fichero también contiene la taxonomía pero sin atributos (fichero clean). Es decir de este fichero solo podemos obtener estructura jerárquica y en el caso de las instancias no aparece su nombre.
Esta separación es debido a que como OWL no esta pensado para almacenar datos, no podríamos visualizar el fichero completo en programas como el Protegeé. De esta forma damos la opción al usuario a visualizar su Ontología en estos programas mediante el fichero clean. En el caso de este proyecto se muestran ambos ficheros según decida el usuario.

Por último subrayar que taxoBuilder utiliza bastantes recursos y es necesario establecer una estrategia de ejecución con tal de que el sistema sea estable. Hace un uso intensivo de la CPU en tareas de análisis de textos (parse, OpenNLP, etc) y cálculos estadísticos, pero sobretodo requiere mucha memoria RAM, unos 200 Mb en una búsqueda de 200 paginas webs a evaluar y una profundidad de 4 (numero máximo de subclases de una clase). Así que intentaremos ejecutar taxoBuilder en un entorno de ejecución distribuido.

6.2- Descripción del sistema.

Como ya hemos introducido anteriormente, el sistema se compone de tres módulos. El constructor de taxonomías, el sistema multi-Agente y la aplicación Web, cada uno de estos módulos encapsula al anterior de la siguiente forma.



- Sistema multi-Agente:

Como explicamos en el apartado 5.1.5, es necesario un diseño del sistema contemplando la carga de taxoBuilder, puesto que se van a utilizar varias instancias de éste para construir la solución final (entre otras tareas bastante complejas), por este motivo esta justificado la decisión de implementar un sistema multi-Agente que cargue con el peso del sistema. Este sistema de agentes proporciona un entorno de trabajo idóneo para satisfacer los objetivos iniciales. Permite separar las fases o trabajos de la aplicación en Agentes y de este modo conseguimos lo siguiente:

- 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 mostrar la información y otro de hacer la búsqueda.

- 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.

- 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 se crean un InternetAgent (agentes buscadores que ejecutan taxoBuilders) para cada búsqueda. Teniendo en cuenta que es una aplicación Web, el número de búsquedas variará según el número de usuarios. También facilita la integración de nuevas funciones, ya sea añadiendo nuevos Agentes, en este caso habrá que hacer algunas modificaciones, o bien añadir nuevos servicios a los Agentes ya existentes, cosa transparente para el resto del sistema.

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

- Aplicación Web.

La decisión de crear una aplicación Web en lugar de una aplicación local, viene dada por las características y objetivos del sistema. De este modo, los usuarios se podrán beneficiar de las ventajas de estos sistemas distribuidos:

- Actualización: todos los usuarios siempre dispondrán de la última versión. Las posibles mejoras, las correcciones de errores y la incorporación de nuevas funcionalidades estarán accesibles para todos transparentemente.

- Accesibilidad: toda aquella persona que desee hacer uso del sistema solo necesita de un PC con conexión a Internet, es decir, se puede gozar de los servicios del sistema independientemente de su localización geográfica.

- Independencia del hardware: puesto que se trata de una aplicación que se ejecuta en un servidor, cualquier maquina con conexión a Internet y con un navegador Web puede utilizar el sistema. A pesar de esto, como veremos mas adelante, parte del trabajo se ejecuta en la maquina del usuario mediante un Applet. Teóricamente un Applet se puede ejecutar en cualquier máquina siempre que en ésta, este instalada la *Maquina Virtual de Java*, pero el trabajo que debe realizar este Applet también es bastante costoso, sobretodo realiza numerosas entradas y salidas (acceso a ficheros), por ello, evidentemente el tiempo de ejecución aumentara según la máquina del usuario.

- Independencia del software y la plataforma: de igual modo que la independencia del Hardware, un usuario solo necesita tener instalado un explorador de Internet (p.e. Mozilla, Explorer, etc) y la *Maquina Virtual de Java*, sin importar que sistema operativo este utilizando (Linux, Windows, etc) así como tampoco necesita instalar el propio sistema en su maquina.

- Autónomo: aunque se trate de una aplicación interactiva, donde es el propio usuario quien guía la construcción de la Ontología, parte del trabajo, la construcción propia de la Ontología, se puede hacer sin la necesidad de que el usuario este conectado al servicio. Es decir, alguien desea hacer una búsqueda para obtener la consiguiente Ontología, como esto puede ser un poco lento (según los parámetros iniciales), el usuario se puede desconectar del sistema y recoger los resultados en otro momento. Esto solo se puede conseguir si la aplicación es distribuida.

La aplicación Web es la encargada de encapsular el sistema haciendo de interface entre el propio sistema y los usuarios. En primer lugar, es una puerta de acceso al sistema, a través de la Web de la aplicación, los usuarios podrán acceder al mismo. Una vez hayan captado los clientes, se encarga de arrancar el sistema multi-Agente y proporcionarle los parámetros del usuario a lo largo del proceso. Por último, arranca un applet para que muestre los resultados en el cliente, evidentemente también le proporciona a éste toda la información que necesita.

6.3- Descripción detallada de la entrada y salida del sistema

Mediante una página Web, 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, es decir número máximo de subclases que tiene una clase.
- El número máximo de páginas Web a considerar (analizar).

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 que tienen asignado un valor por defecto.

Basándose en estos parámetros, se construirá, mediante la librería taxoBuilder, la taxonomía expresada como una Ontología en lenguaje OWL y se almacenará en archivos (full y clean) para que pueda ser visualizada por alguna herramienta de visualización de Ontologías (únicamente el fichero clean), por ejemplo Protégé, que es la que hemos utilizado para visualizar los resultados que obteníamos. La aplicación Web accederá a estos ficheros para mostrárselo al usuario y esperar nuevas peticiones.

6.4- Diseño del sistema Multi-Agente

Debido a que el sistema Multi-Agente se arranca desde la aplicación Web, antes hay que crear un MAIN-Container, para que los agentes que se creen tengan una plataforma donde introducirse. Para esto solo es necesario ejecutar el JADE (jade.Boot) sin agentes y sin interface gráfica (opcional) desde un fichero batch. Ahora Jade permanecerá a la escucha, en LOCALHOST: 1069, a la espera de nuevos contenedores con sus agentes.

6.4.1- Arquitectura del SMA

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

- Agente usuario
- Agente coordinador
- Agente internet

Las funciones del agente usuario serán las siguientes:

- Interactuar con el usuario mediante una página Web para obtener los parámetros de la búsqueda.
- Comunicarse con el agente coordinador para informarle de los parámetros de búsqueda, y luego obtener los resultados.
- Mostrar toda la información obtenida en la página Web.

Las funciones del agente coordinador serán las siguientes:

- Comunicarse con el agente usuario para obtener los datos de la búsqueda, y luego devolverle el resultado de la misma.
- Crear dinámicamente uno o varios agentes internet, y cuando estos finalicen su trabajo destruirlos.

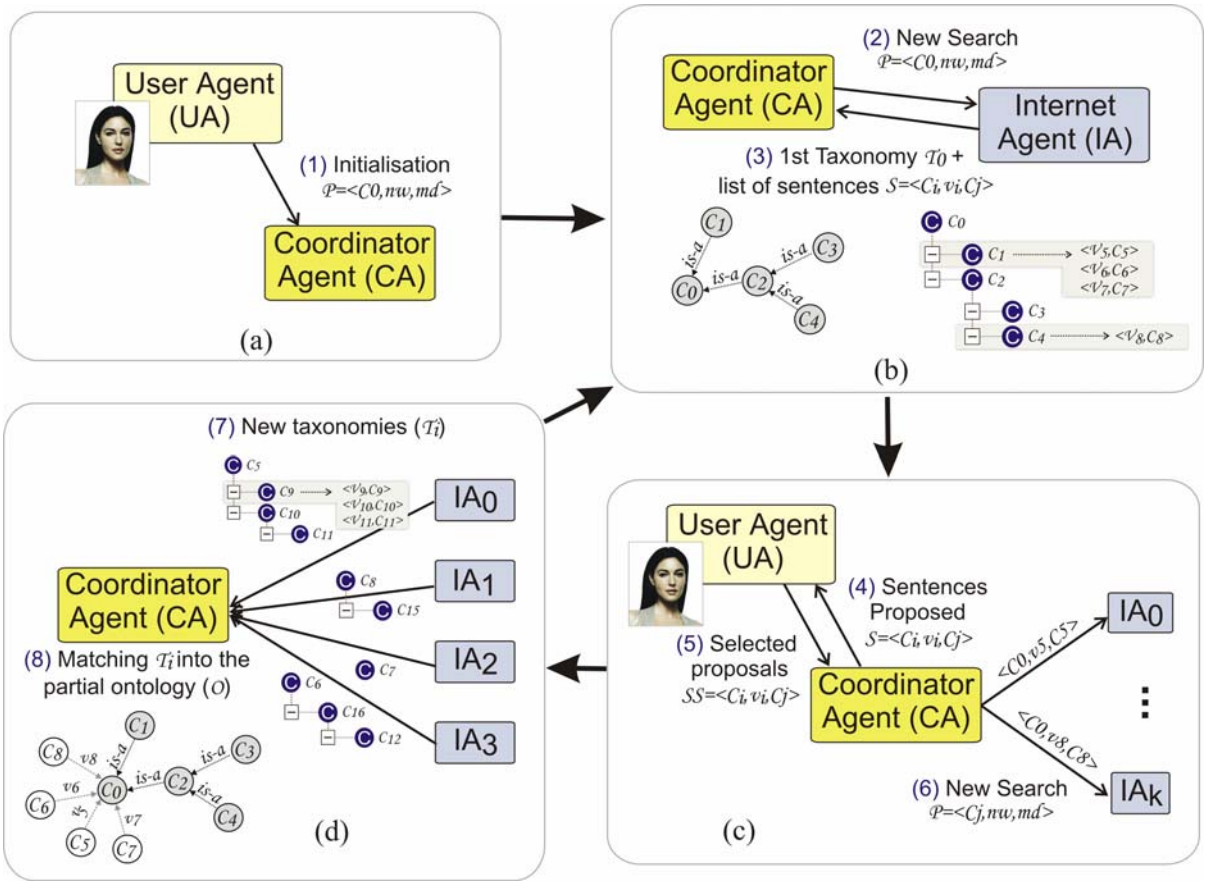
- Comunicarse con el agente internet para informarle de los parámetros de la búsqueda, y después para obtener el resultado.
- Componer la Ontología a base de resultados parciales de taxonomías y las decisiones del usuario.

Y las funciones del agente internet serán las siguientes:

- Este agente interactúa con una librería llamada taxoBuilder que generar la ontología.
- Comunicarse con el agente coordinador para darle los resultados.

El agente usuario y el agente coordinador están asociados a una búsqueda que inicia el usuario. Estos se crean directamente desde la aplicación Web. Para ello es necesario, en primer lugar, arrancar el Jade para que éste cree un contenedor principal (MainContainer). Ahora el sistema ya esta preparado para recibir peticiones de búsqueda (solo una por usuario), para cada petición de búsqueda se crea un contenedor, dentro del MainContainer, con el userAgent y el coordinatorAgent y cuando el usuario finalice esta búsqueda será destruido. El número de agentes internet varia según el numero de búsquedas y después de está se destruirá. El agente coordinador será el encargado de crear o destruir agentes internet dinámicamente.

El gráfico [Sánchez, Isern, Moreno, 2004] general de la arquitectura del sistema multi-agente y el proceso paso a paso sería el siguiente:



6.4.2- Funcionamiento del SMA

Como ya hemos descrito anteriormente, este SMA esta formado de tres agentes, suficiente para alcanzar los objetivos.

El SMA entra en funcionamiento en cuanto la aplicación Web reciba los parámetros iniciales de la búsqueda. Entonces una vez creados el UserAgent y el CoordinatorAgent, la aplicación Web le envía estos parámetros al UserAgent. Este agente crea un mensaje de tipo Request y se lo envía al Coordinador, el coordinador analiza este mensaje y evalúa cuantos Internets Agents necesita. Si se trata de la primera búsqueda solo creara un agente Internet, pero si el usuario ya ha recibido un resultado parcial y lo quiere continuar, seleccionando las frases propuestas por el sistema, entonces creara tantos InternetAgents como búsquedas (frases). Como ya hemos mencionado, las clases tienen asociadas frases que aportan nuevos conceptos a la Ontología, como por ejemplo, la clase inicial *cáncer* puede tener relacionada “fumar provoca cáncer”, “quimioterapia es un tratamiento del cáncer” y “cáncer de colon comienza como un pólipo”. Si el usuario decide buscar estas tres frases el sistema creara tres agentes Internet que buscaran simultáneamente, uno buscará “fumar”, otro “quimioterapia” y el otro “cáncer de pulmón” y se esperará hasta que reciba las tres taxonomías.

Cada agente Internet interactúa con taxoBuilder, librería encargada de la construcción de taxonomías. Cuando taxoBuilder finaliza su tarea, le devuelve a su agente Internet la localización física (path) de los ficheros owl creados, esto es posible gracias a que InternetAgent y taxobuilder siempre están en la misma máquina. Ahora este agente guarda en un buffer el contenido de estos ficheros, ya que los ficheros tienen un tamaño medio de 3Mb, comprimimos el buffer, gracias a librerías de compresión de datos (java.util.zip.Deflater), para evitar sobrecargar la línea entre el host del Internet y el del Coordinador, y disminuir el tiempo de transmisión. Finaliza su trabajo enviando el resultado al Coordinador.

En este punto del proceso, el Coordinador destruye al agente Internet que ya le haya dado el resultado. Es importante esta destrucción para liberar recursos ya que el alto rendimiento y la concurrencia es uno de los motivos de implementar un sistema Multi-Agente. Según vaya recibiendo resultados, los descomprime (java.util.zip.Inflater), y en el caso de que haya mas de uno, por que si es la primera búsqueda solo habrá un resultado, va creando la nueva ontología en un fichero owl, esta ontología es la unión de las taxonomías recibidas, descrita con detalle en los apartados relacionados con el tratamiento y construcción de ontologías, para mostrar en la Web un único resultado para cada dominio. También guarda todas las búsquedas realizadas, así evitamos repetirlas, disminuyendo notablemente la carga del sistema. A este agente solo le queda enviar esta nueva ontología al agente Usuario, de nuevo Coordinador guarda en un buffer el contenido del fichero, lo comprime y se lo envía al agente Usuario.

Agente Usuario recibe el contenido de la ontología a representar la descomprime y lo guarda en un fichero con el nombre de la búsqueda para pasarle su path a la aplicación., cerrando de este modo el ciclo del proceso. Este agente una vez más hace de interfaz entre la aplicación Web y el sistema Multi-agente, proporcionándole un fichero que

representar, a partir de este punto se termina el trabajo del sistema Multi-agente y prosigue en la aplicación Web.

