UNIVERSIDAD CARLOS III DE MADRID

DEPARTAMENTO DE TELEMÁTICA
PROYECTO FIN DE CARRERA

Estudio del manejo de ontologías para la monitorización de pacientes

Autor: Jose María Bañón Peñuelas

Tutor: Pablo Basanta Val

13/10/2013

Resumen

El uso de ontologías en las tecnologías web cada vez está más extendido. El número de ontologías en los repositorios no para de crecer y el desarrollo hacia la estructura web 3.0 avanza poco a poco. Es por ello que este proyecto prueba el uso una ontología para la monitorización de pacientes. Para ello, se diseña una ontología en OWL mediante un editor de ontologías conocido como Protegé, se realiza una comparación entre dos tecnologías diferentes para el manejo de ontologías como Jena y OWLAPI y se desarrolla una aplicación demostrativa del funcionamiento de la monitorización.

Palabras clave: Jena, OWLAPI, OWL, SPARQL, Hermit, monitorización, Protegé

Abstract

The use of ontologies in web technologies is becoming more widespread. The number of ontologies in the repositories is growing and development of the 3.0 web has made some progress. That is why this project tests an ontology for monitoring patients. To do this, an OWL ontology is designed with an ontology editor known as Protegé, a comparison between two different technologies for the management of ontologies as Jena and OWLAPI is developed and an application to show how the monitoring works, is implemented.

Key words: Jena, OWLAPI, OWL, SPARQL, Hermit, monitoring, Protegé

Agradecimientos

Ha pasado mucho tiempo desde que con 15 años de edad decidí que estudiaría Ingeniería de Telecomunicaciones. Claramente no era consciente del esfuerzo que me iba a suponer, ni de las satisfacciones que me iba a llevar. A pesar de ello mi familia siempre me ha apoyado y ayudado en absolutamente todo lo que he necesitado, en los buenos y en los malos momentos, y son los primeros que se llevan todo mi agradecimiento.

Gracias también a mi tutor, Pablo Basanta, tengo la experiencia de otros compañeros, y os aseguro que ningún profesor ha sido tan rápido y eficiente a la hora de corregir y resolver las dudas como él. Enviar el proyecto y que al día siguiente esté corregido entero, es un lujo por el que muchos de mis amigos hubiesen pagado.

No puedo olvidarme de mis amigos Nano, Lis, Marta, Fer y Oli, siempre preguntando, animando, ofreciendo compañía, asilo bibliotequil, o esa cerveza que hace que retomes fuerzas antes de volver a enfrentarte a los libros. Muchas gracias a vosotros y al resto.

Y por último, pero no por ello menos importante, mis compañeros de clase y amigos, especialmente W.U.S.H, ese grupo formado Jaime, Xavi, Rober, Barco y yo. Tardes eternas en el sótano del Torres, sobremesas infinitas, horas y horas de biblioteca, o esperando esa hamburguesa de la casa de "La buhardilla". Un trozo de este PFC es totalmente vuestro porque estoy seguro de que no estaría aquí de no ser por vosotros.

ÍNDICE DE LA MEMORIA

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1

Introducción y objetivos

1.1 Introducción

El desarrollo de las aplicaciones informáticas junto con el de las nuevas tecnologías en la medicina ha acabado desarrollando un nuevo campo de aplicación conocido como eSalud. Una de las intenciones básicas de eSalud es simplificar las tareas tanto al médico como al paciente. Es por ello que se produce la informatización de todos los datos del paciente lo cual nos conduce a la monitorización de las constantes vitales de los pacientes una vez estén hospitalizados.

Dentro de la monitorización han surgido varias vías de resolución, pero la que aplica en este proyecto es la del manejo de ontologías. Al desarrollar una ontología sobre el paciente, las enfermedades y sus constantes vitales y enunciar unas reglas lógicas que deban cumplirse, se consiguen herramientas de diagnóstico y medicación muy eficientes.

La idea de este proyecto es desarrollar una ontología sencilla, y trabajar de dos maneras distintas con las ontologías, tratándola como base de datos pura, o teniendo en cuenta la lógica implementada de manera que se puedan conseguir conclusiones sobre el comportamiento de ambas.

1.2 Objetivos del proyecto

El objetivo de este proyecto es evaluar el funcionamiento de las ontologías en el campo de la monitorización de los pacientes en tiempo real y comprobar las principales diferencias en eficiencia al tratar las ontologías como base de datos pura o como ontología en sí misma.

Para ello se definen ciertos objetivos parciales:

1. **Desarrollar una ontología y las reglas correspondientes**

   La idea es realizar una ontología capaz de clasificar a los pacientes entre enfermos y sanos a partir de sus constantes vitales básicas.

2. **Manejo de la ontología a través de las APIs OWLAPI y Jena**

   Esto es, desarrollar dos programas (uno para cada API) que consigan agregar y quitar individuos a la ontología así como modificar sus parámetros (constantes vitales)

3. **Inferencia y consultas SPARQL**

   Esto es, conseguir que al realizar consultas a la ontología a través de OWLAPI, gracias al motor de razonamiento sea capaz de inferir el estado de los pacientes, y que al realizar las consultas como base de datos pura a través de Jena también sea capaz de clasificarlos.

4. **Estudio comparativo**

   Esto es, medir el tiempo de clasificación y el uso de CPU para cada uno de los dos métodos y para diferentes tipos de ontologías.

5. **Desarrollo de una interfaz gráfica demostrativa**

   Esto es, la implementación de una interfaz gráfica que muestre la evolución de los enfermos de un hospital y los clasifique en tiempo real.

1.3 Estructura de la memoria

La memoria se ha estructurado de la siguiente manera

- Bloque I: Introducción
  - Capítulo 1: Introducción y objetivos
- Bloque II: Estado del arte

  Análisis de las diferentes tecnologías utilizadas en el proyecto así como el estado actual de las mismas
  - Capítulo 2: Estado del arte
  - Capítulo 3: Tecnologías utilizadas

- Bloque III: Diseño, implementación y validación empírica

  Diseño e implementación de cada uno de los objetivos propuestos, así como comparación de la eficiencia de los dos métodos propuestos en base a pruebas realizadas
  - Capítulo 4: Diseño e implementación de la propuestas
  - Capítulo 5: Evaluación comparativa

- Bloque IV: Conclusiones y futuras líneas de trabajo

  Resultado y conclusiones sobre la consecución del proyecto para indicar el grado de satisfacción de los objetivos
  - Capítulo 6: Conclusiones y líneas futuras de trabajo

- Bloque V Apéndices
- Bloque VI Glosario
- Bloque VII Referencias e hiperenlaces

Capítulo 2

El estado del arte

Este capítulo junto con el Capítulo 3 estudia en profundidad en estado actual de las tecnologías y conceptos que se han utilizado en este proyecto. Además explica los conceptos más importantes y expone proyectos similares que se han llevado a cabo.

2.1 eSalud

El concepto de eSalud se define como la aplicación de las Tecnologías de Información y Comunicación en el amplio rango de aspectos que afectan el cuidado de la salud, desde el diagnóstico hasta el seguimiento de los pacientes, pasando por la gestión de las organizaciones implicadas en estas actividades. En el caso concreto de los ciudadanos, la eSalud les proporciona considerables ventajas en materia de información, incluso favorece la obtención de diagnósticos alternativos. En general, para los profesionales, la eSalud se relaciona con una mejora en el acceso a información relevante, asociada a las principales revistas y asociaciones médicas, con la prescripción electrónica asistida y, finalmente, con la accesibilidad global a los datos médicos personales a través de la Historia Clínica Informatizada.

Es un concepto cuya importancia ha ido en aumento en los últimos años. Y cuyas aportaciones más importantes pueden dividirse en cuatro ramas principales:

- *La vigilancia epidemiológica*: Existen ejemplos en nuestro sistema sanitario sobre posibilidades de predecir resultados de salud (como el tiempo de estancia media, la mortalidad hospitalaria o las variaciones en la expansión de enfermedades) a través de la explotación del Conjunto Mínimo Básico de Datos (CMBD), que tienen que reportar de forma obligatoria todos los hospitales sobre los pacientes que atienden y sobre las variaciones de las epidemias a través de la participación de los usuarios en las redes sociales.
- *Educación de los pacientes de forma online a través de la telemedicina educativa*: Se ha observado, a través del proyecto CARME (CAtalan Remote Management Evaluation) [1], que la telemedicina usada como medio de educación puede reducir un 68% las hospitalizaciones de pacientes con insuficiencia cardiaca. Este estudio se ha aplicado a casos de otras enfermedades crónicas, como la diabetes, ofreciendo resultados similares.

- *La tele-asistencia*: A veces las diferencias de población entre núcleos urbanos y rurales, y la concentración de los especialistas médicos, generalmente en las grandes capitales, dificulta en gran medida el ser atendido para consultas o diagnósticos debido a las distancias que se encuentran estos de algunos pacientes. Aquí es donde entra la telemedicina, donde se han producido avances tecnológicos que permiten una atención médica a distancia más segura y de calidad. En España, se prevé que entre el 70 y el 80 % de los organismos de sanidad, pública o privada, habrán iniciado proyectos de telemedicina en el próximo año (2014).

  La consulta con el médico es una de las posibilidades de la telemedicina, especialmente útil en especialidades donde el contacto físico médico-paciente no es fundamental como la psicología, o la dermatología.