Tanto el agente Usuario como el agente Coordinador se quedan esperando nuevas búsquedas, por si el usuario decide seleccionar alguna frase propuesta y no se destruirán hasta que la el usuario desee terminar con la construcción de la Ontología.

6.4.3- Comunicación entre agentes

Un SMA se caracteriza por tener una gran capacidad de comunicación, hecho que les hace ser muy útiles para la resolución de problemas complejos.

- Ontología del SMA

La ontología del SMA es la que se ha utilizado en la comunicación entre los agentes, es decir, describe al sistema la información que ha de manejar. A continuación detallaremos los diferentes conceptos, predicados y acciones que contiene la ontología.

- Conceptos de la Ontología

- MsgSerial: estructura de datos que contiene el contenido comprimido del fichero owl, el tamaño de este fichero, útil para descomprimir el contenido, y el nombre del mismo.

Nombre propiedad	**Tipo**
Msg	Byte[]
longi	int
nomFich	String

- Search: estructura de datos que contiene la información introducida por el usuario. Las palabras de búsqueda (p.e cáncer o cáncer de pulmón, etc), el dominio de la búsqueda, número de palabras que forman la búsqueda (p.e cáncer=1, cáncer de pulmón=3). También incluye un booleano que indica al coordinador si de una petición concreta se necesita realizar la búsqueda o solo tiene que incluir una frase, proporcionada por el sistema como dato relevante, en la ontología.

Nombre propiedad	**Tipo**
words	String
relationSearch	String
numWords	Int
buscar	booleano

- Request: estructura de datos que contiene toda la información introducida por el usuario, lista de objetos Search, una para cada agente Internet, profundidad de la búsqueda y número de páginas.

Nombre propiedad	Tipo
searchList	ArrayList de Search
Depth	int
Size	int

- Inform: estructura de datos que contiene una lista de elementos del tipo MsgSerial que es el resultado que un agente internet devuelve al Coordinador cuando ha realizado su trabajo y este al Agente Usuario.

Nombre propiedad	Tipo
cont	ArrayList de msgSerial

- Predicados de la Ontología

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

- Acciones de la Ontología.

SearchWord: acción que debe realizar todo agente Coordinador y Internet cuando recibe un mensaje del tipo *Request* del agente Usuario o Coordinador (en el caso del Internet), y que consiste en buscar una palabra con *Taxo*, en el caso del Internet y analizar los ficheros Owl obtenidos y devolver al agente Usuario la información necesaria mediante un mensaje del tipo *Inform* para mostrar la ontología. Esta acción contiene un concepto del tipo *Request*.

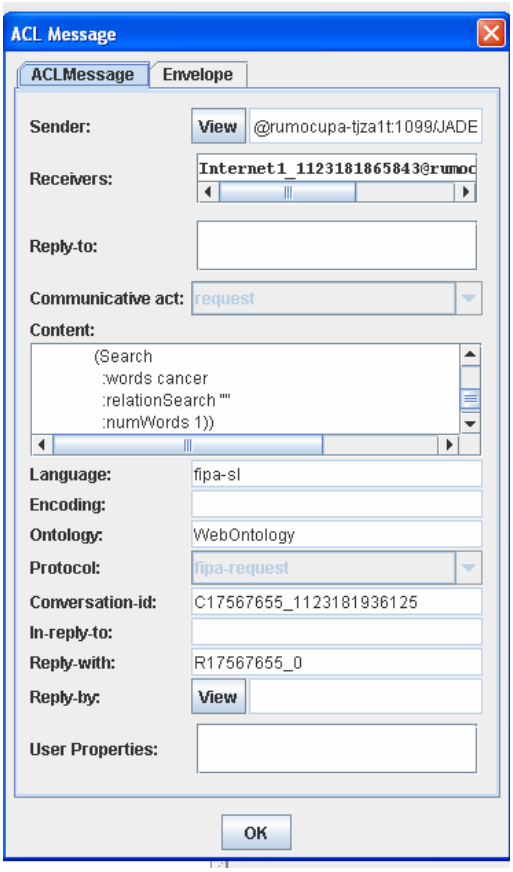
Nombre propiedad	Tipo
req	Request

- Intercambio de mensajes entre agentes

El protocolo que utilizaremos para la comunicación (transmisión de mensajes) entre los agentes del sistema, es el protocolo *FIPA-Request*. Este es, sin duda, el que mejor se adapta a las necesidades del proceso, ya que el agente usuario hace una petición al agente coordinador, con los parámetros de búsqueda y este hace lo propio con el agente Internet. Luego Internet le devuelve un informe de resultados al coordinador, quien trata este informe y le devuelve otro informe al agente Usuario. Este protocolo sirve precisamente para eso, para pedir un servicio a un agente y obtener un resultado.

El agente Usuario inicia un protocolo *FIPA-Request* con el agente Coordinador enviándole un mensaje *Request* pidiendo como servicio que realice la acción *SearchWord*, es decir, la búsqueda y análisis de las páginas Web. Cuando el agente

Coordinador recibe un *Request* comprueba si entiende el contenido y es una petición *SearchWord.*



Figura, extraida de JADE, donde se muestra un mensaje Request concreto. Se quiere crear una Ontología sobre cáncer.

Este le contesta con un mensaje de tipo *Agree*, si todo es correcto, o bien con un *Failure* o *Refuse* y el motivo, si no es correcto. Seguidamente Coordinador envía otro *Request* de *SearchWord* al agente Internet, creado por él previamente, y de nuevo este le contesta con *Agree*, *Failure* o *Refuse*. Cuando el agente Internet acaba la búsqueda le envía un *Inform* con el resultado, es decir, con el contenido de la ontología creada, al Coordinador, y este se lo guarda o lo trata, según sea la primera búsqueda o una continuación, y crea otro *Inform* con la ontología definitiva hacia el agente Usuario quién la mostrara.

El agente Coordinador también mantiene una comunicación con el AMS en la creación y destrucción de los agentes de Internet. En este caso el protocolo también es *FIPA-Request,* ya que de nuevo, se trata de un agente que pide un servicio determinado a otro agente. El proceso es semejante al anterior, Coordinador envía un mensaje del tipo *Request* al AMS, pidiéndole como servicio *CreateAgent* o *KillAgent*, según sea crear o eliminar un agente. El AMS le devuelve, primero un *Agree*, si todo es correcto, y acto y seguido un *Inform* sin información adicional. Para poder realizar esto el coordinador le ha de indicar al AMS, mediante el Request, en qué contenedor se ha de crear éste nuevo agente, el nombre del agente y la clase Java con el código de éste. A partir de aquí Jade tiene todos los datos necesarios para integrar un nuevo agente en el sistema.

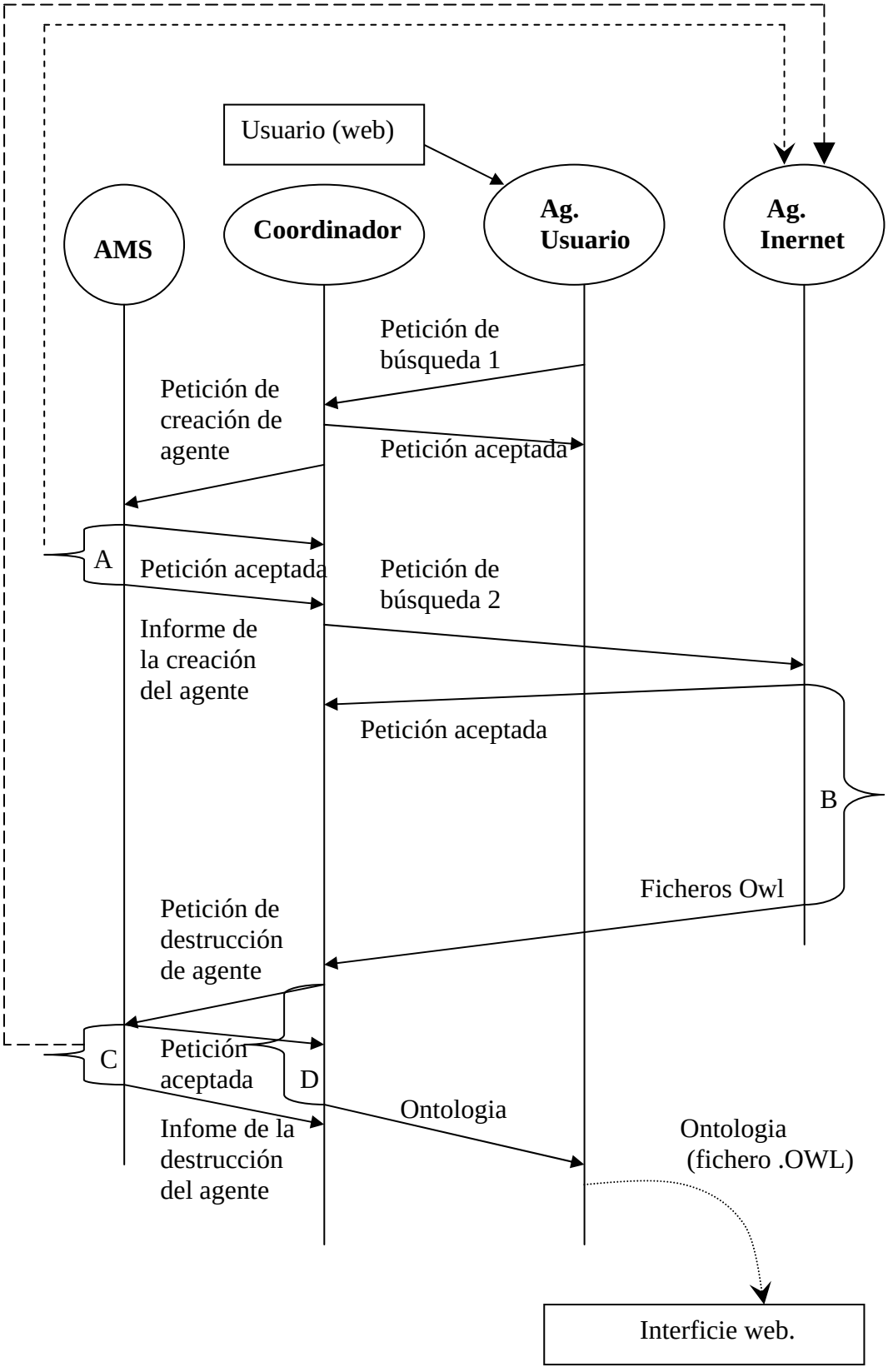
Con tal de explicar, esta acción, con la máxima precisión posible se procede a describir el código de la creación dinámica de Agentes.

- Primero se crea una instancia de la clase CreateAgent;
- Rellenamos esta clase con los datos necesarios para que JADE lo pueda crear.
    - Nombre del agente nuevo.
    - Código de la nueva clase java que ejecuta ese agente.
- Se comprueba si se conoce el nombre del contenedor donde ira el nuevo agente. El contenedor se puede obtener a través del propio agente Coordinador (el contenedor donde esta él).
- Añadimos los argumentos a la clase CreateAgent
- Creamos una instancia de la clase Request, que será enviado junto con el Request.
- Rellenamos la clase acción con la instancia de createAgent.
- Indicamos que la acción que realiza la clase acción la realiza el AMS
- Creamos una Instancia de la clase Request
    - Añadimos a esta clase la Ontología utilizada por el Request.
    - Le indicamos el protocolo a seguir.
    - Le indicamos el lenguaje a utilizar.
    - Le indicamos quien es el emisor.
    - Indicamos que el receptor del mensaje es el propio AMS.
- Enviamos el mensaje junto con la acción.
- Añadimos un nuevo comportamiento al AMS para que cuando reciba un Request de tipo createAgent, proceda a la creación del nuevo agente según la instancia rellenada (createAgent).

En la siguiente figura, se puede observar todo el proceso de la búsqueda. Para no cargar el ejemplo, consideraremos que se trata de una búsqueda de la palabra inicial, o bien, el usuario solo ha seleccionado una frase, con el fin de crear únicamente un InternetAgent. El ejemplo se podría hacer extensible para n búsquedas concurrentes (varias frases seleccionadas).

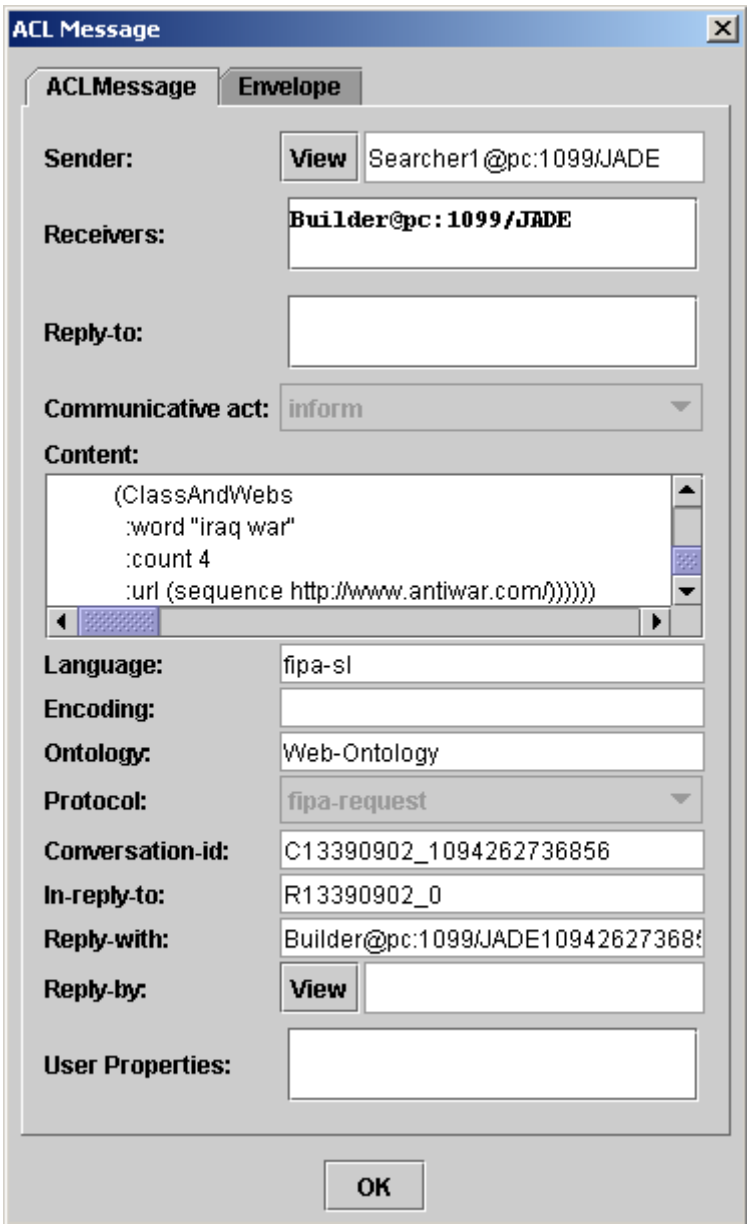
- *Petición de búsqueda 1:* contiene los parámetros introducidos por el usuario desde la aplicación Web, junta con la acción SearchWord.
- *Petición de creación de agente:* petición de creación de un nuevo agente, tal como se ha descrito anteriormente.
- *A (proceso)*: el AMS realiza la acción de creación del nuevo agente.
- *Inform 1:* informe la creación dinámica del agente.
- *Petición de búsqueda 2:* petición de construcción de taxonomía según el deseo del usuario.
- *B (proceso):* ejecución de taxoBuilder, construcción de la taxonomía.
- *Ficheros Owl:* contenido, comprimido, de los dos ficheros OWL generados por taxoBuilder.
- *Petición de destrucción de agente:* petición de destruir al agente Internet puesto que ya ha finalizado su trabajo.
- *Inform 3:* informe sobre la destrucción del InternetAgent.
- *C (proceso):* el AMS realiza la acción de destrucción de InternetAgent.
- *D (proceso):* el coordinador crea la Ontología, en el caso de que reciba mas de una taxonomía las unirá formando la Ontología.

- *Ontología:* informe con el contenido, comprimido, del fichero OWL, de la ontología final.



-

Y en la siguiente figura lo que observamos es la estructura de un mensaje de tipo *Inform* que envía un agente Internet al Coordinador. En este caso el emisor (*sender*) es el agente Internet (*Internet*), el único receptor (*receiver*) es el agente Coordinador (*Coordinator*), 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.



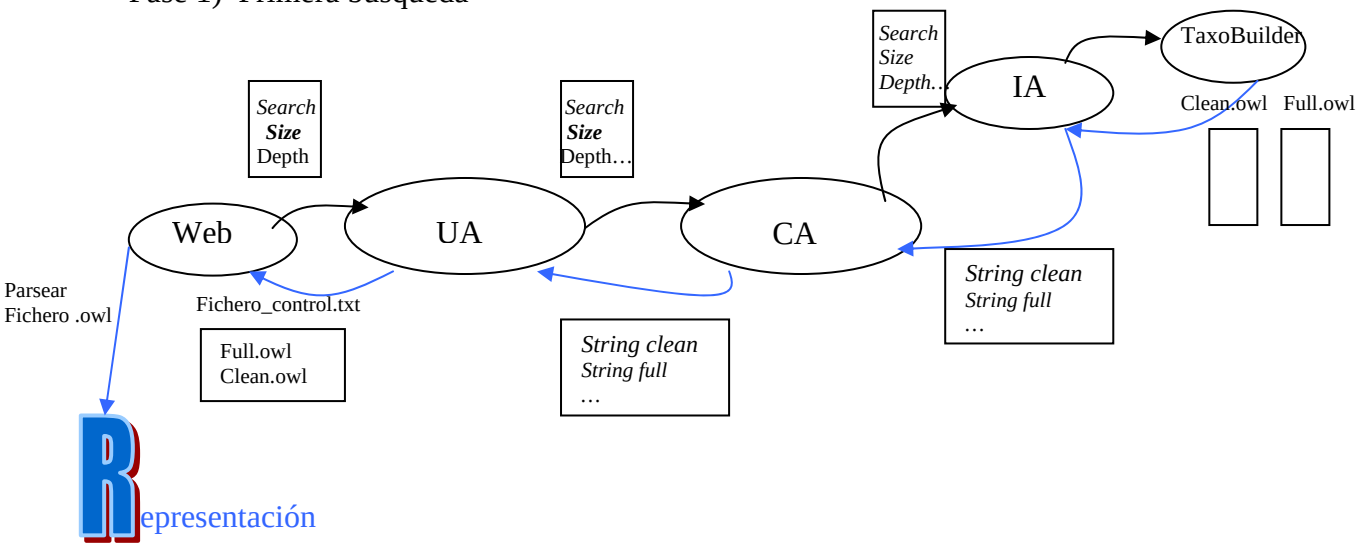
6.5 – Tratamiento de ontologías OWL

Como ya se ha mencionado anteriormente, OWL permite que los resultados de este sistema sean interoperables para así poderlos mostrar en otros programas como Protegeé. Además este lenguaje se ajusta a las necesidades de la aplicación ya que ha sido diseñado para representar ontologías de recursos Web, como es el caso.

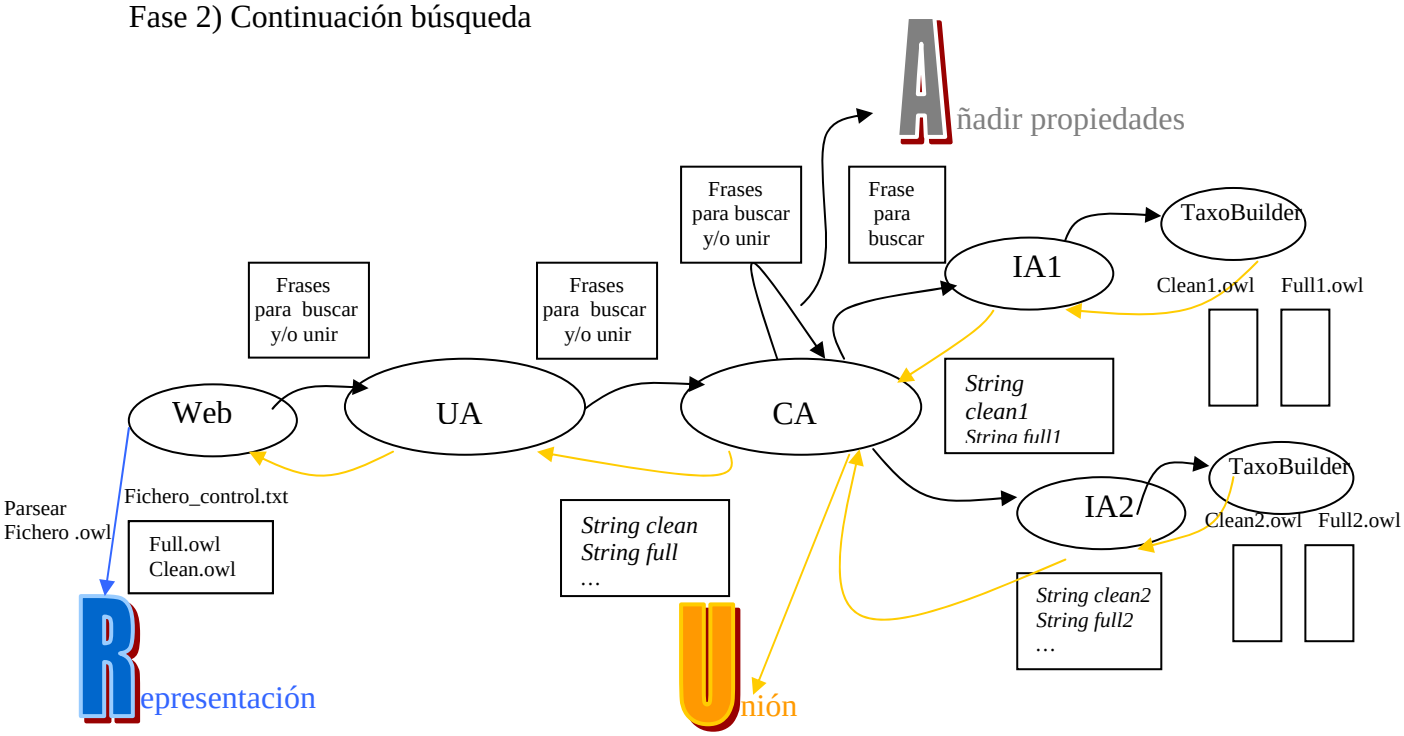
Sobre los resultados procedentes de taxoBuilder básicamente se realizan tres tareas, representación de la Ontología, añadir propiedades y unir varias ontologías (procesos descritos posteriormente). Todas estas tareas se realizan utilizando los métodos que Owl ofrece, pero este lenguaje tiene limitaciones ya sea por su corta evolución o por su incapacidad de almacenar datos (atributos). Uno de los atributos que mas problemas da es el que almacena las URL de una clase ya que frecuentemente contienen caracteres que los métodos de OWL no entiende (p.e &) o bien símbolos propios del lenguaje (p.e. <>). Así pues es necesario diseñar métodos capaces de leer el contenido de un fichero OWL y cambiar esos caracteres por otros que OWL si pueda leer. Es por esto que se devuelve dos ficheros OWL, el completo (fichero full) y el simple (fichero clean). El completo contiene toda la información de la Ontología, es decir, la estructura de clases y subclases junto con sus atributos con su contenido como las URLs asociadas a la clase, la relevancia, etc. Este fichero no se podría visualizar en ninguna aplicación capaz de representar ontologías descritas en OWL con lo que estamos limitando la interoperabilidad que pretendíamos. Para resolver este problema, el sistema también genera otro fichero OWL pero sin los atributos de las clases. Así en estos ficheros solo se describe la estructura jerárquica del dominio. Ahora si que el usuario puede visualizar su contenido en cualquier programa capaz de leer ficheros OWL.

El tratamiento de ontologías realizado es diferente según la fase en la que nos encontremos en la aplicación:

Fase 1) Primera búsqueda



Fase 2) Continuación búsqueda



Representación …

Como hemos introducido anteriormente, las librerías de OWL no funcionan cuando en el contenido de los atributos de las clases aparecen ciertos caracteres. Estos son &, y < y > si no pertenecen a etiquetas, o forman flechas (->). Se ha implementado un parse de OWL propio, capaz de sustituir estos caracteres por cadenas de caracteres que no aparecen en todo el fichero. Los cambios son "-and-", "-menor-"y "–mayor-"respectivamente. Cuando vamos a mostrar las URLs o frases, propuestas para continuar la búsqueda, deshacemos el cambio.

Dado que la representación la hemos querido hacer de manera jerárquica, para que fuese más fácil para el usuario, ver las relaciones entre las palabras, hemos tenido que leer de forma recursiva las distintas clases que componen la ontología. Esto es, primero buscamos entre todas las clases cuales son las "padre", ya que se han podido unir ontologías o añadir propiedades y puede haber más de una.

Una vez que tenemos todas las clases "padre", de cada una de ellas tenemos que saber cuales son sus instancias, propiedades, relevancia, clases "hijas", y de cada clase "hija" lo mismo y así sucesivamente.

Para poder leer la relevancia, URLs y frases, se ha tenido que implementar un método, que trata todo el fichero OWL como una cadena de caracteres, buscando la etiqueta del atributo, de la clase en cuestión, y accediendo a su contenido. Por ejemplo, la relevancia está ubica entre '<a:confidence>' y '</a:confidence>'. Esto es causa de otra

limitación de la librería actual de OWL ya que lo único que permite es leer todos los atributos de la clase, pero no su contenido.

Añadir Property

Esta tarea consiste en relacionar dos clases, que no son de la misma taxonomía, es decir, es el nuevo concepto introducido en la frase, que se añade a la ontología que ya teníamos.
Continuando con el ejemplo del cáncer, supongamos que al usuario se le muestran frases y selecciona las siguientes:

[NP cáncer] [VP recibe] [NP radioterapia]
[NP fumar] [VP provoca] [NP cáncer]

En la primera, la propiedad es 'recibe' (verbo) y el objeto 'radioterapia' (predicado), de tal modo que la 'radioterapia' está relacionada con 'cancer'. Esta relación no taxonómica, se incluirá en la taxonomía, añadiendo la nueva clase (en este caso 'radioterapia') al mismo nivel que 'cancer'.
En la segunda, la propiedad es 'provoca' (verbo) y el objeto 'fumar' (sujeto), de tal modo que el 'cancer' está relacionado con 'fumar'. Esta relación también se incluirá en la taxonomía, añadiendo la nueva clase (fumar) al mismo nivel que 'cancer' a la ontología.

Unión

La unión entre dos ontologías se intentó realizar con las funciones básicas de OWL, extraíamos de los ficheros resultantes de la anterior búsqueda, la Ontología correspondiente y añadíamos como property la clase padre absoluta de la ontología de la búsqueda actual. Se pensó que así, al añadir dicha clase, se añadirían todos los demás atributos relacionados con ella, pero no fue así, solo se añadía esa clase, sin sus clases hijas, instancias, propiedades, y ni tan solo sus anotaciones (atributos). Por lo tanto tuvimos que desestimar esta opción.
De nuevo nos hemos visto obligados a crear una función que nos permita hacer esto, mediante el tratamiento de strings (cadena de caracteres).
En primer lugar, extraemos el string que contiene la Ontología anterior y le quitamos la etiqueta de final de ontología.
Cogemos el String de la Ontología actual, que ha mandado el InternetAgent, le quitamos la cabecera y la etiqueta de fin de la ontología. Como las etiquetas que abren y cierran las anotaciones se llaman igual que la clase padre absoluta (vease ejemplo), las tenemos que sustituir por la clase padre de la anterior ontología. Una vez hecho esto, concatenamos este String con el anterior y añadimos la etiqueta de fin de ontología.
Llegados a este punto, esta ontología, si hubiese que unirla con otra se haría de la misma manera, en este caso se unirían la nueva, obtenida tras la unión, y la actual, recibida por el InternetAgent, y así sucesivamente hasta que no quedasen búsquedas pendientes.
De esta manera obtenemos la ontología resultante de la unión de dichas ontologías, y se la enviamos al UserAgent.

Ejemplo:

Anotación ontología anterior: **<cancer:confidence>456</cancer:confidence>**
Anotación nueva ontología**:<radioterapia:confidence>34</radioterapia:confidence>**

Al unir la nueva Ontología con la antigua, las etiquetas quedarían de la siguiente forma:
<cancer:confidence>34</cancer:confidence>

6.6 – Aplicación Web

6.6.1 – Justificación del uso de una aplicación Web

El propósito de esta aplicación es proporcionar al usuario una herramienta fácil para construir una ontología. Esto se consigue permitiéndole consultar los resultados parciales de forma amigable e interactiva, guiándole de forma intuitiva a lo largo de la construcción.
Los resultados se muestran automáticamente, ya sea de la búsqueda actual o de búsquedas guardadas. Estos se mostrarán como estructuras jerárquicas del área del conocimiento que el usuario desee explorar.
Finalmente permite un fácil acceso a los recursos Web de cada elemento de la ontología resultante.