  La utilización de los nuevos sistemas de información y comunicación en las consultas de los pacientes pueden suponer algunas ventajas como:

  · Incremento en la eficiencia de los servicios
  · Agilización de los resultados
  · Ahorro en tiempo de traslado hasta la consulta y de espera en esta.
  · Mejor utilización de los recursos

- *La gestión electrónica de los sistemas y servicios de salud*: A los pacientes les gusta la idea de saber si hay retraso, si mañana viene el médico o no, si está de urgencias, poder tener una especie de como un calendario público. La gestión a través de las redes sociales de los servicios de salud ofrece las siguientes ventajas:

  · Ayuda a cambiar la relación médico-paciente.
  · Facilita accesibilidad a la información.
  · Capacidad de dar información de forma masiva.
  · Prescripción de información fiable.
  · Sirve para ofrecer información sobre el funcionamiento de la consulta.
  · Ahorra tiempo.

2.2 Proyectos eSalud

Existen muchos proyectos relacionados con la eSalud. De entre todos, el más famoso actualmente puede ser el que está relacionado con el nuevo gadget que Google Inc. está desarrollando en este momento: Google Glass [2].

2.2.1 Cirugía asistida por Google Glass

Google Glass [2] es un dispositivo con forma de gafas, que proyecta en un pequeño cristal superior la información que se desee y además es capaz de grabar y retransmitir lo que vemos cuando las tenemos puestas.

El 21/06/2013 el doctor de traumatología Pedro Guillén de la clínica Cemtro de Madrid realizó una intervención quirúrgica con dichas gafas puestas por primera vez en la historia [3]. Dicha operación se retransmitió internacionalmente a un gran número de universidades de medicina que pudieron asistir a la intervención desde el punto de vista del doctor. En esta primera exploración no se le proporcionó información adicional sobre el procedimiento a través de las lentes, pero sí que mantuvo contacto con médicos de la universidad de Stanford que le realizaron preguntas a lo largo de la operación. La potencial ayuda que puede ofrecer este dispositivo durante una operación es realmente enorme ya que podría aportar imágenes de radiografías del paciente, videos de intervenciones similares para la ayuda, o acceso a la información de cualquier tipo que necesite el cirujano en tiempo real.

Figura 1: El doctor Pedro Guillén durante la intervención utilizando las Google Glass [3]

2.2.2 mobiCeliac

En el ámbito de la telemedicina educativa se encuentra este proyecto que ha conseguido una gran acogida en España. MobiCeliac [4] consiste en una aplicación para el teléfono móvil diseñada para los celiacos (alérgicos al gluten) en la que al realizar una foto al código de barras de cualquier alimento, te indica si dicho alimento contiene o no gluten. Esta aplicación se encuentra dentro del Directorio Europeo de Aplicaciones de salud donde podemos encontrar un gran número de aplicaciones de eSalud.

Figura 2: Directorio Europeo de aplicaciones de salud [4]

2.2.3 Hygehos Home

Centrándonos más en el tema principal de este proyecto, la monitorización de los pacientes, aparecen casos como el proyecto Hygehos Home [5]. La Clínica de la Asunción situada en Tolosa y la empresa Bilbomática han elaborado el sistema de monitorización de pacientes a domicilio Hygehos Home que permite, gracias a las nuevas tecnologías, que el médico se ponga en contacto con el paciente y vigilar las constantes vitales o si toma la medicación. El sistema se vale de dispositivos móviles (teléfonos inteligentes y tabletas) para ofrecer un conjunto de servicios adaptados a las características concretas de cada paciente. De este modo, el médico puede hacer el seguimiento online de la patología, tomar las constantes vitales del paciente (peso, tensión, glucosa en sangre, frecuencia cardiaca), saber si toma la medicación y tener contacto directo con el personal sanitario cuando lo necesite. La Clínica de la Asunción está realizando una prueba piloto con doce personas con diferentes patologías crónicas. En este sentido, los sanitarios reciben todas las notificaciones de los pacientes y pueden actuar en consecuencia, sin abandonar el entorno de la historia clínica. Este sistema está actualmente en fase de validación sanitaria. De este estudio se extraerá información valiosa que contribuirá a la mejora del sistema, que se implantará en todas las especialidades de la clínica a finales de año o comienzos del años próximo. Los responsables de Hygehos Home han asegurado que contribuirá a mejorar la calidad de vida y la atención sanitaria que se proporciona a los pacientes de la clínica, ya que se reducirán desplazamientos innecesarios y se evitarán las

recaídas y descompensaciones de los pacientes. Además, fomentará una mejor gestión de los recursos. Este sistema de monitorización se encuentra enmarcado dentro del proyecto Kronos y está financiado parcialmente por el Gobierno Vasco a través de la convocatoria Etorgai 2011.

2.2.4 Sistema iOSC3

El proyecto más similar al desarrollado en este documento se llama iOSC3 [6]. Es un sistema de monitorización de los enfermos ingresados en la UCI (Unidad de Cuidados Intensivos) del hospital Meixoeiro de la ciudad de Vigo que se basa en ontologías y en función de las reglas de inteligencia artificial desarrolladas y de los valores de los principales indicadores de salud del enfermo (presión arterial, frecuencia cardiaca, temperatura, frecuencia respiratoria y nivel de oxígeno en sangre) propone una medicación a ser administrada automáticamente siempre bajo la supervisión de un médico.

El funcionamiento del sistema se puede ver en la Figura 3.

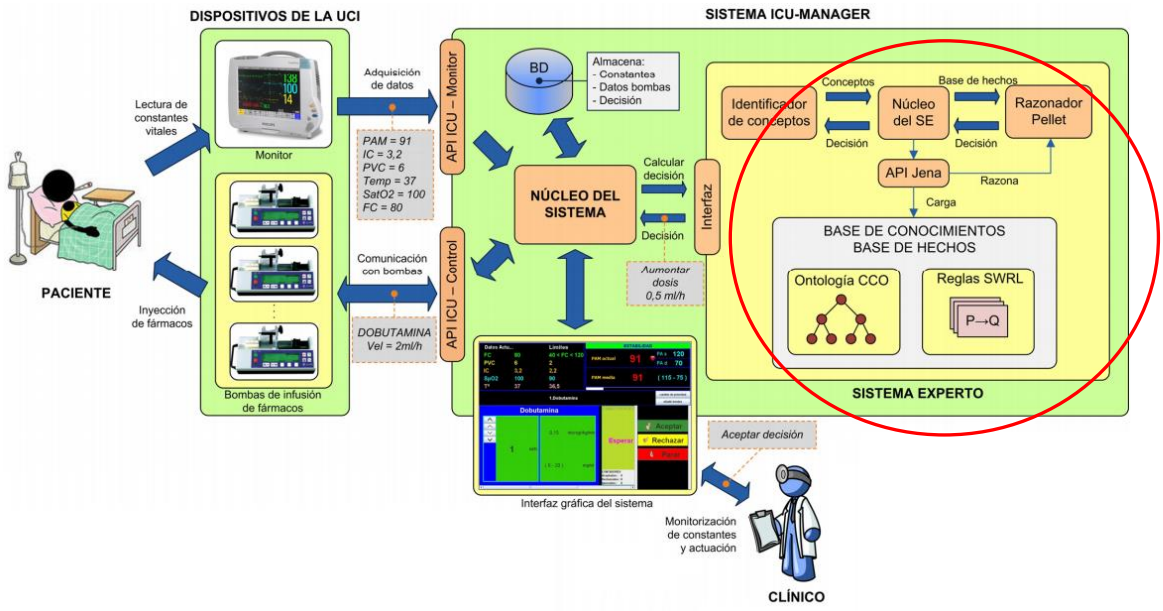

Figura 3: Estructura del sistema iOSC [6]

Este proyecto al igual que este, se basa en ontologías para comprobar el estado de los pacientes y la conexión con las ontologías la realiza a través del API de Jena y un razonador externo (Pellet). iOSC3 comenzó en 2011 y su diseño principal se ha visto truncado debido a que Jena ha cambiado su estructura y ha dejado de ofrecer soporte externo para razonadores (aunque pretende volver a ofrecerlo más adelante), por lo que se vio obligado a cambiar un poco su estructura y evitar el uso del API de Jena que no le permitía el aprovechar las principales ventajas de la inteligencia artificial asociada a las ontologías.

Figura 4: iOSC3 instalado en una sala de la UCI del Hospital Meixoeiro [6]

2.3 Introducción a la web semántica

Para entender una ontología hay que entender de dónde vienen y como se usan, y las ontologías se usan principalmente en el entorno de la web semántica.

La web semántica es un campo creciente donde confluyen la inteligencia artificial y las tecnologías web. Es lo que se conoce como la evolución de la Web 2.0 a la Web 3.0. La idea principal es conseguir a través de descripciones explícitas sobre el significado de los recursos que las máquinas alcancen cierto grado de comprensión de la web para conseguir que la misma máquina se encargue de realizar una parte del trabajo que los usuarios de la red realizan manualmente.

Frente a la semántica implícita, el crecimiento caótico de recursos y la ausencia de una organización clara de la web actual, la web semántica opta por dar una estructura, clasificar y anotar los recursos con semántica explícita procesable por máquinas. Además, la web semántica mantiene lo principios de éxito de la web actual como la descentralización, compartición, compatibilidad, facilidad de acceso y contribución o la apertura al crecimiento y uso no previstos de antemano.

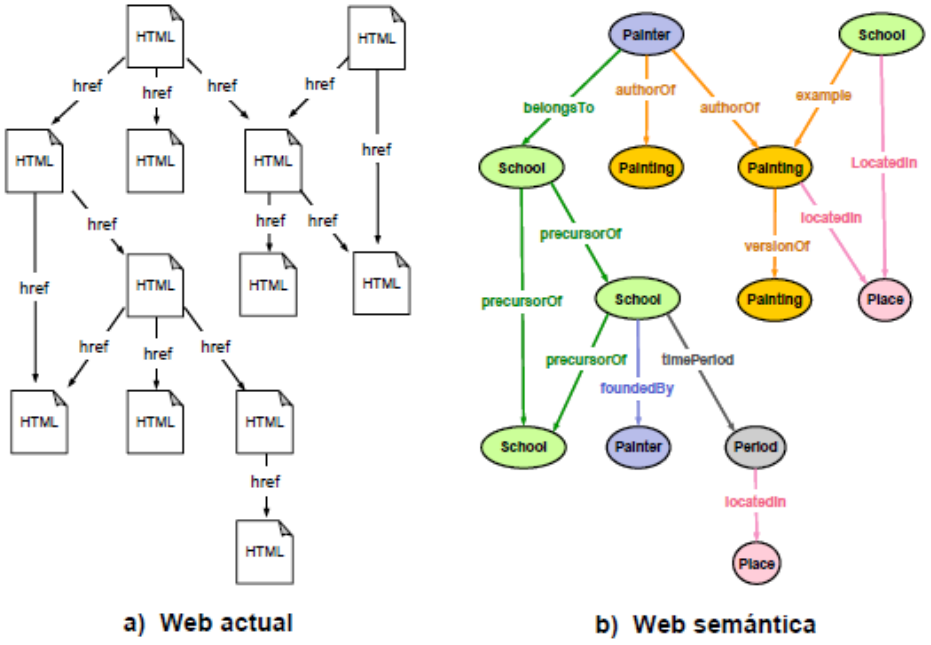

Figura 5: Estructura de la web actual y de la web semántica [7]

En la Figura 5 podemos apreciar las principales diferencias entre la web actual y la web semántica. La web tal como la conocemos actualmente, es un grafo formado por nodos del mismo tipo y arcos (enlaces) igualmente indiferenciados. No se puede diferenciar, por ejemplo, entre la página de la universidad y la de un periódico cualquiera. Por otro lado en la web semántica los nodos tienen tipo y los arcos representan relaciones diferenciadas.

El problema clave a la hora de desarrollar la web semántica es alcanzar un entendimiento entre las partes que intervienen en la construcción y explotación de la web: usuarios, desarrolladores y programas de diversos perfiles. Para solucionarlo, toma prestada la noción de ontología del campo de la inteligencia artificial.

2.4 Concepto de ontología

Una ontología es una jerarquía de conceptos con atributos y relaciones, que define una terminología consensuada para crear redes semánticas de unidades de información interrelacionadas. Las ontologías ofrecen un vocabulario de clases y relaciones que sirven para describir un dominio, poniendo énfasis en la compartición del conocimiento y el consenso en la representación de éste. Por ejemplo una ontología sobre música debería incluir las clases Artista, Grupo, Canción, Estilo Musical, Casa Discográfica, y las relaciones como autor de la canción, grupos pertenecientes a un estilo musical o canciones publicadas por una casa discográfica.

La idea principal sería conseguir a partir de la web semántica una red de nodos tipificados e interconectados a través de una ontología compartida por los distintos autores. De esta manera

conseguiríamos (siguiendo el ejemplo anterior) juntar todas las canciones de todas las casas discográficas manteniendo una misma estructura y facilitando la posible integración de todas ellas.

Aparte del acceso a contenidos, la web semántica también tiene la capacidad de ofrecer servicios como comprar una entrada, reservar mesa en restaurantes o simular el precio de una hipoteca. Para ello se definen ontologías de funcionalidad y procedimientos para describir servicios web. Estas descripciones son procesables por máquinas y permiten automatizar la ejecución de servicios así como la comunicación entre unos y otros.

Se puede observar la estructura de una ontología relativamente sencilla y ver como todos sus nodos y arcos están tipificados y jerarquizados (ver Figura 6):

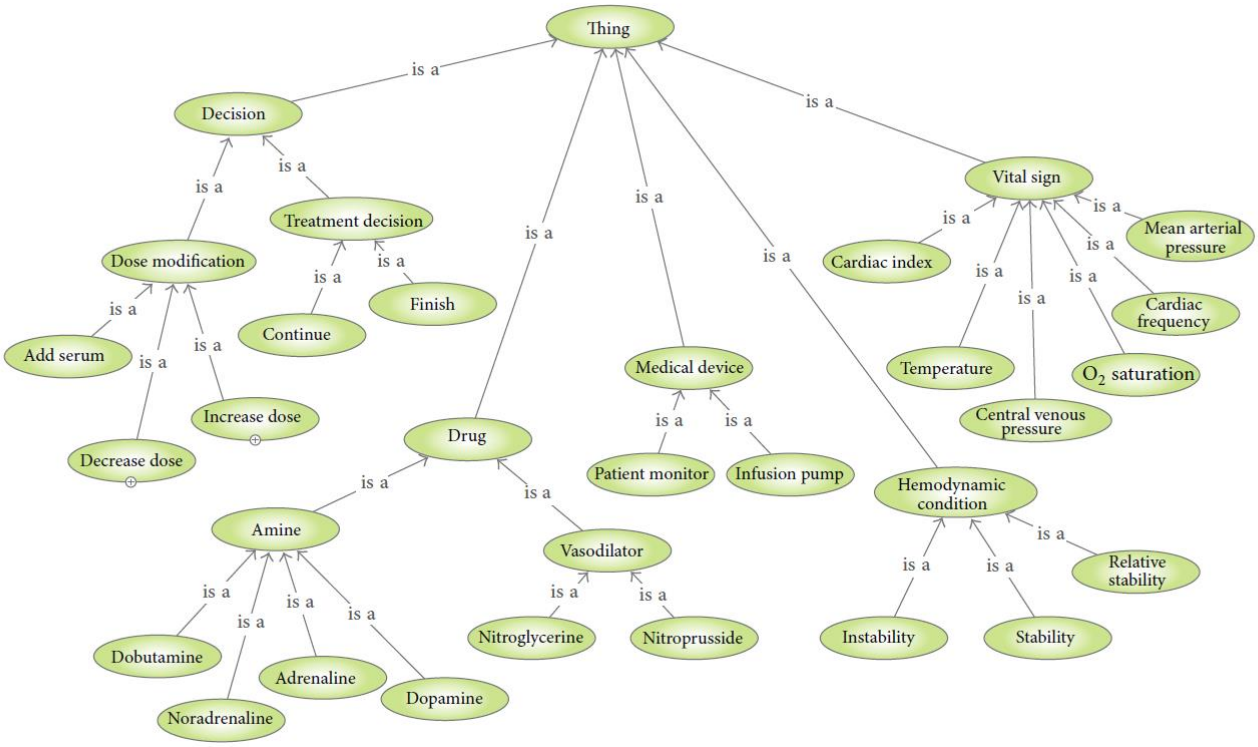

Figura 6: Ontología desarrollada para el sistema iOSC3 [6]

En la estructura de la web semántica es muy variada y compleja y en ella se superponen multitud de lenguajes y protocolos como podemos apreciar en la Figura 7.

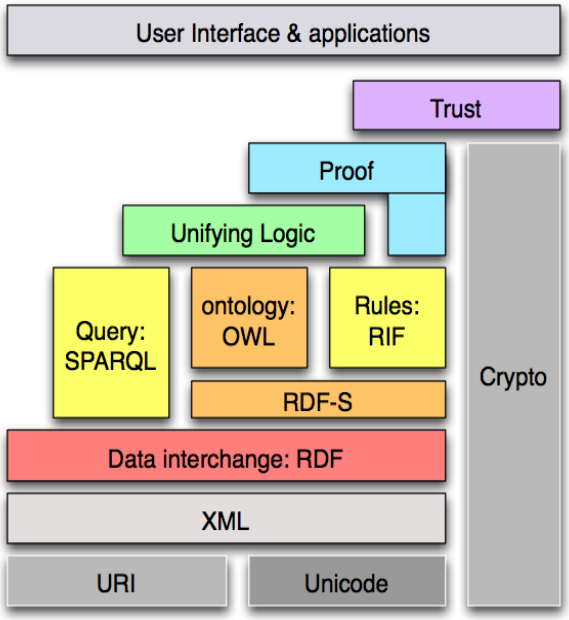

Figura 7: Estructura de la web semántica [8]

Todos los elementos (los recursos y las propiedades) de las ontologías tendrán una URI (Uniform Resource Identifier) que es una cadena de caracteres que referencia a cada uno de estos objetos. Son las conocidas URL de las páginas web que todo el mundo conoce. Por ejemplo se programa una ontología para pacientes, la URI del recurso paciente puede ser www.ontologiapacientes.com/#Paciente y la URI de la propiedad "frecuencia cardiaca" del paciente sería www.ontologiapacientes.com/#frecCardiaca, en resumen, las URIS son los identificadores de cada uno de los elementos de la ontología. Para que se pueda utilizar en la web semántica la ontología irá codificada sobre código XML lo cual simplifica las cosas ya que se reutiliza el lenguaje de almacenamiento de datos en forma legible recomendado por la W3C. En función de la complejidad de la ontología a desarrollar se puede implementar en RDF [9], RDFS [9] u OWL [11] que son los lenguajes para ontologías estudiados en el capítulo 3 de esta memoria. Paralelamente a ellos se puede usar SPARQL [12] que es un lenguaje de consulta de bases de datos para preguntar cualquier sobre cualquier dato a la ontología.

El resto de módulos que se encuentran por encima de OWL y SPARQL pertenecen a la inteligencia artificial de la web semántica. Ahí es donde entran en juego los motores de inferencia como el Hermit [13] que será utilizado en este proyecto.