6.6.2 – Funcionamiento de la aplicación Web

La aplicación Web consta de estas fases:

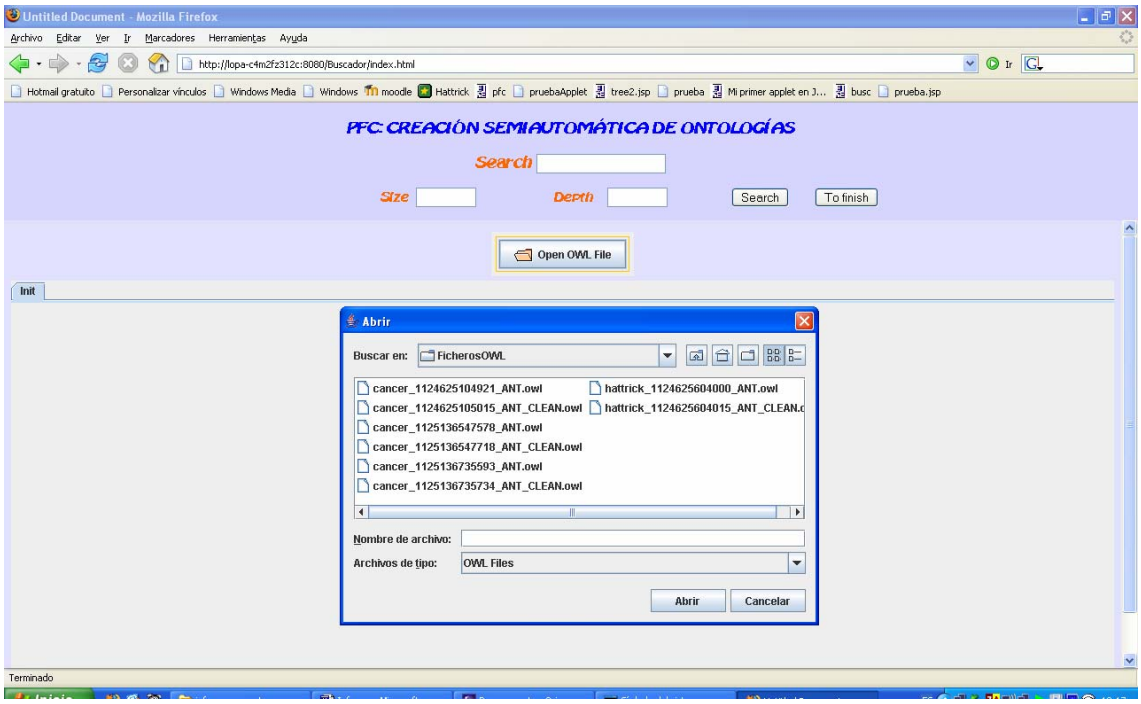
- Petición de búsqueda o abrir búsqueda ya realizada.
- Obtención,
- Interpretación,
- Representación de los resultados OWL y almacenamiento.

Petición de búsqueda o abrir búsqueda ya realizada

Una vez que el usuario introduce los parámetros y hace clic en “Search”, la aplicación crea dos agentes: el CoordinatorAgent y el UserAgent, pasándole a éste último los parámetros introducidos por el usuario. Si es la primera vez que éste accede a la aplicación, el sistema guardará un fichero de identificación de usuario, en la máquina de éste, es decir, una cookie “permanente” (véase apartado 5), que proporcionará a la aplicación información relativa de su búsqueda para posteriores accesos al sistema. Esto permite al usuario desconectarse del sistema, mientras éste busca (proceso que puede tardar) y cuando se vuelva a conectar, visualizará los resultados, si la búsqueda ha concluido. El usuario podrá hacer una nueva búsqueda si finaliza la que está en proceso, haciendo clic en “to finish”, de esta manera se borra la cookie y se matan los agentes que realizaban esa búsqueda.

Para conseguir esta persistencia del servicio, como hemos introducido, se tiene que guardar una cookie con el identificador de los agentes que dan servicio a ese usuario. Además en el servidor se almacena un 'bean' de aplicación (véase apartado 5), que contiene una lista con la información relativa a las búsquedas de cada usuario. Se puede acceder a la búsqueda de un usuario concreto buscando en la lista el identificador guardado en la cookie, recuperando el contexto de esta búsqueda para mostrársela al usuario.

El usuario también puede abrir una ontología guardada de una búsqueda pasada, para ello, solo debería hacer click al botón "Open OWL File", éste siempre mostrará el contenido del directorio "Ficheros OWL", creado en el directorio raíz del disco duro, pero el usuario es libre de seleccionar cualquier otro.



Obtención de resultados

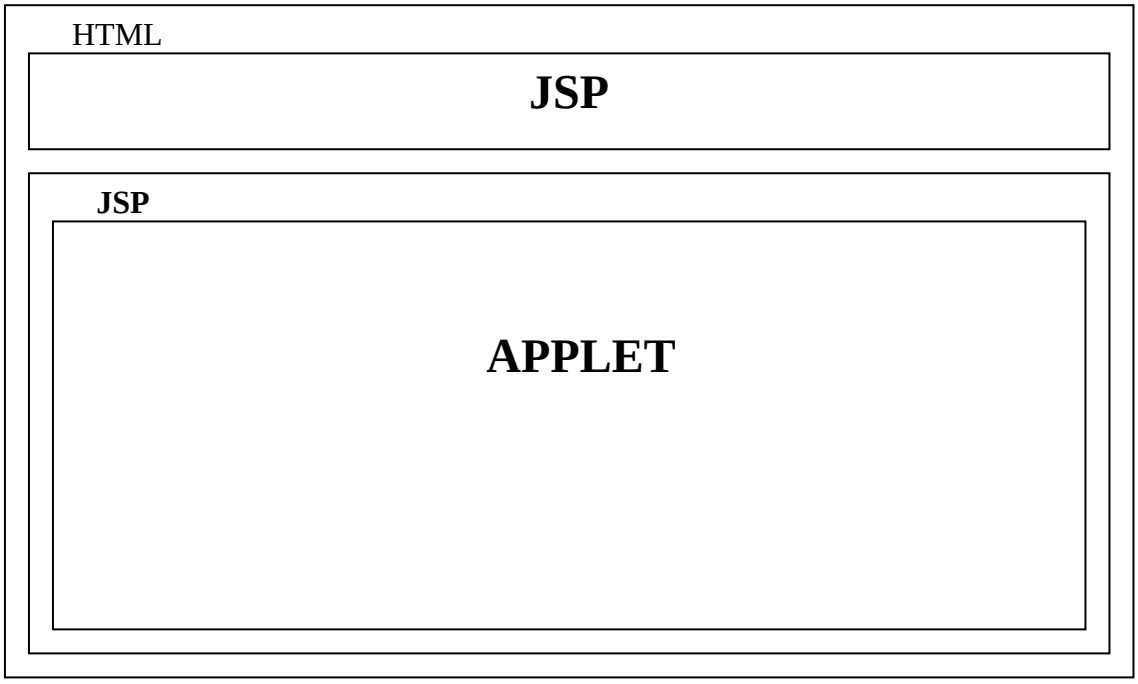
Como hemos mencionado anteriormente, los resultados llegan a nuestra aplicación a través de ficheros, procedentes del UserAgent (agente interficie entre aplicación Web y sistema multi-Agente). A priori, la aplicación no conoce el nombre de dichos ficheros, pero sí conoce el identificador de los agentes que ha creado, por eso, antes el Agente Coordinador crea un fichero de control, que tiene como nombre "controlFich" más el identificador de agentes, donde estarán guardados los paths de los ficheros .OWL resultantes. De esta forma, la aplicación Web accede a este fichero para averiguar que ficheros tiene que mostrar al usuario.

Interpretación de resultados

De cada clase guardamos sus clases hijas, instancias y propiedades, esto nos sirve para poder crear un árbol en el que de cada nodo cuelgan sus clases hijas e instancias y así recursivamente.
Además de esta información también guardamos la relevancia (atributo del fichero OWL) que como indica su nombre, nos indica que importancia tiene esa clase dentro de la Ontología.

Representación de resultados

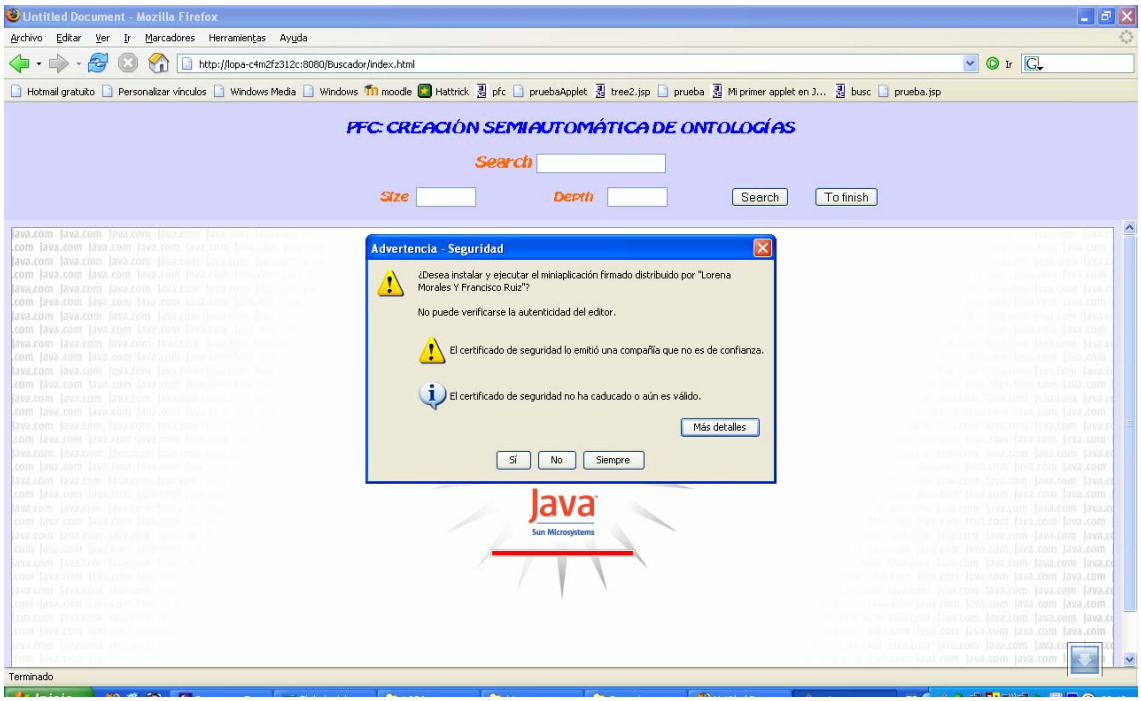
Los resultados que ha de mostrar esta aplicación van cambiando según la búsqueda, por lo tanto se han de mostrar en estructuras dinámicas, de estas estructuras, denominadas páginas dinámicas, se han utilizado los JSP (explicados anteriormente). Estos no permiten mostrar estructuras jerárquicas dinámicas (árboles), por este motivo incluimos un applet que si que lo permite.



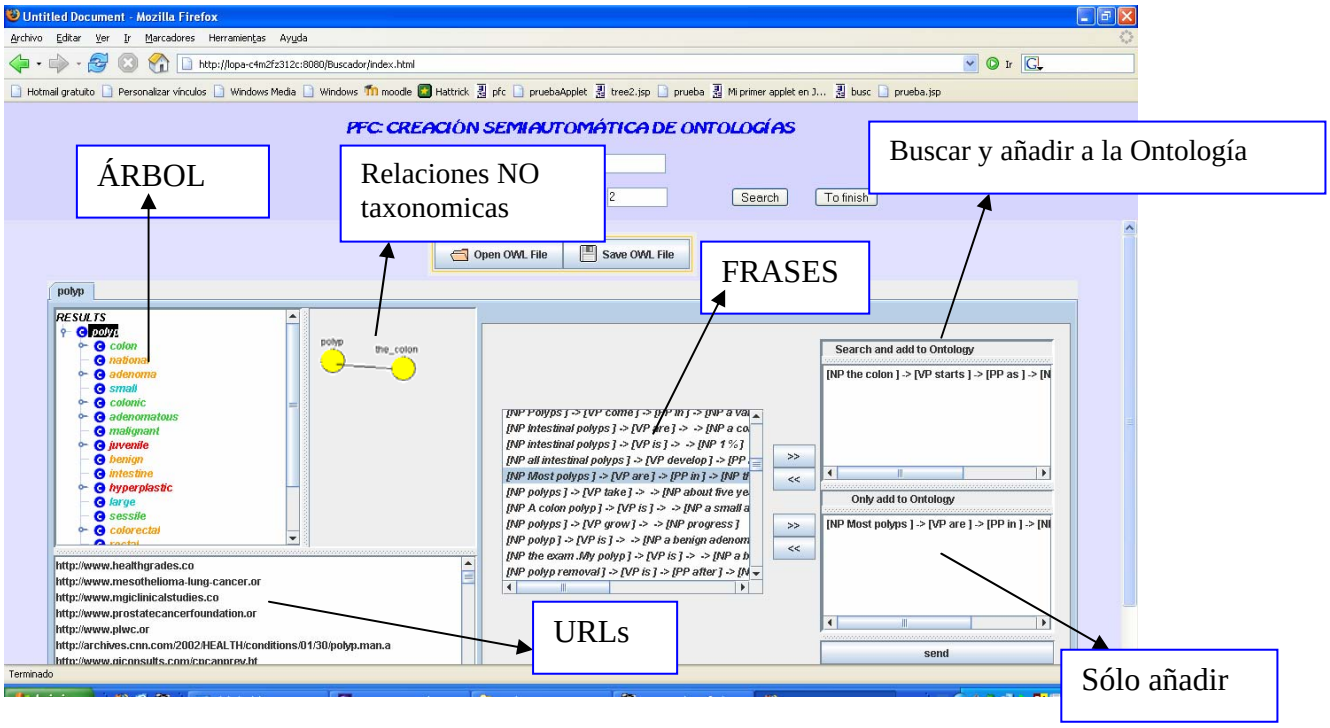
Este applet mostrará por un lado, la jerarquía de clase y subclases (árbol), gráficos de relaciones no taxonómicas, lista de URL's relacionadas con cada clase y frases propuestas por el sistema para continuar la búsqueda. Por otro lado, permite visualizar ontologías guardadas, a través de pestañas, o guardar la actual, ficheros OWL, (ha aparecido el botón, "Save OWL File") y el botón send para continuar la construcción.

Como hemos explicado anteriormente los applets, tienen restricciones de seguridad, por eso para poder leer los ficheros OWL del usuario o para poder guardarlos, nos hemos visto obligados a hacer un certificado, que nos permita estas tareas. Este no ha sido suficiente, puesto que solo nos ha concedido permisos de escritura, pero no de lectura,

por eso, cuando el usuario quiere abrir un fichero, el applet debe crear en el directorio del usuario el archivo ".java.policy", concediendole permisos de lectura.



Al iniciar la aplicación Web se le pregunta al usuario, si acepta o no el certificado.



En la anterior imagen se puede ver como las clases están de diferentes colores, según su grado de relevancia:

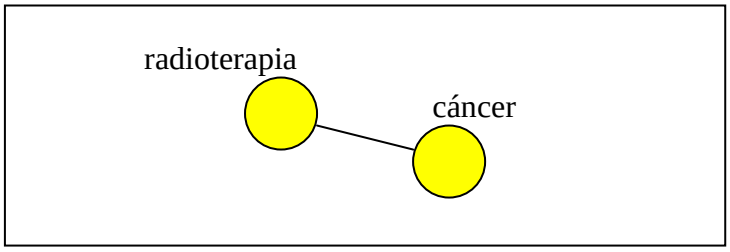
- Rojo
| Naranja
| Azul
+ Verde

De esta manera de un simple vistazo podrá ver que es lo que más o menos le conviene para su búsqueda.