En la Figura 8 se puede apreciar la manera general en la que se trabaja con las ontologías para la web semántica

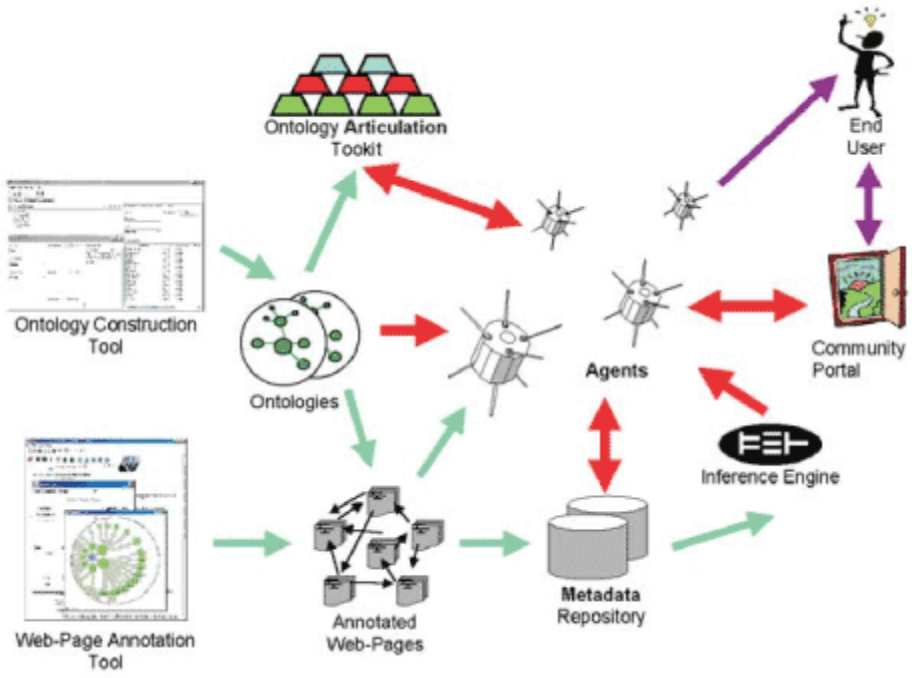

Figura 8: Uso de las ontologías en el entorno de la web semántica [8]

En primer lugar para desarrollar la ontología utiliza un entorno de desarrollo de ontologías (Ontology Construction Tool). El más utilizado es el entorno Protegé [14] del que se exponen las características en el capítulo 3 de esta memoria. Para más información sobre cómo proceder con el diseño de ontologías se puede consultar el Anexo B de esta memoria.

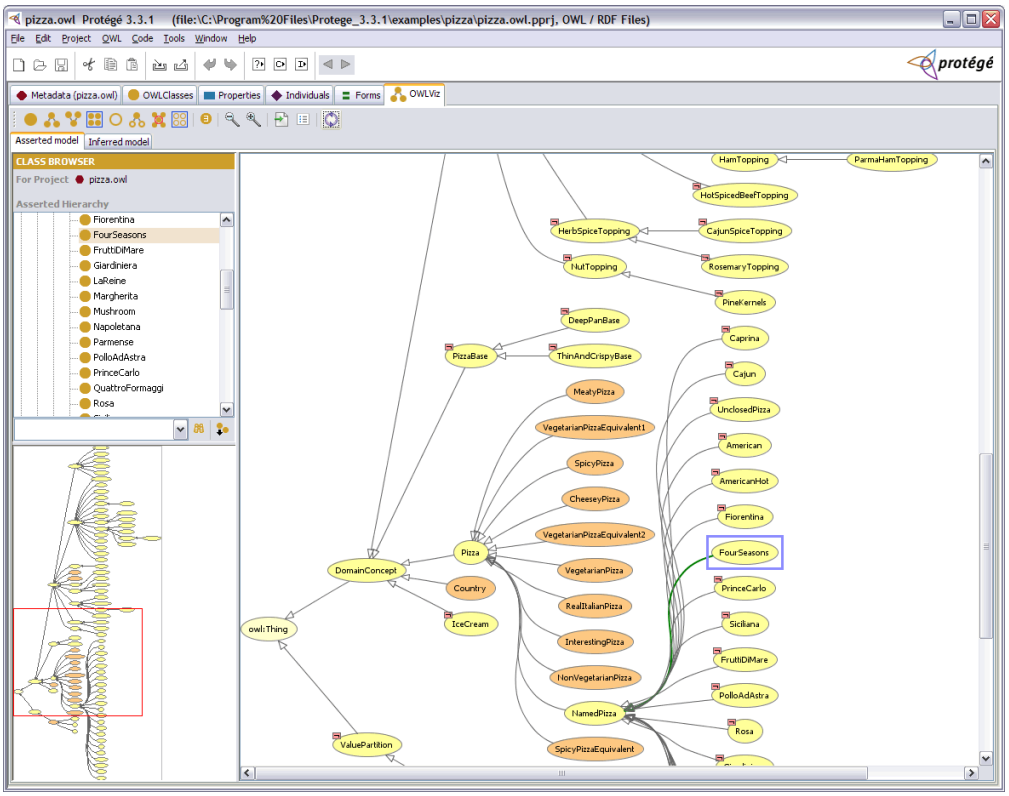

Figura 9: Entorno de desarrollo de ontologías Protegé [7]

Una vez la ontología está diseñada necesitamos poder trabajar con ella, poder preguntarle cosas, agregarle propiedades o añadir individuos, para eso se desarrollan los agentes de ontologías. En este proyecto se han utilizado dos agentes para ontologías, los dos funcionando sobre Java: Jena y OWLAPI que harán uso del Ontology Articulation Toolkit realizarán las modificaciones necesarias. Para poder aprovecharse de las reglas de la ontología y poder inferir conocimiento, se utilizan motores de razonamiento externos (Hermit en este caso).

Finalmente las ontologías se suelen subir a los repositorios para que cualquier usuario las pueda tener disponibles y que permitir que su aplicación se actualice al actualizar la ontología sin necesidad de actualizar el resto del programa.

Capítulo 3

Tecnologías utilizadas

En este capítulo se explicarán las tecnologías utilizadas para el desarrollo de este proyecto, se empezará por las más básicas como Java y Jswing que se ha utilizado para el desarrollo de la interfaz gráfica de la aplicación desarrollada. Después se pasa a desarrollar las propiedades de los lenguajes ontológicos RDF y OWL, se sigue explicando los agentes para ontologías que trabajan sobre Java: Jena [15] y OWLAPI [16] así como el consultor para la ontología SPARQL, y se termina explicando las funcionalidades principales del entorno de desarrollo de ontologías Protegé.

3.1 Java

Java es un lenguaje de programación orientado a objetos qu**e** fue originalmente desarrollado por James Gosling de Sun Microsystems y publicado en el 1995 [17]. Su sintaxis deriva mucho de C y C++, pero tiene menos facilidades de bajo nivel que cualquiera de ellos.

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar el paradigma de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
3. Debería incluir por defecto soporte para trabajo en red.
4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

3.1.1 Orientación a objetos

La primera característica, orientado a objetos ("OO"), se refiere a un método de programación y al diseño del lenguaje. Aunque hay muchas interpretaciones para OO, una primera idea es diseñar el software de forma que los distintos tipos de datos que usen estén unidos a sus operaciones. Así, los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos. Un objeto puede verse como un paquete que contiene el "comportamiento" (el código) y el "estado" (datos). El principio es separar aquello que cambia de las cosas que permanecen inalterables. Frecuentemente, cambiar una estructura de datos

implica un cambio en el código que opera sobre los mismos, o viceversa. Esta separación en objetos coherentes e independientes ofrece una base más estable para el diseño de un sistema software. El objetivo es hacer que grandes proyectos sean fáciles de gestionar y manejar, mejorando como consecuencia su calidad y reduciendo el número de proyectos fallidos. Otra de las grandes promesas de la programación orientada a objetos es la creación de entidades más genéricas (objetos) que permitan la reutilización del software entre proyectos, una de las premisas fundamentales de la Ingeniería del Software. Un objeto genérico "cliente", por ejemplo, debería en teoría tener el mismo conjunto de comportamiento en diferentes proyectos, sobre todo cuando estos coinciden en cierta medida, algo que suele suceder en las grandes organizaciones. En este sentido, los objetos podrían verse como piezas reutilizables que pueden emplearse en múltiples proyectos distintos, posibilitando así a la industria del software a construir proyectos de envergadura empleando componentes ya existentes y de comprobada calidad; conduciendo esto finalmente a una reducción drástica del tiempo de desarrollo. La reutilización del software ha experimentado resultados dispares, encontrando dos dificultades principales: el diseño de objetos realmente genéricos es pobremente comprendido, y falta una metodología para la amplia comunicación de oportunidades de reutilización. Algunas comunidades de "código abierto" (open source) quieren ayudar en este problema dando medios a los desarrolladores para diseminar la información sobre el uso y versatilidad de objetos reutilizables y bibliotecas de objetos.

3.1.2 Independencia de la plataforma

La segunda característica, la independencia de la plataforma, significa que programas escritos en el lenguaje Java pueden ejecutarse igualmente en cualquier tipo de hardware. Este es el significado de ser capaz de escribir un programa una vez y que pueda ejecutarse en cualquier dispositivo, tal como reza el axioma de Java, "*write once, run everywhere*".

Para ello, se compila el código fuente escrito en lenguaje Java, para generar un código conocido como "bytecode" (específicamente Java bytecode) —instrucciones máquina simplificadas específicas de la plataforma Java. Esta pieza está "a medio camino" entre el código fuente y el código máquina que entiende el dispositivo destino. El bytecode es ejecutado entonces en la máquina virtual (JVM), un programa escrito en código nativo de la plataforma destino (que es el que entiende su hardware), que interpreta y ejecuta el código. Además, se suministran bibliotecas adicionales para acceder a las características de cada dispositivo (como los gráficos, ejecución mediante hilos o threads, la interfaz de red) de forma unificada. Se debe tener presente que, aunque hay una etapa explícita de compilación, el bytecode generado es interpretado o convertido a instrucciones máquina del código nativo por el compilador JIT (Just In Time). Todo este proceso se ilustra en la Figura 10.

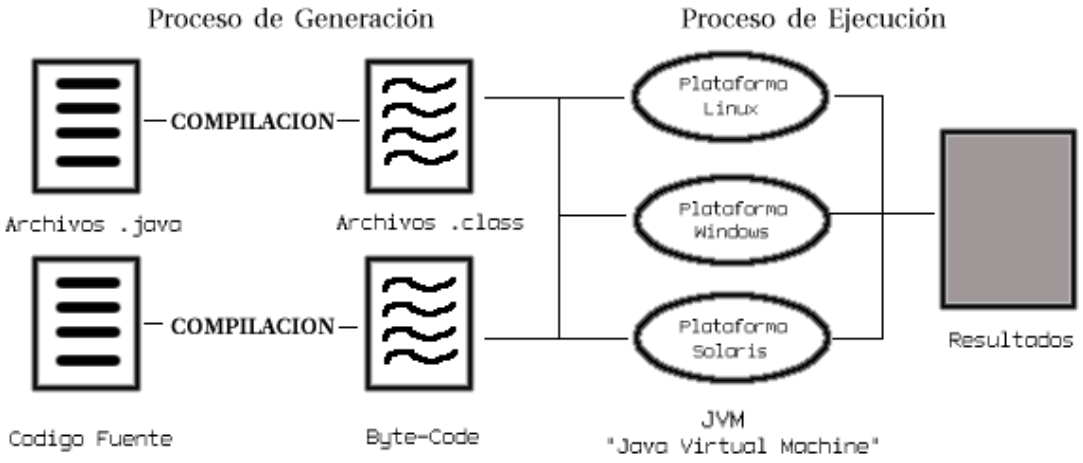

Figura 10: Proceso de compilación en Java [17]

Hay implementaciones del compilador de Java que convierten el código fuente directamente en código objeto nativo, como GCJ. Esto elimina la etapa intermedia donde se genera el bytecode, pero la salida de este tipo de compiladores sólo puede ejecutarse en un tipo de arquitectura.

La portabilidad es técnicamente difícil de lograr, y el éxito de Java en ese campo ha sido dispar. Aunque es de hecho posible escribir programas para la plataforma Java que actúen de forma correcta en múltiples plataformas de distinta arquitectura, se producen un gran número de estas con pequeños errores o inconsistencias. El concepto de independencia de la plataforma de Java cuenta, sin embargo, con un gran éxito en las aplicaciones en el entorno del servidor, como los Servicios Web, los Servlets y los Java Beans.

3.1.3 El recolector de basura

En Java el problema fugas de memoria se evita en gran medida gracias a la recolección de basura (o *automatic garbage collector*). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos. Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aun así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios—es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos y mayor seguridad.

3.2 JSwing

El paquete Swing [18] es un paquete gráfico que apareció en la versión 1.2 de Java como parte de la JFC, Java Foundation Classes. Está formado por un extenso conjunto de componentes de interfaces de usuario abarcando entre ellas botones, tablas y marcos.

Swing existe desde la JDK 1.1 aunque antes de la existencia del paquete swing, las interfaces gráficas se realizaban a través de AWT (Abstract Window Toolkit), de quien Swing hereda todo el manejo de eventos. Es por ello que, normalmente, para toda componente AWT existe una componente Swing que la reemplaza, por ejemplo, la clase Button de AWT es reemplazada por la clase JButton de Swing. Las componentes de Swing utilizan la infraestructura de AWT, incluyendo el modelo de eventos AWT, el cual se rige como una componente que reacciona a eventos tales como, eventos de ratón o eventos de teclado. Es por esto, que la mayoría de los programas Swing necesitan importar los paquetes de AWT *java.awt* y *java.awt.event*. Las componentes propios de Swing se identifican porque pertenecen al paquete *javax.swing*.

La arquitectura Swing presenta una serie de ventajas respecto a su antecedente AWT:

- Amplia variedad de componentes.
- Aspecto modificable (*look and feel*): Se puede personalizar el aspecto de las interfaces o utilizar varios aspectos que existen por defecto (Metal Max, Basic Motif, Window Win32).
- Arquitectura Modelo-Vista-Controlador: Esta arquitectura da lugar a todo un enfoque de desarrollo muy arraigado en los entornos gráficos de usuario realizados con técnicas orientadas a objetos. Cada componente tiene asociado una clase de modelo de datos y una interfaz que utiliza. Se puede crear un modelo de datos personalizado para cada componente, con sólo heredar de la clase *Model*.
- Gestión mejorada de la entrada del usuario: Se pueden gestionar combinaciones de teclas en un objeto k*eyStrok* y registrarlo como componente. El evento se activará cuando se pulse dicha combinación si está siendo utilizado el componente, la ventana en que se encuentra o algún hijo del componente.
- Objetos de acción (*action objects*): Estos objetos cuando están activados (*enabled*) controlan las acciones de varios objetos componentes de la interfaz. Son hijos de *ActionListener*.
- Contenedores anidados: Cualquier componente puede estar anidado en otro. Por ejemplo, un gráfico se puede anidar en una lista.
- Escritorios virtuales: Se pueden crear escritorios virtuales o "interfaz de múltiples documentos" mediante las clases *JdesktopPane* y *JInternalFrame*.

- Bordes complejos: Los componentes pueden presentar nuevos tipos de bordes. Además el usuario puede crear tipos de bordes personalizados.
- Componentes para tablas y árboles de datos: Mediante las clases *Jtable* y *JTree*.
- Potentes manipuladores de texto: Además de campos y áreas de texto, se presentan campos de sintaxis oculta *JPassword*, y texto con múltiples fuentes *JTextPane*. Además hay paquetes para utilizar ficheros en formato HTML o RTF.
- Capacidad para "deshacer": En gran variedad de situaciones se pueden deshacer las modificaciones que se realizaron.

Swing incorpora su nuevo conjunto de eventos para sus componentes. Los más importantes a destacar son los siguientes:

- *AncestorEvent*: Antecesor añadido desplazado o eliminado.
- *CaretEvent*: El signo de intercalación del texto ha cambiado.
- *ChangeEvent*: Un componente ha sufrido un cambio de estado.
- *DocumentEvent*: Un documento ha sufrido un cambio de estado.
- *HyperlinkEvent*: Algo relacionado con un vínculo hipermedia ha cambiado.
- *ListDataEvent*: El contenido de una lista ha cambiado o se ha añadido o eliminado un intervalo.
- *ListSelectionEvent*: La selección de una lista ha cambiado.
- *MenuEvent*: Un elemento de menú ha sido seleccionado o mostrado o bien no seleccionado o cancelado.
- P*opupMenuEvent*: Algo ha cambiado en un *JPopupMenu*.
- *TableColumnModelEvent*: El modelo para una columna de tabla ha cambiado.
- *TableModelEvent*: El modelo de una tabla ha cambiado.
- *TreeExpansionEvent*: El nodo de un árbol se ha extendido o se ha colapsado.
- *TreeModelEvent*: El modelo de un árbol ha cambiado.
- *TreeSelectionEvent*: La selección de un árbol ha cambiado de estado.

3.3 RDF

RDF (Resource Description Framework) [9] fue creado en 1999, y es un lenguaje explícitamente desarrollado para la definición de ontologías y metadatos en la web. Hoy en día es el estándar más popular y extendido en la comunidad de la web semántica. Dicha popularidad radica en su elemento principal, lo que se conoce como el “triple”, tripleta o sentencia. Este elemento consiste en dos nodos (sujeto y objeto) unidos por un arco (predicado). Los nodos representan recursos y los arcos propiedades. Con RDF Schema (RDFS), se pueden definir

jerarquías de clases de recursos especificando las propiedades y relaciones que se admiten entre ellas. En RDF las clases, relaciones y las propias sentencias son también recursos, y por lo tanto se pueden examinar y recorrer como parte del grafo.

En la Figura 121 podemos comprobar lo que es una estructura de triples formando una jerarquía (que viene dada por "subclassOf"), donde las elipses son los recursos y los arcos los predicados.

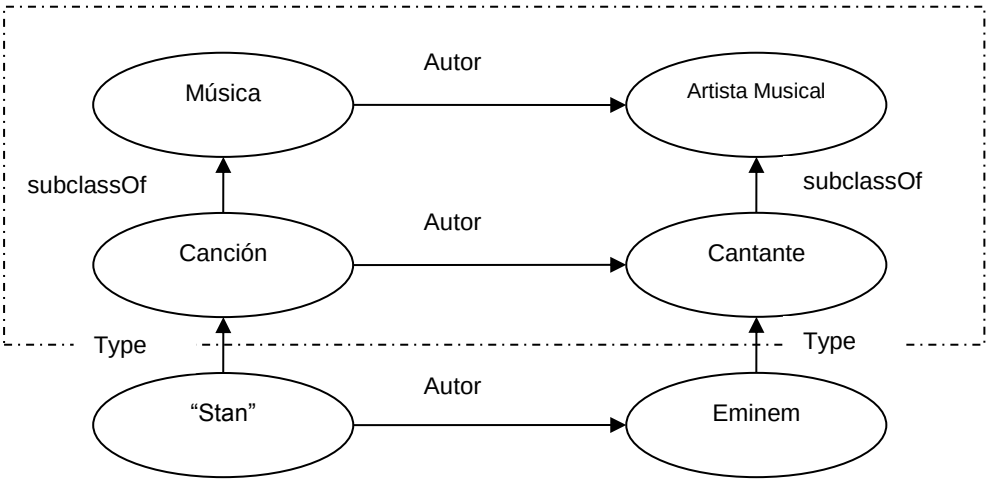

Figura 11: Representación de nodos RDF

Una herramienta muy útil a la hora de poder expresar búsquedas complejas a través del grafo RDF es RDF QueryLanguage (RDFQL). Usando una estructura muy similar al conocido SQL es capaz de atravesar el árbol y devolver los valores deseados de una forma muy eficiente. RDQL permite encontrar desde todas las canciones de un artista, las colaboraciones que tiene con otros artistas o incluso los discos que sacó hasta determinado año. RDF es en sí un lenguaje bastante potente para definir ontologías, pero le detalles para ser más completo como la posibilidad de utilizar expresiones lógica sal definir clases que son subsanados por OWL, su evolución lógica.