Al hacer click en cualquier elemento del árbol (ya sea nodo o hoja), en la parte inferior de la página, se mostrarán las URL's relacionadas con dicho elemento y a su vez en la parte derecha se podrán ver frases también relacionadas con dicho componente.
Al seleccionar cualquiera de las URL's se abrirá una nueva ventana con la página.
El usuario podrá seleccionar todas las frases que quiera y agregarlas según le convenga utilizando los botones de flechas. Si las introduce en la lista de buscar, significará que desea más información de dichas frases y si las introduce en la lista de añadir, significará que solo desea que se introduzca en la búsqueda anterior sin buscar más información sobre ellas. Al ir seleccionando estas oraciones se irá creando un gráfico de las interrelaciones de éstas con las clases. Como se ve en la figura anterior (Relaciones no taxonómicas).

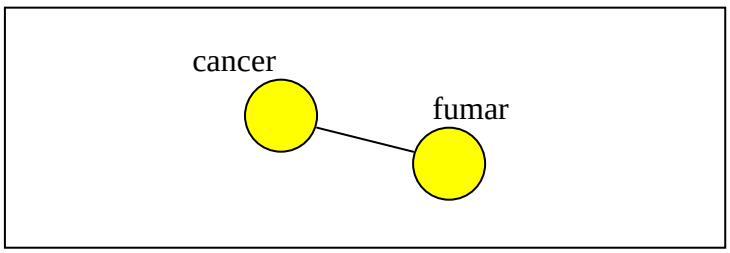
Estas relaciones no taxonómicas se muestran de la siguiente manera:

Si el usuario ha buscado cáncer y ha seleccionado la frase "cáncer recibe radioterapia", para buscar y/o añadir en la ontología, el gráfico de la relación se mostrará de la siguiente manera:



Este gráfico se visualizará cuando se pulse la clase radioterapia y no cuando se pulse cáncer, ya que radioterápica que es la que está relacionada con cáncer y no al revés (cáncer es sujeto).

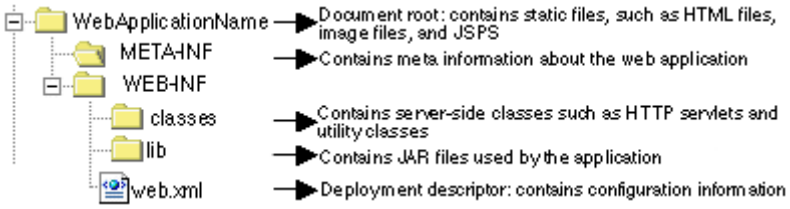
En cambio si el usuario selecciona "fumar provoca cáncer", el gráfico será:



Ahora se mostrará cuando se pulse cáncer, por que esta vez cáncer es el que está relacionado con fumar y no al revés (cáncer es predicado).

6.6.3.4 – Crear una aplicación Web

La especificación Servlet 2.2 define la estructura de directorios para los ficheros de una aplicación Web. El directorio superior -- o directorio raíz -- debería tener el nombre de la aplicación y definirá la **raíz de documentos** para nuestra aplicación Web. Todos los ficheros debajo de esta raíz pueden servirse al cliente excepto aquellos ficheros que están bajo los directorios especiales META-INF y WEB-INF en el directorio raíz. Todos los ficheros privados -- como los ficheros class de los servlets -- deberían almacenarse bajo el directorio WEB-INF.
En la siguiente figura podemos ver la estructura de directorios de una aplicación Web:



Para crear la aplicación Web, empezamos creando esta estructura de directorio. Si tuviésemos algún servlet se situaría su clase en el directorio WEB-INF/clases en el que también se incluyen nuestras otras clases java.

Si hemos definido que nuestro servlet o clase pertenece a un paquete, debemos seguir las reglas estándar de Java y crear los subdirectorios apropiados para que la JVM pueda encontrar nuestras clases. Por ejemplo, si nuestra clase UserAgent está definida en un paquete Agentes, deberíamos crear la siguiente estructura de directorios:

```
.../WEB-INF
   |--
   classes
      |--
      Agentes
```

Tendríamos que situar dicha clase en el subdirectorio Agentes.

Una alternativa útil para copiar los ficheros de clases al directorio apropiado es configurar nuestro entorno de construcción (un Makefile o IDE) para salvar las clases compiladas directamente en los directorios requeridos. Hacer esto nos ahorrará este paso durante el desarrollo.

Una vez tenemos todos nuestros ficheros en su lugar, si tenemos servlets, necesitamos realizar otra tarea: actualizar el descriptor de despliegue para registrar nuestros servlets con el contenedor. Para crear fácilmente un descriptor de despliegue, simplemente editamos uno existente. Abajo tenemos un esqueleto de un fichero web.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<Web-app>

   <!-- Tus definiciones van aquí  -->

</Web-app>
```

Insertamos nuestros descriptores de despliegue de servlets entre las etiquetas <Web-app> y </Web-app> de este fichero. El descriptor de despliegue de un servlet debe incluir las siguientes etiquetas (en este orden):

```
 <servlet>

   <servlet-name>nombre</servlet-name>

   <servlet-class>package.nombre.MiClass</servlet-class>

 </servlet>
```

También están permitidas antes de la etiqueta de cierre </servlet> varias etiquetas opcionales, que definen las propiedades de ejecución del servlet,. Estas etiquetas definen propiedades como parámetros de inicialización, si el servlet debería o no cargarse en la arrancada, los roles de seguridad, y las propiedades de pantalla (incluyendo iconos grandes y pequeños, nombre de pantalla y una descripción).
Hasta ahora, nuestro descriptor de despliegue ha descrito el servlet al contenedor de servlets. Luego, debemos describir cuándo el contenedor de servlets debe invocar al servlet -- nos referimos a esto como *mapeo*. En otras palabras, debemos describir cómo se mapea una URL al servlet. En el fichero web.xml, las URLs se mapean de esta forma:

```
 <servlet-mapping>

   <servlet-name>nombre</servlet-name>

   <url-pattern>pattern</url-pattern>

 </servlet-mapping>
```

Veamos un ejemplo de un descriptor de despliegue de una aplicación Real. Abajo podemos ver el fichero web.xml mínimo que describe nuestro servlet de ejemplo RequestDetails:

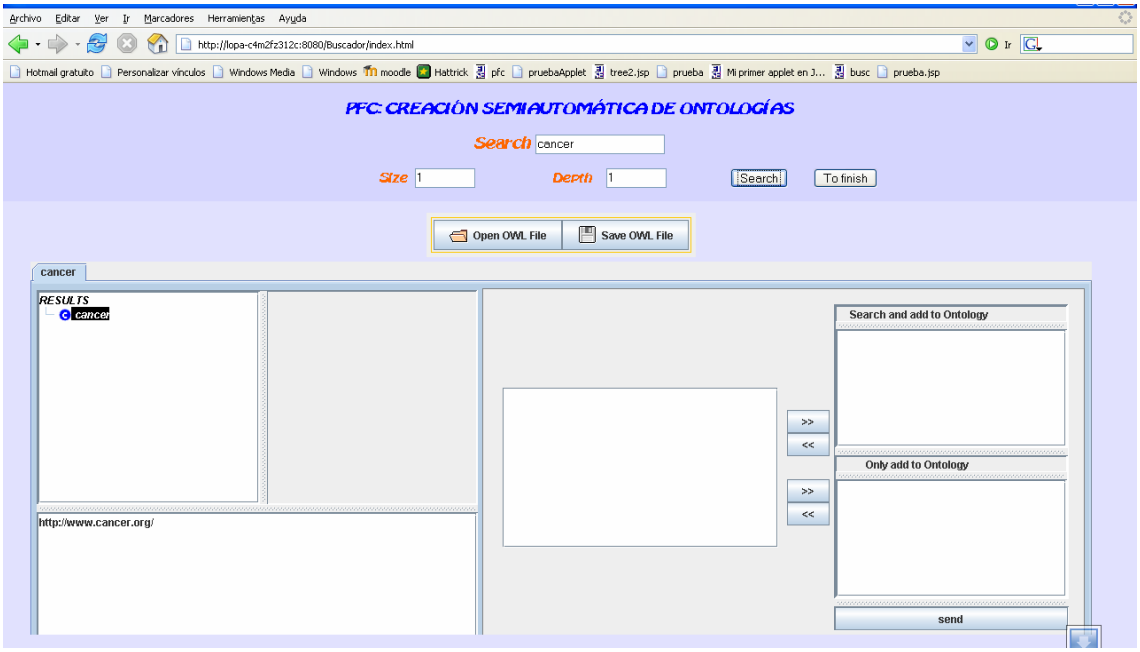
```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

```
<web-app>

<servlet>
  <servlet-name>RequestDetails</servlet-name>
  <servlet-class>org.stevengould.javaworld.RequestDetails</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>RequestDetails</servlet-name>
  <url-pattern>SampleServlet</url-pattern>
</servlet-mapping>

</Web-app>
```

7 - EVALUACIÓN

Se han realizado varias pruebas para probar el correcto funcionamiento del sistema, todas ellas realizadas en un portátil centrino de 1,73 GHz de CPU, 2 MB de memoria caché y 1024 MB de RAM, con una conexión ADSL de 512 KB.

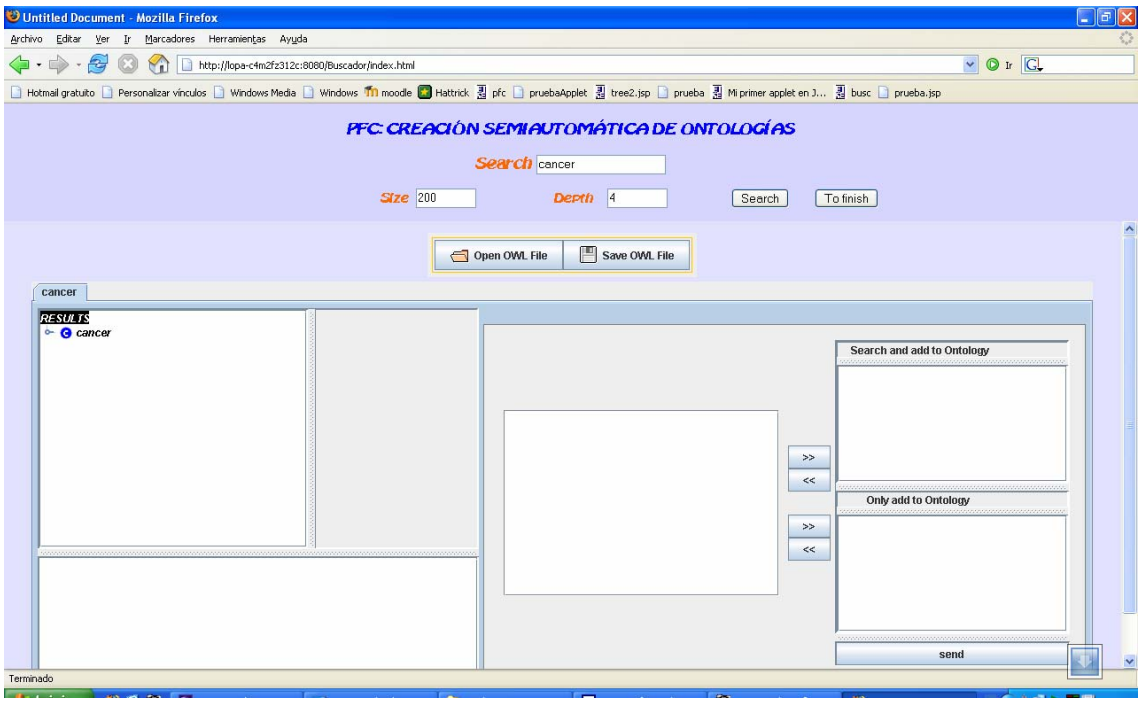
Palabra a buscar	cáncer
Depth	1
Size	1

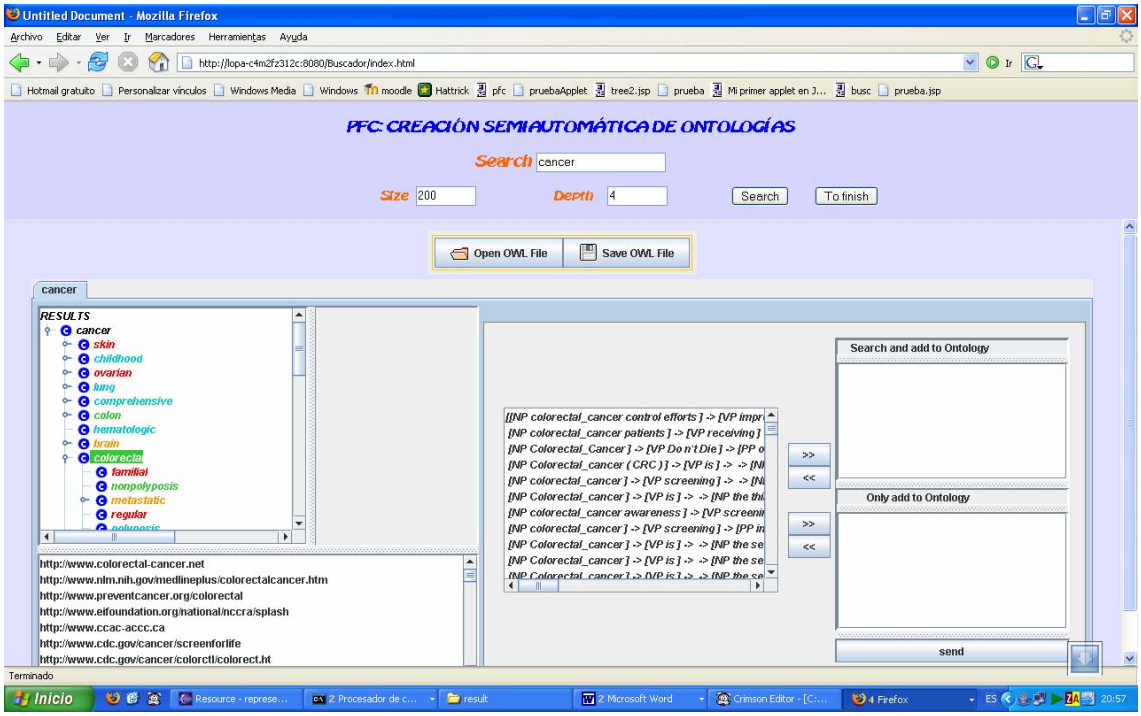


Es una prueba, rápida y sencilla. Entre la búsqueda y la representación, no tarda más de un minuto.