3.4 OWL

OWL como ya se ha comentado anteriormente facilita un mejor mecanismo de interpretabilidad de contenido web que RDF y RDFS proporcionando vocabulario adicional y la lógica necesaria para poder desarrollar una ontología completa. Consta de tres sublenguajes con un nivel de expresividad creciente: OWL Lite [11], OWL DL [11] y OWL Full [11].

3.4.1 Sublenguajes de OWL

OWL Lite [11] está diseñado para los usuarios que necesiten una clasificación jerárquica y restricciones simples. Por ejemplo, admite restricciones de cardinalidad, pero solo

permite establecer valores cardinales de 0 o 1. Está diseñado de esta manera para que sea más sencillo proporcionar herramientas de soporte a OWL Lite que a sus OWL DL y OWL Full que contarán con mayor nivel de expresividad.

OWL DL [11] está diseñado para aquellos usuarios que quieren la máxima expresividad conservando completitud computacional (se garantiza que todas las conclusiones sean computables), y resolubilidad (limitación de los cálculos en tiempo). OWL DL incluye todas las construcciones del lenguaje de OWL, pero sólo pueden ser usados bajo ciertas restricciones (por ejemplo, mientras una clase puede ser una subclase de otras muchas clases, una clase no puede ser una instancia de otra). OWL DL es denominado de esta forma debido a su correspondencia con la lógica de descripción (Description Logics, en inglés), un campo de investigación que estudia la lógica que compone la base formal de OWL.

OWL Full [11] está dirigido a usuarios que quieren máxima expresividad y libertad sintáctica de RDF sin garantías computacionales. Por ejemplo, en OWL Full una clase puede ser considerada simultáneamente como una colección de clases individuales y como una clase individual propiamente dicha. OWL Full permite una ontología para aumentar el significado del vocabulario preestablecido (RDF u OWL). Es poco probable que cualquier software de razonamiento sea capaz de obtener un razonamiento completo para cada característica de OWL Full aunque es cierto que motores como Hermit han mostrado un buen comportamiento.

Cada uno de estos sublenguajes es una extensión de su predecesor más simple, respecto a lo que puede ser expresado legalmente y a la validación de sus conclusiones. El siguiente grupo de relaciones se mantienen, pero las relaciones inversas no se mantienen.

- Cada ontología legal de OWL Lite es una ontología legal de OWL DL
- Cada ontología legal de OWL DL es una ontología legal de OWL Full
- Cada conclusión válida de OWL Lite es una conclusión válida de OWL DL
- Cada conclusión válida de OWL DL es una conclusión válida de OWL Full

Los desarrolladores de ontologías que adoptan OWL deberían considerar cuál es el sublenguaje que mejor se adapta a sus necesidades. La elección entre OWL Lite y OWL DL depende de las necesidades de los usuarios sobre la expresividad de las construcciones, proporcionando OWL DL las más expresivas. La elección entre OWL DL y OWL Full depende principalmente de las necesidades de los usuarios sobre los recursos de metamodelado del esquema RDF (por ejemplo, definir clases de clases, o definir propiedades de clases). Cuando se usa OWL Full en comparación con OWL DL, el soporte en el razonamiento es menos predecible, ya que no existen en este momento implementaciones completas de OWL Full.

OWL Full puede ser considerada como una extensión de RDF, mientras que OWL Lite y OWL DL pueden ser consideradas como extensiones de una versión restringida de RDF.

3.4.2 Características generales de OWL Lite

A la hora de definir una ontología en OWL Lite [11], existen multitud de campos que se pueden rellenar, muchos de ellos con un significado muy parecido pero diferenciarlos correctamente será muy útil a la hora de que una máquina sea capaz de interpretarlos. Los campos que incluyen *rdf* o *rdfs* delante son campos incluidos dentro de dichos lenguajes y totalmente legales e interpretables en OWL.

Class: Una clase define un grupo de individuos que pertenecen a la misma porque comparten algunas propiedades. Por ejemplo, Ana y Juan son miembros de la clase Persona. Las clases pueden organizarse en una jerarquía de especialización usando subClassOf. Se puede encontrar una clase general llamada Thing que es una clase de todos los individuos y es una superclase de todas las clases de OWL. También se puede encontrar una clase general llamada Nothing que es la clase que no tiene instancias y es una subclase de todas las clases de OWL.

rdfs: subClassOf: Las jerarquías de clase deben crearse haciendo una o más indicaciones de que una clase es subclase de otra. Por ejemplo, la clase Persona podría estar definida como subclase de la clase Mamífero. De esta manera cualquier máquina con un motor lógico adecuado será capaz de deducir que si un individuo es una Persona, entonces, también es un Mamífero.

rdf: Property: las propiedades pueden utilizarse para establecer relaciones entre individuos o de individuos a valores de datos. Ejemplos de propiedades pueden ser tieneHijo, tieneFamiliar, tieneHermano, y tieneEdad. Los tres primeros pueden utilizarse para relacionar una instancia de la clase Persona con otra instancia de la clase Persona (siendo casos de ObjectProperty), y el último (tieneEdad) puede ser usado para relacionar una instancia de la clase Persona con una instancia del tipo de datos Entero (siendo un caso de DatatypeProperty). Ambas, owl:ObjectProperty y owl:DatatypeProperty, son subclases de la clase de RDF rdf:Property.

rdfs: subPropertyOf: Las jerarquías de propiedades pueden crearse haciendo una o más indicaciones de que una propiedad es a su vez subpropiedad de una o más propiedades. Por ejemplo, tieneHermano puede ser una subpropiedad de tieneFamiliar. De esta forma, un motor de razonamiento puede deducir que si un individuo está relacionado con otro por la propiedad tieneHermano, entonces está también relacionado con ese otro por la propiedad tieneFamiliar.

rdfs: domain: Un dominio de propiedad reduce los individuos a los que puede aplicarse la propiedad. Si una propiedad relaciona un individuo con otro individuo, y la propiedad tiene una clase como uno de sus dominios, entonces el individuo debe pertenecer a esa clase. Por ejemplo, puede establecerse que la propiedad tieneHijo tenga como dominio la clase Mamífero. De esto,

un razonador puede deducir que si “Juan tieneHijo Paco”, entonces Juan debe ser de la clase Mamífero. Obsérvese que rdfs: domain se denomina restricción global debido a que la restricción se refiere a la propiedad y no sólo a la propiedad cuando está asociada con una clase en concreto.

rdfs: range: El rango de una propiedad reduce los individuos que una propiedad puede tener como su valor. Si una propiedad relaciona a un individuo con otro individuo, y ésta como rango a una clase, entonces el segundo individuo debe pertenecer a dicha clase. Por ejemplo, la propiedad tieneHija debe establecerse que tiene como rango la clase Mamífero. A partir de aquí, un razonador puede deducir que si Ana está relacionada con Alicia mediante la propiedad tieneHija, (por ejemplo, Ana tieneHija Alicia), entonces Ana es un Mamífero. El rango es, al igual que el dominio, una restricción global.

Individual: Los individuos son instancias de clases, y las propiedades pueden ser usadas para relacionar un individuo con otro. Por ejemplo, un individuo llamado Pablo puede ser descrito como una instancia de la clase Persona y la propiedad tieneTrabajoEn puede ser usada para relacionar el individuo Pablo con el individuo UniversidadCarlosIII.

3.4.3 Características lógicas de OWL

Las siguientes características de OWL Lite [11] describen las relaciones de igualdad en las ontologías.

equivalentClass: Es posible definir dos clases como equivalentes. Las clases equivalentes tienen las mismas instancias. El valor de igualdad puede ser utilizado para crear clases sinónimas. Por ejemplo, Coche puede definirse como equivalentClass (clase equivalente) de Automóvil. De esta manera, un razonador puede deducir que cualquier individuo que sea instancia de Coche, es también una instancia de Automóvil y viceversa.

equivalentProperty: Se pueden definir dos propiedades como equivalentes. Las propiedades equivalentes relacionan a un individuo con el conjunto de otros individuos similares. Por ejemplo, tieneLíder debe definirse como equivalentProperty (propiedad equivalente) de tienePresidente. De este modo, un razonador puede deducir que si X está relacionado con Y por la propiedad tieneLíder, X está también relacionado con Y por la propiedad tienePresidente y viceversa. Un razonador también puede deducir que tieneLíder es una subpropiedad de tienePresidente, y tienePresidente es un subpropiedad de tieneLíder.

sameAs: Es posible definir dos individuos como iguales. Estas construcciones pueden utilizarse para crear un número de nombres diferentes que se refieren al mismo individuo. Por ejemplo, el individuo Pablo puede establecerse como un individuo igual a PabloGutierrez.

differentFrom: Es posible definir un individuo como diferente de otros individuos. Por ejemplo, el individuo Juan puede establecerse como distinto de los individuos Ana y Pablo. Por tanto, si los dos individuos, Juan y Ana, constituyen valores para una propiedad que se ha establecido como funcional (de este modo la propiedad tiene como máximo un único valor), entonces hay una contradicción. Establecer explícitamente que los individuos son diferentes entre sí puede ser importante al utilizar lenguajes como OWL (y RDF) que no asumen que los individuos tienen un único nombre exclusivamente. Por ejemplo**, sin información adicional, un razonador no puede deducir que Juan y Ana hagan referencia a individuos distintos.**

AllDifferent: Se puede definir un número de individuos como mutuamente distintos mediante una indicación AllDifferent. Por ejemplo, se podría establecer que Juan, Ana y Pablo son mutuamente distintos usando la construcción AllDifferent. Al contrario que la indicación differentFrom anterior, esto además resaltaría que Ana y Pablo son distintos (no únicamente que Juan es distinto de Ana y que también es distinto de Pablo). La construcción AllDifferent es particularmente útil cuando hay conjuntos de objetos distintos, y cuando los diseñadores están interesados en reforzar la suposición de nombres únicos dentro de estos conjuntos de objetos. Se usa junto con distinctMembers para establecer que todos los miembros de una lista son distintos, y disjuntos en pares.

3.4.4 Características de las propiedades en OWL Lite

Además de los identificadores ObjectProperty y DatatypeProperty vistos en la sección 3.4.2 existen identificadores especiales en OWL Lite que se utilizan para proporcionar la información referente a las propiedades y a sus valores:

inverseOf: Es posible definir una propiedad como la inversa de otra propiedad. Si se estableciera la propiedad P1 como inversa de la propiedad P2, y relacionáramos X con Y mediante la propiedad P2, entonces Y estaría relacionado con X mediante la propiedad P1. Por ejemplo, si la propiedad tieneHijo es la propiedad opuesta de tienePadres y se enuncia en OWL que Juan tienePadres Pablo, entonces un razonador podrá deducir que Pablo tieneHijo Juan.

TransitiveProperty: Es posible definir propiedades como transitivas. Una propiedad es transitiva si el par (x, y) es una instancia de la propiedad transitiva P, y el par (y, z) es otra instancia de la propiedad transitiva P, entonces el par (x, z) también es una instancia de P. Por ejemplo, si se indica que la propiedad Antepasado es transitiva, si Pablo Antepasado de Juan es una instancia de la propiedad Antepasado) y si Ana Antepasado de Pablo entonces un razonador puede deducir que Ana es un antepasado de Juan.

SymmetricProperty: Es posible definir propiedades como simétricas. Si una propiedad es simétrica, y el par (x, y) es una instancia de esa propiedad simétrica P, entonces el par (y, x) es también una instancia de la propiedad simétrica P. Por ejemplo, Amigo puede considerarse una

propiedad simétrica. De esta forma, si se indica a un razonador que Pablo es amigo de Juan, éste puede deducir que Juan es amigo de Pablo.

FunctionalProperty: Es posible definir propiedades para que tengan un valor único. Si una propiedad es FunctionalProperty, ésta no tendrá más de un valor para cada individuo (es posible que no tenga un valor para algún individuo). Esta característica se denomina propiedad única. FunctionalProperty es una forma abreviada para indicar que la cardinalidad mínima de la propiedad es 0 y la cardinalidad máxima es 1. Por ejemplo, tieneJefe puede establecerse como FunctionalProperty. Es por ello que un razonador puede deducir que ningún individuo pueda tener más de un jefe principal. Sin embargo, esto no implica que todas las instancias de Persona deban tener al menos un jefe principal.

InverseFunctionalProperty: Es posible definir propiedades para que sean funcional inversa. Si una propiedad es funcional inversa, entonces la inversa de la propiedad será funcional. Por tanto, la inversa de la propiedad tiene como máximo de un valor para cada individuo. Esta característica también se denomina propiedad inequívoca. Por ejemplo, tieneDNI puede establecerse como funcional inversa (o inequívoca). La inversa de esta propiedad (esDNIde) tiene como máximo un valor para cada individuo dentro de la clase de los números de DNI. De esta manera, el número de DNIde cualquier una persona es el único valor para su propiedad esDNIde. Es por esto que un motor de razonamiento puede deducir que no existen dos individuos diferentes, instancias de Persona, que tengan idéntico el número de DNI. Además, un razonador puede deducir que si dos instancias de Persona tienen el mismo número de DNI, es porque dichas instancias se refieren al mismo individuo.

3.4.5 Restricciones de propiedad en OWL Lite

OWL Lite permite establecer restricciones sobre la forma en que las propiedades son utilizadas por las instancias de una clase. Las dos restricciones que siguen a continuación indican los valores que pueden ser utilizados por una propiedad:

allValuesFrom: La restricción allValuesFrom se establece sobre una propiedad con respecto a una clase. Esto significa que esta propiedad sobre una determinada clase tiene una restricción de rango local asociada a ella. De este modo, si una instancia de la clase está relacionada con un segundo individuo mediante esa propiedad, entonces puede deducirse que el segundo individuo es una instancia de una clase de la restricción de rango local. Por ejemplo, la clase Persona puede tener una propiedad denominado tieneHija restringida a tener allValuesFrom de la clase Mujer. Esto significa que si un individuo Ana está relacionado mediante la propiedad tieneHija con un individuo Alicia, entonces un razonador puede deducir que Alicia es una instancia de la clase Mujer. Esta restricción permite que la propiedad tieneHija sea utilizada por otras clases, como puede ser la clase Gato, y tener una restricción de valor apropiada asociada con el uso de

la propiedad sobre esa clase. En este caso, tieneHija podría tener la restricción de rango local Gato cuando se asocie con la clase Gato y la restricción de rango local Persona cuando se asocie a la clase Persona. Obsérvese que un razonador no puede deducir únicamente de la restricción AllValuesFrom que realmente existe al menos un valor para la propiedad.

someValuesFrom: La restricción someValuesFrom también se establece sobre una propiedad con respecto a una clase. Una clase particular puede tener una restricción sobre una propiedad que haga que al menos un valor para esa propiedad sea de un tipo concreto. Por ejemplo, la clase ArticuloSobreWebSemántica puede tener una restricción de dicho tipo sobre la propiedad tienePalabraClave que indica que algunos valores de la propiedad tienePalabraClave pueden ser instancias de la clase TemaSobreWebSemántica. Esto permite la opción de tener múltiples palabras claves y, siempre que una o más sean instancias de la clase TemaSobreWebSemántica, el papel será consistente con la restricción someValuesFrom. A diferencia de allValuesFrom, someValuesFrom no restringe que todos los valores de una propiedad sean instancias de una misma clase. Si miArticulo es una instancia de la clase ArticuloSobreWebSemántica, entonces miArticulo está relacionada por la propiedad tienePalabraClave con al menos una instancia de la clase TemaSobreWebSemántica. Obsérvese que un razonador no puede deducir, como podría hacer con las restricciones allValuesFrom, que todos los valores de tienePalabraClave son instancias de la clase TemaSobreWebSemántica.

3.4.6 Restricciones de cardinalidad de OWL Lite

OWL Lite incluye además restricciones de cardinalidad. Las restricciones de cardinalidad de OWL (y OWL Lite) se refieren a restricciones locales, puesto que se definen en las propiedades con respecto a una clase particular. Es decir, las restricciones fuerzan la cardinalidad de esa propiedad sobre instancias de esa clase. Las restricciones de cardinalidad de OWL Lite son limitadas porque permiten solamente realizar indicaciones referentes a cardinalidades de valor 0 o 1 (no está permitido añadir valores arbitrarios para la cardinalidad, como es el caso en OWL DL y OWL Full).

minCardinality: La cardinalidad se establece sobre una propiedad con respecto a una clase particular. Si se establece minCardinality (cardinalidad mínima) de 1 sobre una propiedad con respecto a una clase, entonces cualquier instancia de esa clase estará relacionada al menos con un individuo mediante esta propiedad. Esta restricción es otra manera de decir que es necesario que la propiedad tenga un valor para todas las instancias de la clase.

maxCardinality: Esta cardinalidad se establece sobre una propiedad con respecto a una clase particular. Si se establece maxCardinality (cardinalidad máxima) de 1 sobre una propiedad con respecto a una clase, entonces cualquier instancia de esa clase estará relacionada como máximo con un individuo mediante dicha propiedad.

cardinality: La cardinalidad se presenta como una ventaja cuando es útil establecer que una propiedad tiene sobre una clase minCardinality 0 y maxCardinality 0, o ambos minCardinality 1 y maxCardinality 1. Por ejemplo, la clase Persona tiene exactamente un único valor para la propiedad tieneMadreBiológica. De esta manera, un razonador puede deducir que dos instancias distintas de la clase Madre no pueden ser valores para la propiedad tieneMadreBiológica de la misma persona.