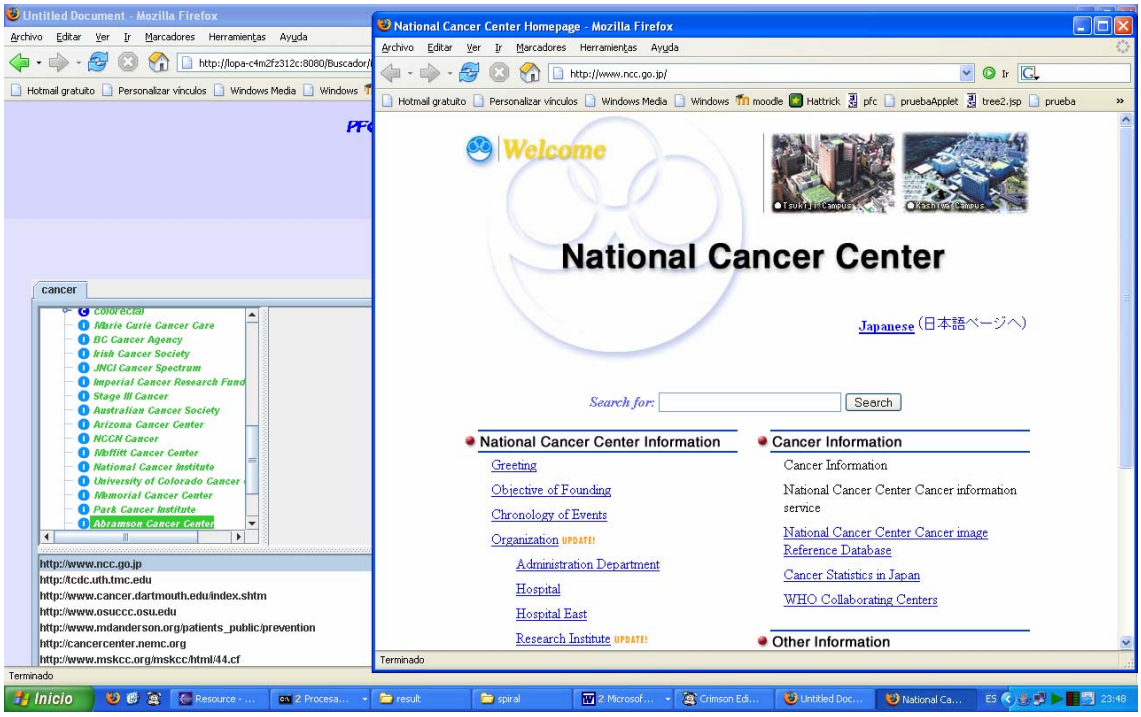
Palabra a buscar	cáncer
Depth	4
Size	200

Búsqueda con frases y propiedades, ha tardado aproximadamente 8,5 horas.

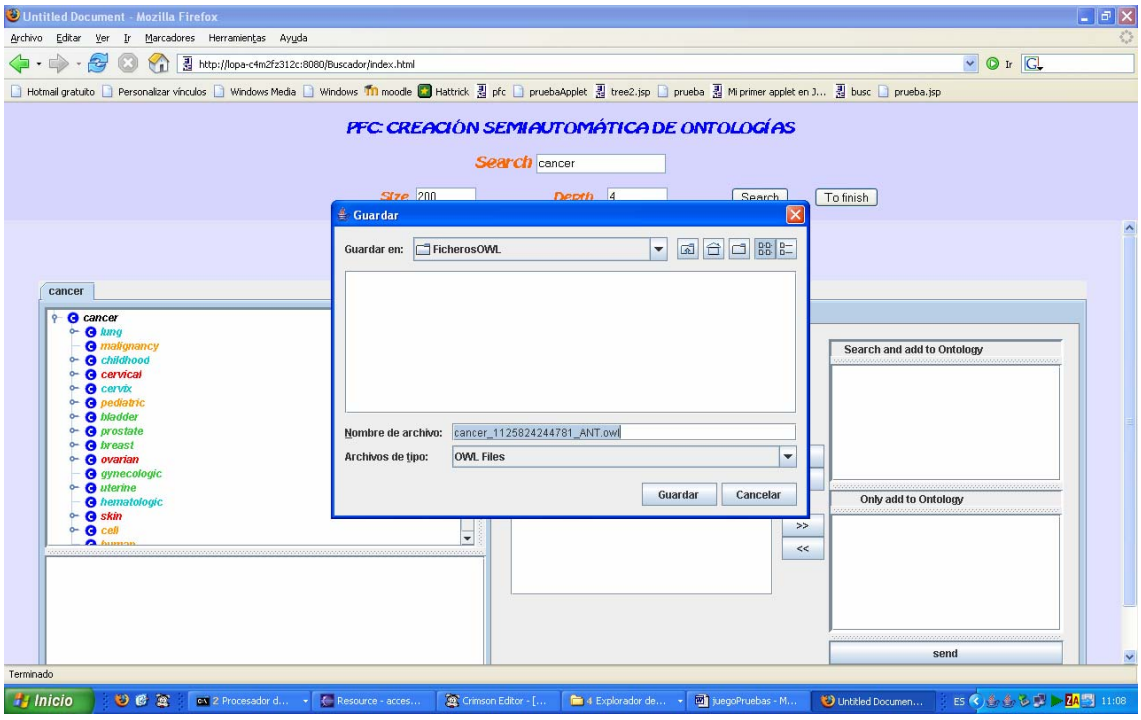




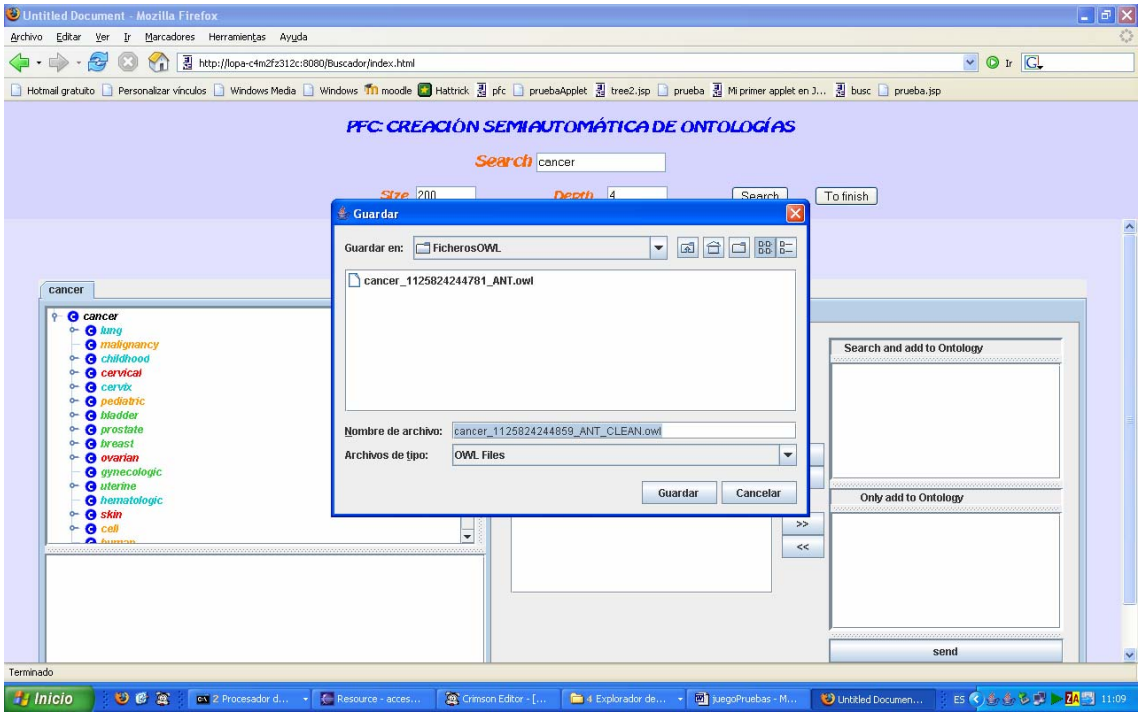
Se puede observar como la estructura está clasificada por colores según la relevancia.



Aquí se puede ver como cuando se selecciona una URL se abre la página automáticamente.

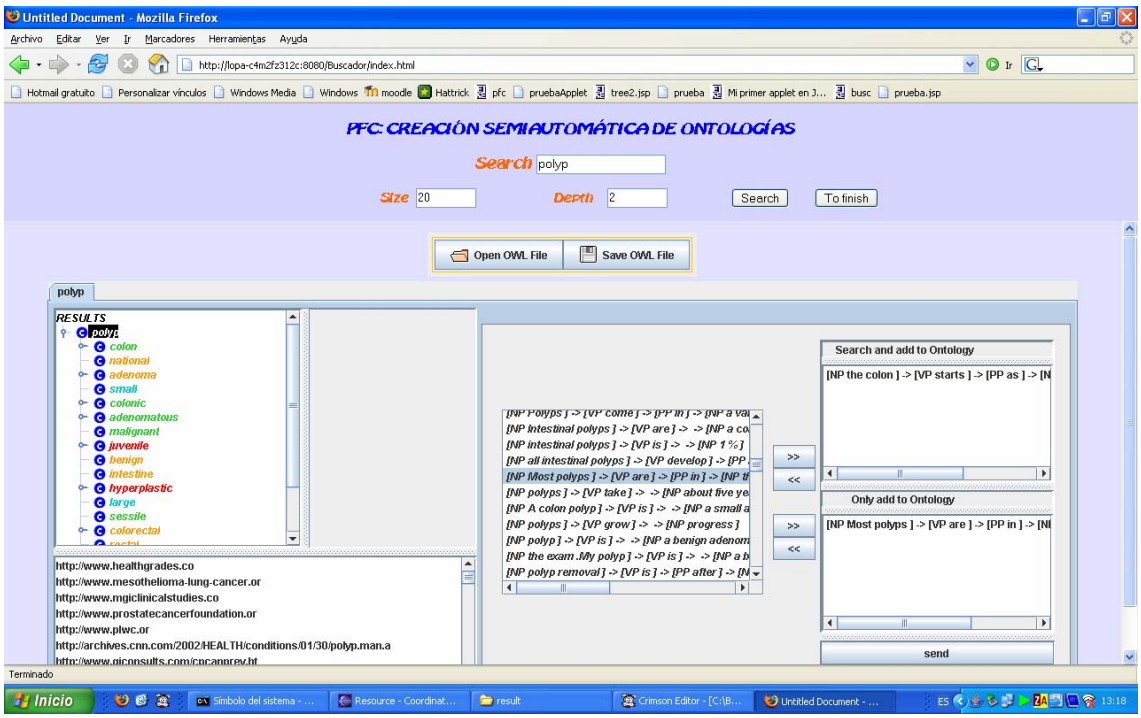


Cuando se le da al botón Save OWL File se abre primero esta ventana, en la que se indica que se va a guardar el archivo full y en la siguiente figura el archivo clean.

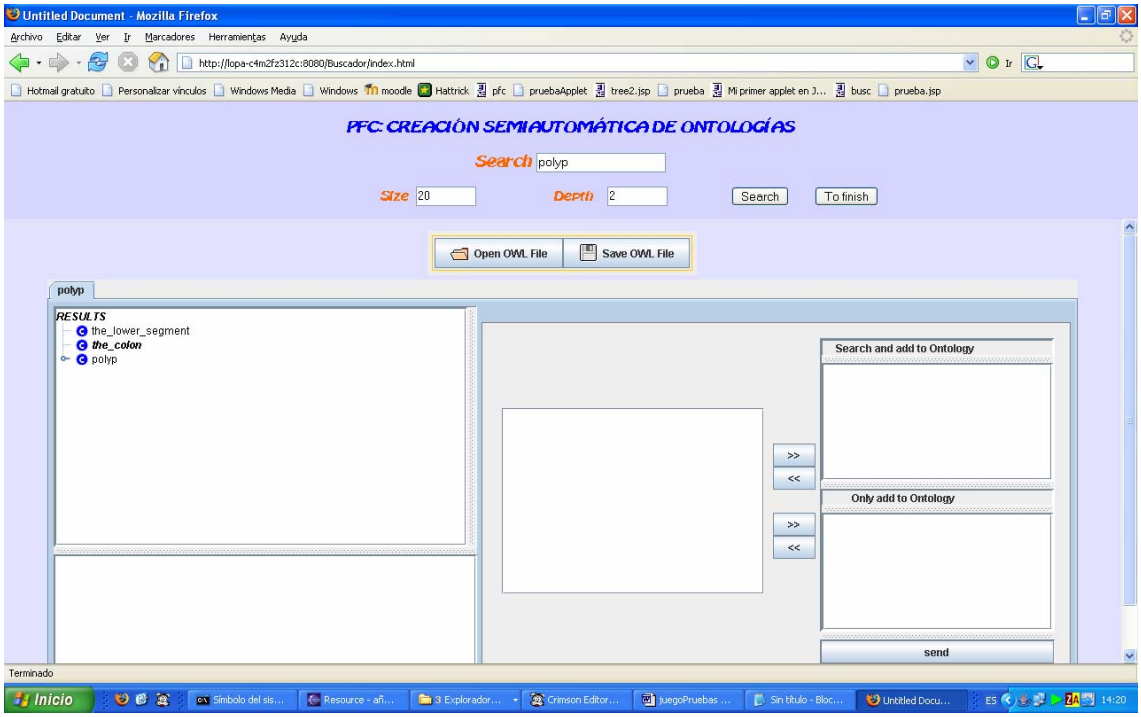


Palabra a buscar	Polyp
Depth	2
Size	20

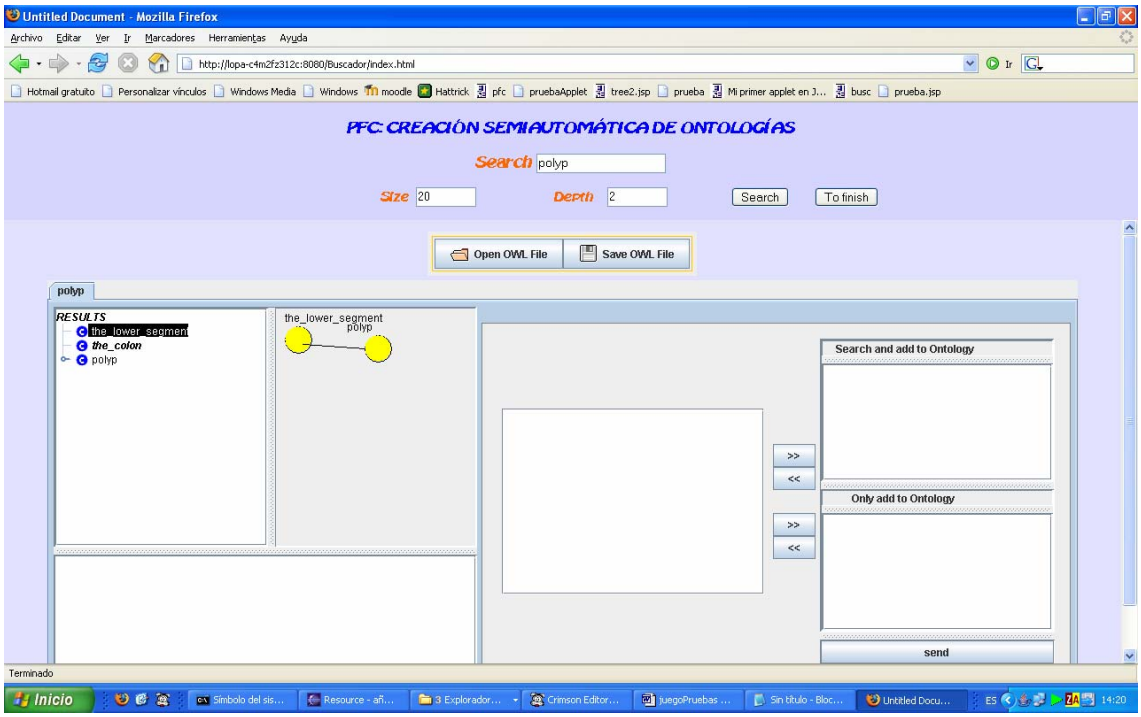
Ha tardado aproximadamente 20 minutos.



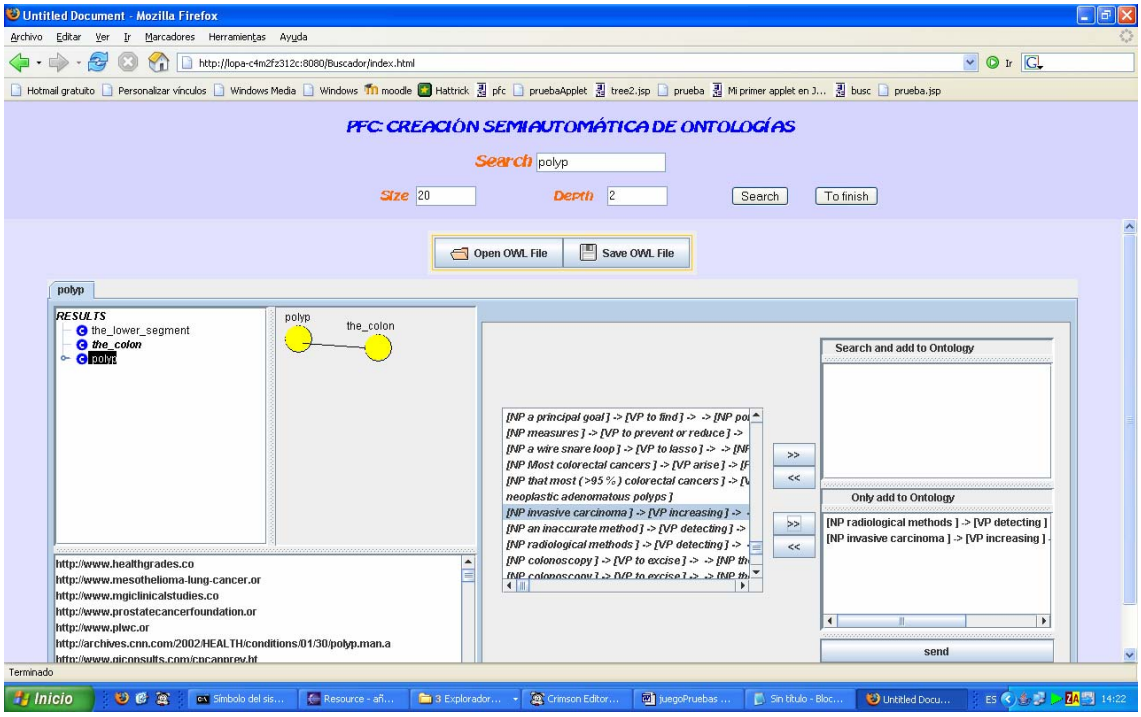
En esta figura podemos ver como añadimos dos frases a la ontología y una de ellas continua la búsqueda.



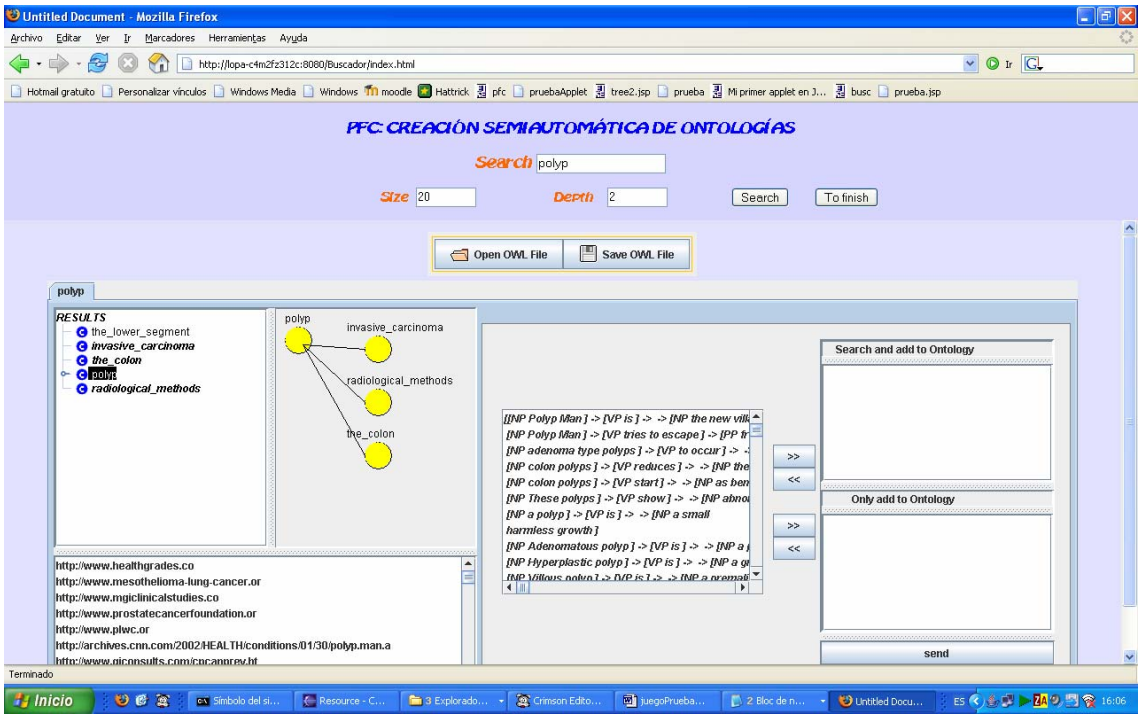
Se puede observar en diferente grosor que clases tienen propiedades



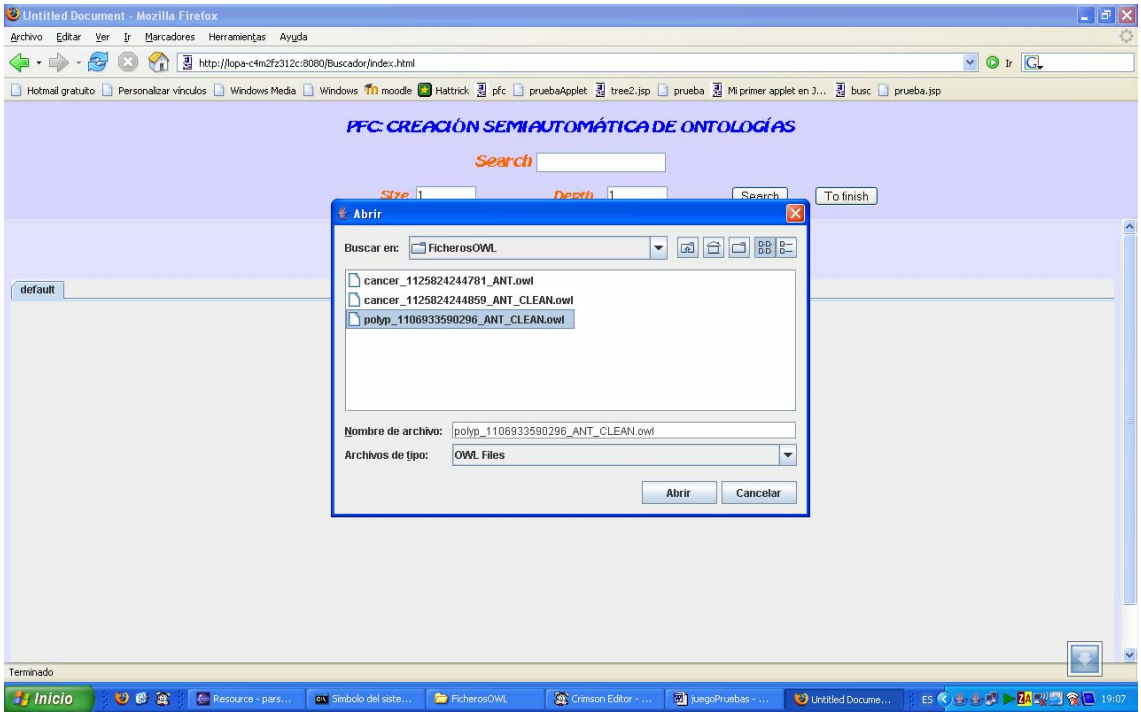
Como en la frase polyp era sujeto la relación sale cuando se pulsa la nueva clase.
El fichero clean de esta ontología se ha guardado para una posterior visualización.



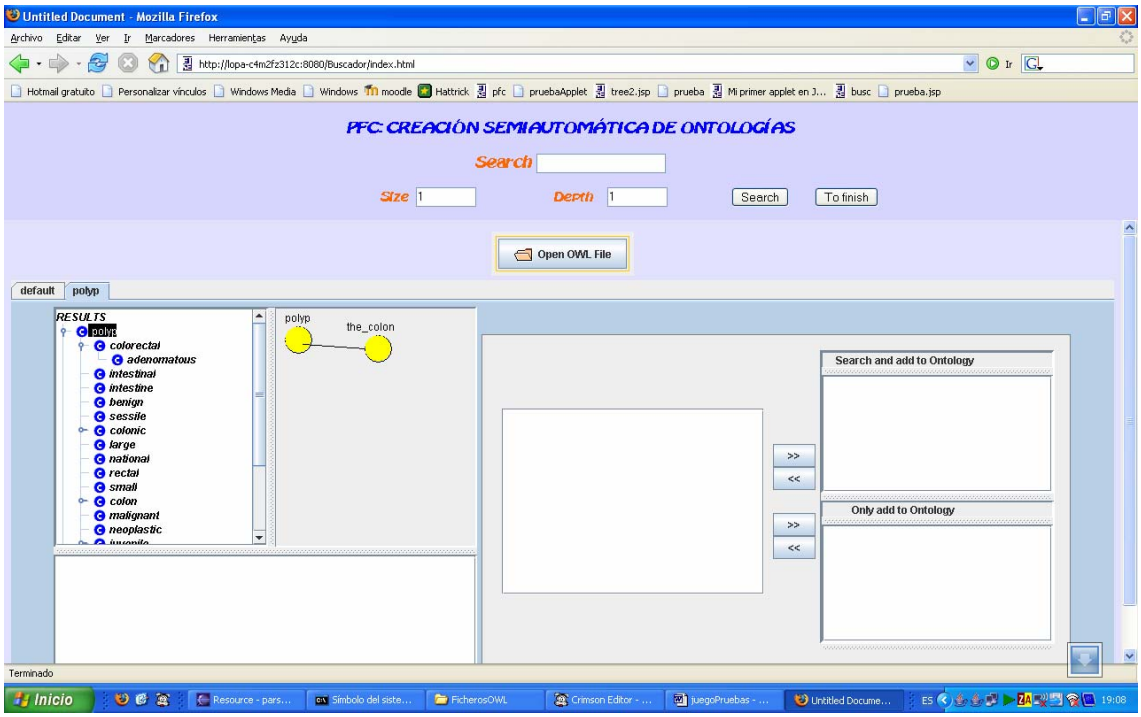
Como en la frase polyp era predicado, la relación sale cuando se pulsa éste.
Se añaden dos nuevas frases.



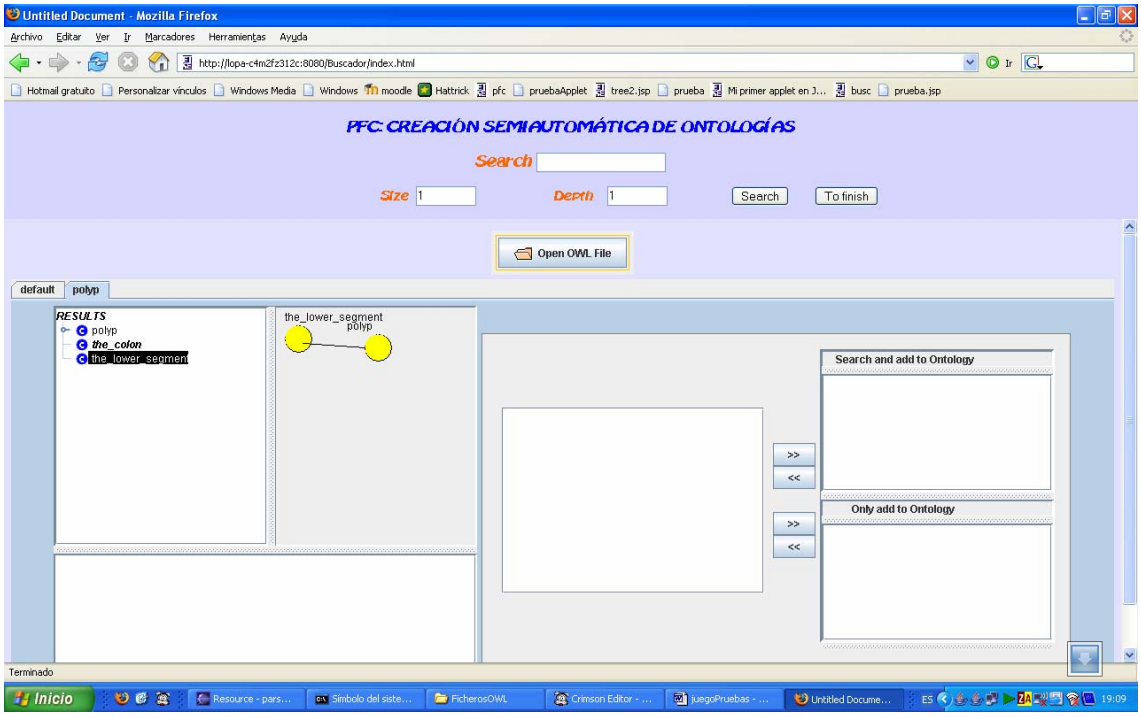
Se muestran las nuevas relaciones.



Se abre el fichero CLEAN guardado anteriormente.



Como se puede observar ahora no se muestran las relevancias, frases y URLs, pero si las relaciones.



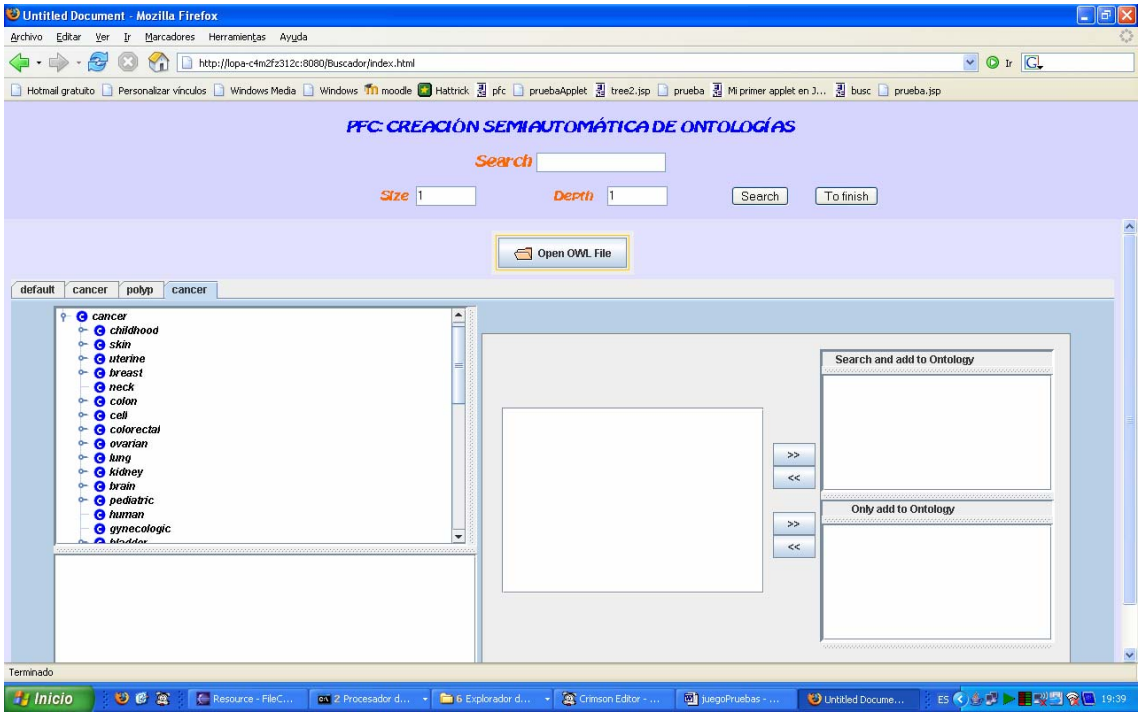


Imagen que demuestra que se pueden ver varios ficheros a la vez.

En todas estas pruebas se ha podido demostrar el correcto funcionamiento mostrando un resultado claro e intuitivo para el usuario.

8- CONCLUSIONES

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.
Del diseño del sistema se pueden extraer las siguientes conclusiones. En primer lugar, implementar un sistema multi-Agente, para alcanzar los objetivos, además de las ventajas ya mencionadas, facilita el planteamiento del diseño y la distribución de las tareas entre unidades independientes (en este caso agentes), además de poder resolver problemas mucho mas complejos, ya que seria inviable o extremadamente complicado mediante *Threads o Fork.* Por otro lado, señalar la importancia de las Ontologías para poder tratar conocimientos computacionalmente y que este conocimiento sea estándar e interoperable. Sin una Ontología (OWL), una máquina nunca sabrá las relaciones existentes entre los conceptos de un dominio para poder tomar decisiones o realizar tareas con ellas, usándolas a posteriori para procesos de tratamiento de datos. Por último, destacar la importancia del constructor de taxonomías, ofreciendo unos resultados de muy alta calidad.

La realización de este proyecto, ha sido muy enriquecedora, tanto en los conocimientos adquiridos como en las aptitudes adoptadas. A pesar de haber realizado numerosas prácticas durante la carrera, y estas se han podido realizar por varias personas, ha sido gracias al proyecto, debido a su mayor complejidad, donde realmente hemos consolidado nuestra capacidad de colaboración y desarrollo de un objetivo común. Hemos tenido que discutir las decisiones, dar soluciones conjuntas a problemas globales, también nos hemos visto obligados a ser más estructurados y claros en nuestros diseños, etc. Por otra parte, hemos adoptado una visión global de casi todas las asignaturas cursadas durante la carrera, obteniendo una mejor consciencia de lo aprendido, es decir, los conocimientos adquiridos en la carrera son mas de los que uno piensa. También hemos aumentado nuestra capacidad de investigación, evidentemente hay muchos temas en los que tienes la base pero necesitas profundizar o bien concretizar, por ejemplo crear applets, arrancar agentes desde una aplicación Web, solucionar las restricciones de seguridad de los applets, etc. Finalmente, nos gustaría destacar la valiosa ayuda, tanto en conocimientos y experiencia como en consejos y apoyo, de los directores del proyecto guiándonos en todo momento por el camino adecuado.

9- TRABAJO FUTURO

Para hacer este un proyecto lo mas beneficioso posible nos gustaría proponer algunos trabajos futuros basados en él.

Si aprovechamos la arquitectura de este proyecto, se podría diseñar un servidor (red de computadores) que contara con un numero determinado de maquinas, un balanceador de carga (load balancer), un sistema de almacenamiento rápido y con suficiente capacidad (RAID), un sistema de backups, y una conexión de red con una alta velocidad de transmisión. De esta forma este proyecto, posiblemente con algunas modificaciones, seria capaz de dar un servicio rápido, de calidad y alta conectividad. Además en esta línea de trabajo, también se le podría dotar al agente coordinador de métodos de planificación de tareas, para ajustar la ejecución de los agentes a los recursos disponibles.

Otro posible trabajo futuro, podría ser la incorporación de una base de datos con búsquedas y resultados parciales en el servidor para optimizar búsquedas futuras. Esto además de mejorar el almacenamiento y acceso de los resultados, también liberaría al sistema actual de esa tarea, teniendo que preocuparse solo de realizar consultas (querys) a la base de datos.

También se podría crear nuevas versiones del sistema utilizando mas funciones del constructor de taxonomías, tales como desambiguar polisemias o tratar sinónimos. Como también mejorarlo realizando un estudio de la casuística de unión de ontologías parciales para evitar redundancias, ciclos o ambigüedad en los resultados.

Una última propuesta, seria la de crear un “workspace” (espacio de trabajo) de búsqueda en el servidor, donde varios usuarios accedan a él ya sea para construir o para consultar Ontologías. Es decir un usuario podría acceder a ese workspace elegir una Ontología creada por otro usuario y consultarla o bien seguir su construcción.

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, jadeTools.jar, Base64.jar, iiop.jar, http.jar

http://jade.cselt.it/

- OWL: para crear y leer 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/

- taxo.jar, para realizar la búsqueda por internet y crear la ontología, esta librería es una modificación de una aplicación que realizaba esta tarea, por lo que no se puede obtener de ninguna url. No obstante ha sido creado por GruSMA y en esta Web hay mucha información relacionada.

http://www.etse.urv.es/recerca/banzai/toni/MAS/

- Apache Tomcat: Servidor Web necesario para poder hacer funcional la aplicación Web. Solo es necesario instalarlo en el pc servidor para que funcione.

http://jakarta.apache.org/site/downloads/downloads_tomcat-5.cgi

10.1.2- Programas

- **Eclipse:** Entorno de desarrollo java creado por Sun-Microsystem, que proporciona un entorno ideal para integrar en un mismo proyecto una aplicación Web y un sistema Multi-Agente. La utilización de Eclipse es una decisión subjetiva y no influye en el resultado final.

http://www.eclipse.org/platform

- **Tomcat-Plug-in:** Plug-in para integrar el servidor Tomcat en Eclipse

http://www.apache.org

- **Lomboz Plug-in:** Plug-in para Eclipse que permite la creación y compilación de servlets, JSP, y html.

http://lomboz.objectweb.org

- **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/ (API)**

Java Swing

- **http://java.sun.com/docs/books/tutorial/uiswing/components/ (componentes swing)**
- **http://www.tutorialized.com/tutorials/Java/Swing/1 (Tutorial)**
- **http://www.csie.nctu.edu.tw/document/java/tutorial/uiswing/components/spinner.html (tutorial)**

http://www.programacion.com/java/tutorial/swing/36/ (progress bar)
http://www.programacion.com/java/tutorial/swing/32/ (FileChooser)
http://www.programacion.com/java/tutorial/swing/23/ (pestañas)
http://www.programacion.com/java/tutorial/swing/20/ (Jpanel)

JADE

- **http://sharon.cselt.it/projects/jade/community-faq.htm#q20 (preguntas frecuentes)**
- **http://mia.ece.uic.edu/~papers/MediaBot/jade/doc/examples/protocols.html (ejemplos de implementaciones)**

OWL

- **http://owl.man.ac.uk/api.shtml (contiene API)**
- **http://www.w3.org/2004/OWL/ (pagina oficial)**
- **http://www.cs.man.ac.uk/~horrocks/ISWC2003/Tutorial/ (tutorial)**

Protégé

- **http://protege.stanford.edu/ (pagina oficial)**

Applets y programación web.

- **http://www.forosdelweb.com/showthread.php?t=304951 (preguntas frecuentes)**
- **http://www.sc.ehu.es/sbweb/fisica/cursoJava/applets/intro/primero.htm (applets)**
- **http://www.java2s.com/ExampleCode/Swing-JFC/JTextPanedemowithvariousformatandhtmlloadingandrenderering.htm (ejemplos de implementaciones)**
- **http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=firmams (firma applets)**
- **http://www.programacion.com/java/tutorial/servlets_jsp/ (servlet jsp)**

BIBLIOGRAFÍA

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

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

- Grefenstette, G. (1997). SQLET: Palliating One-Word Queries by Providing Intermediate Structure to Text. In: *Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology*, volumen 1299, capitulo 6, 97-114. Springer. SCIE-97. Italy.

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

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

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

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

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

- 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
Estudio de la implementación de agentes en dispositivos móviles

- 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
Provisió de serveis sanitaris a través d'una plataforma d'agents

- Laura Prieto Rebollo, Dr. Antonio Moreno Ribas, Dra. Aïda Valls Mateu
Proyecto Final de Carrera, Escola Técnica Superior d'Enginyeria (ETSE)
Universitat Rovira i Virgili (URV), 2004
SMA para la construccion automática de ontologias

APÉNDICE

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

Instalación Tomcat para utilizar Jsp y Servlets

Para poder utilizar el servidor Tomcat para la creación de servlets o páginas JSP, se tienen que realizar los siguientes pasos:

a) Instalar el servidor Tomcat y alguna versión del J2SDK (nosotros utilizamos Tomcat 5.5.9 y J2SDK 1.5), indicándole el lugar donde se encuentra instalada la JVM (Java Virtual Machine), y de preferencia, dejando como puerto el 8080 para el acceso del servidor

b) Posteriormente se procede a configurarlo en la computadora. Para esto se crea una "variable de entorno". Si se usa Windows 95/98/2000/XP, se crea modificando el archivo autoexec.bat ubicado en c:\ (esta como archivo oculto). Su modificación se hace agregándole la siguiente línea:

set classpath = "ruta"

Donde "ruta" es la ubicación de los archivos jsp-api.jar y servlet-api.jar que se encuentran en la carpeta: ruta_de_instalacion_del_servidor_tomcat/common/lib.

A continuación un ejemplo de línea a agregar al autoexec.bat, suponiendo que el servidor tomcat se instalo en c:\archivos de programa:

SET CLASSPATH= C:\Archivos de programa\Apache Software Foundation\Tomcat 5.0\common\lib\jsp-api.jar;C:\Archivos de programa\Apache Software Foundation\Tomcat 5.0\common\lib\jsp-api.jar

Recordando que también se le agrega al classpath la dirección del la ruta donde se encuentra la carpeta \bin del j2sdk (compilador de java).

c) Una vez echo lo anterior, para agregar el primer servlet al sitio, se tiene que buscar el archivo web.xml, el cual se encuentra en ruta_de_instalacion_del_servidor_tomcat\Apache Software Foundation\Tomcat 5.5\webapps\ROOT\WEB-INF\ y se le agrega a la carpeta webapps el nuevo_servlet.class (que es el resultado de la compilacion del nuevo_servlet.java). Además hay que añadir estas líneas al archivo web.xml

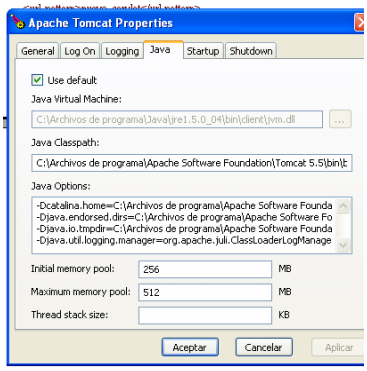
```
<servlet>
   <servlet-name>nuevo_servlet</servlet-name>
   <servlet-class>nuevo_servlet</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>nuevo_servlet</servlet-name>
   <url-pattern>nuevo_servlet</url-pattern>
</servlet-mapping>
```

Pero como el archivo ya tiene escrito, se coloca de tal manera que queda así:

```
<servlet>
   <servlet-name>org.apache.jsp.index_jsp</servlet-name>
   <servlet-class>org.apache.jsp.index_jsp</servlet-class>
</servlet>
<servlet>
   <servlet-name>nuevo_servlet</servlet-name>
   <servlet-class>nuevo_servlet</servlet-class>
</servlet>


<servlet-mapping>
   <servlet-name>org.apache.jsp.index_jsp</servlet-name>
   <url-pattern>/index.jsp</url-pattern>
</servlet-mapping>
<servlet-mapping>
   <servlet-name>nuevo_servlet</servlet-name>
   <url-pattern>nuevo_servlet</url-pattern>
</servlet-mapping>
```

d) Por último se prueba el servlet escribiendo en el explorador http:\\localhost:8080\nuevo_servlet y listo, debería verse el resultado.

Finalmente dadas las características de este proyecto y la cantidad de recursos que utiliza se ha de hacer una última configuración con tal de utilizar más de 100 MB de RAM (Standard de ejecución Java). Abrimos el menú de configuración del tomcat 5.5.9 y en la pestaña Java indicamos en “inicial Memory pool” 256 y en “Maximum memory Pool” 512, como se muestra en la figura.



Esto sería equivalente a compilar todo el proyecto con la opción –Xms256m-Xmx512m.
Aún así hemos tenido muchos problemas para que la aplicación funcione correctamente en el navegador Explorer, en cambio no ha sido así en el resto de navegadores Mozilla Firefox, Nestcape…

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.

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 2 o 3 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 50 y 70 páginas, aunque lo ideal seria mas de 100, pero hay que tener mucho cuidado con los recursos.

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).

Para visualizar cualquier fichero OWL, el usuario simplemente le tiene que dar al botón “Open OWL File” y elegir el que quiera.