3.4.7 Características de OWL DL y OWL Full

OWL DL y OWL Full utilizan el mismo vocabulario aunque OWL DL está sujeto a algunas restricciones. De forma general, OWL DL requiere separación de tipos (una clase no puede ser un individuo o una propiedad, una propiedad no puede ser tampoco un individuo o una clase). Esto implica que no se pueden aplicar restricciones a elementos del lenguaje de OWL (algo que se permite en OWL Full). A continuación se describe el vocabulario de OWL DL y OWL Full, que extiende las construcciones de OWL Lite:

oneOf: (Clases enumeradas): Las clases se pueden describir mediante la enumeración de los individuos que la componen. Los miembros de la clase son exactamente el grupo de los individuos enumerados. Por ejemplo, la clase díasDeLaSemana puede describirse simplemente enumerando los individuos Lunes, Martes, Miércoles, Jueves, Viernes, Sábado y Domingo. De esta forma, un razonador puede deducir la cardinalidad máxima (7) de cualquier propiedad que tenga díasDeLaSemana como restricción de allValuesFrom.

hasValue: (Valores de la propiedad): Una propiedad puede ser necesaria que tenga un determinado individuo como un valor. Por ejemplo, instancias de la clase ciudadanosHolandeses pueden ser caracterizadas como las personas que tiene PaísesBajos como valor de su nacionalidad.

disjointWith: Es posible establecer que las clases sean disjuntas unas de otras. Por ejemplo, Hombre y Mujer pueden definirse como clases disjuntas. A partir de esta declaración de disjointWith, un razonador puede deducir una inconsistencia cuando un individuo se indica como instancia de ambas clases y de igual manera un razonador puede deducir que si Paco es una instancia de Hombre, entonces Paco no es una instancia de Mujer.

unionOf, complementOf, intersectionOf (Combinaciones booleanas): OWL DL y OWL Full permiten combinaciones booleanas arbitrarias de clases y de restricciones: unionOf, complementOf, e intersectionOf. Por ejemplo, usando unionOf, podemos indicar que la clase CiudadanosPeninsulaIberica contiene elementos que son ciudadanosDePortugal o ciudadanosDeEspaña. Usando complementOf, podríamos indicar que los niños no son PersonasMayores es decir, la clase Niños es una subclase de complemento de

PersonasMayores. La ciudadanía de la Unión Europea se podía describir como la unión de las ciudadanías de todos los países miembros.

minCardinality, maxCardinality, cardinality (Cardinalidad completa): Mientras que en OWL Lite, las cardinalidades se ciñen a como mínimo, como máximo o exactamente 1 o 0, OWL Full permite realizar declaraciones de cardinalidad para números enteros no negativos arbitrarios. Por ejemplo la clase de MatrimonioconDobleSueldoSinHijos restringiría la cardinalidad de la propiedad tieneSueldo a una cardinalidad mínima de 2 (mientras que la propiedad tieneHijo tendría que ser restringida a cardinalidad 0).

3.5 Herramientas para OWL

Aparte del hecho de que OWL es una recomendación del W3C, uno de las probables razones de su éxito y relativa amplia adopción en el mundo académico y la industria es que hay una amplia gama de herramientas OWL disponibles. Estas herramientas ofrecen apoyo a la creación y edición de ontologías OWL, el razonamiento sobre ontologías y el uso de las mismas en aplicaciones. En términos generales, todas estas herramientas requieren algún tipo de API subyacente que permita cargar, manipular y consultar ontologías. Con respecto al razonamiento, las aplicaciones de tipo cliente requieren generalmente del uso de interfaces generales del razonador para que puedan crear con facilidad diferentes configuraciones. Por el oro lado, los desarrolladores de motores de razonamiento pueden beneficiarse de la posibilidad de proporcionar una interfaz genérica y sencilla para sus razonadores.

3.5.1 OWLAPI

OWLAPI [19] ha sido diseñado para satisfacer las necesidades de personas que desarrollan aplicaciones basadas en OWL, así como editores y razonadores de OWL. Se trata de una API de Java de alto nivel que está estrechamente alineada con la especificación OWL 2. Incluye soporte para cargar modificar y crear ontologías, validadores para los distintos perfiles de OWL 2, así como interfaces generales para el uso de razonadores. Este API también tiene un diseño muy flexible que permite a terceras partes proporcionar implementaciones alternativas para cada uno de los componentes principales.

3.5.1.1 Diseño de OWLAPI

En esencia, OWLAPI consiste en un conjunto de interfaces para la inspección, la manipulación y el razonamiento de ontologías OWL. Dicho API soporta la carga y el guardado de ontologías en una gran variedad de sintaxis. Sin embargo, ninguno de los interfaces de modelo en el API están relacionados a ninguna sintaxis en particular. Por ejemplo, a diferencia de otras API como Jena, la representación de las expresiones de clases y axiomas no se realiza a través de las tripletas RDF. De hecho, el diseño de la API OWL se basa directamente en la especificación estructural de OWL 2. Esto significa que una ontología es simplemente visto

como un conjunto de axiomas y anotaciones como se muestra en la Figura 11. Los nombres y jerarquías de interfaces para las instancias, clases y axiomas de OWLAPI corresponden a la especificación estructural antes nombrada. Gracias a esta relación tan semejante al lenguaje ontológico es fácil obtener el alto nivel de OWL 2 con los métodos e interfaces de OWLAPI.

3.5.1.2 Gestión de Ontología

Además de las interfaces para OWL que representan clases y axiomas, es necesario tener cierta maquinaria que permita a las aplicaciones poder manejar las ontologías. La Figura 12 muestra una visión general de alto nivel de esta idea. La interfaz OWLOntology proporciona un punto de acceso de manera eficiente a los axiomas contenidos en una ontología. Por ejemplo, a los axiomas se puede acceder por tipo o por nombre entre otras opciones. OWLOntologyManager es el gestor de las ontologías que proporciona un punto central para crear, cargar, modificar y guardar ontologías, que son instancias de la interfaz OWLOntology.

Cada ontología se crea o se carga por un gestor de la ontología. Cada instancia de una ontología es exclusiva de un gestor en particular, y todos los cambios en una ontología se aplican a través de ese mismo gestor. Este diseño centralizado de la gestión permite a las aplicaciones de tipo cliente tener un punto de acceso a las ontologías, proporcionar mecanismos de redirección y otras personalizaciones en la carga de ontologías. El gestor también esconde mucha de la complejidad asociada a la elección de los *parsers* y *renderers* adecuados para cargar y guardar ontologías y por tanto evita a los usuarios de OWLAPI este tipo de problemas.

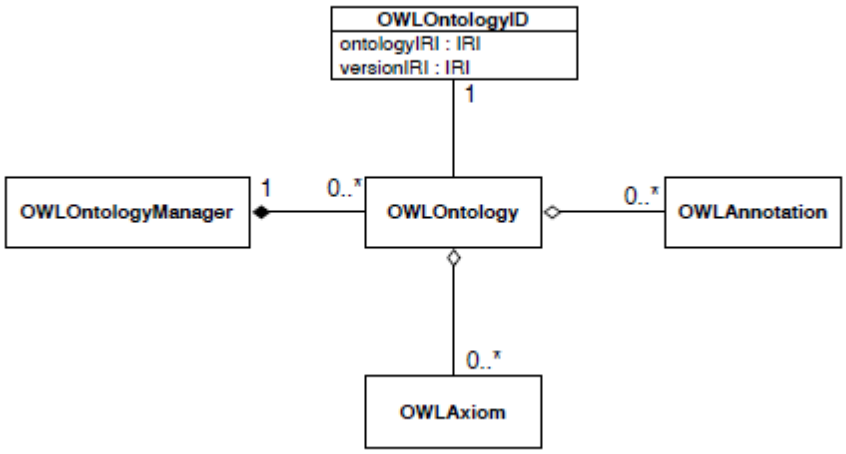

Figura 12: Diagrama UML del manejo de ontologías en OWLAPI [19]

3.5.1.3 Cambio de Ontología

Los cambios en las ontologías se aplican utilizando objetos que implementan la interfaz OWLOntologyChange. Además de añadir y eliminar los axiomas, los cambios que se pueden realizar con OWLAPI incluyen a establecer el ID de una ontología, añadir y eliminar

anotaciones, y añadir y eliminar las importaciones. Todo esto hace a OWLAPI ideal para el uso en los editores y otros sistemas que necesitan la capacidad de registrar cambios.
Todos los cambios de ontologías se aplican a través de un gestor de la ontología. Esto significa que es posible aplicar una lista de cambios de ontologías, que hacen múltiples cambios en múltiples ontologías, como una sola unidad. Esto funciona bastante bien para aplicaciones tales como editores de ontologías, donde una operación de edición tal como cambiar el nombre de una entidad (cambio de IRI) puede implicar la adición y la eliminación de varios axiomas a partir de múltiples ontologías-es posible agrupar los cambios juntos para formar una "operación del editor" y aplicar estos cambios a la vez.

3.5.1.4 Parsers y renders de OWL

Una ventaja importante de la alineación de OWLAPI [19] y la especificación estructural de OWL es que no hay compromiso con una sintaxis concreta particular. Mientras sólo hay una sintaxis con la que implementación OWL debe ser compatible (RDF/XML), existen varias otras sintaxis que están optimizados para diferentes propósitos. Por ejemplo, la sintaxis Turtle ofrece un poco más legibilidad a la serialización RDF. Del mismo modo, la sintaxis Manchester OWL ofrece un serialización legible para el ser humano para ontologías OWL.
OWL/XML es un nuevo diseño de sintaxis que permite a las ontologías ser almacenadas en XML "normal" y que a diferencia de RDF/XML, puede ser utilizada directamente por herramientas y tecnologías como XPath XML.
Por tanto, OWLAPI incluye además soporte para la lectura y escritura de ontologías en varias sintaxis, incluyendo RDF/XML, Turtle[9], OWL/XML, sintaxis funcional de OWL, sintaxis Manchester, KRSS y la formato de archivo plano OBO. Debido al diseño subyacente de la API, es posible que al terminar la importación de una ontología esta, contenga ontologías que provengan de diferentes sintaxis. La implementación de referencia de la API de OWL utiliza un registro de los parsers y renderers, lo que hace que sea fácil de añadir soporte para sintaxis personalizadas. El analizador apropiado se selecciona automáticamente en tiempo de ejecución cuando se carga una ontología. Por defecto, las ontologías se guardan de nuevo en el formato del que fueron analizadas, pero es posible reemplazar esto con el fin de realizar las tareas de conversión de sintaxis y "guardar como" la sintaxis deseada.

3.5.1.5 Estructura de datos de almacenamiento

La implementación de referencia proporciona las estructuras de datos para una representación de ontologías eficiente en uso de memoria. Para muchos propósitos, con esto es suficiente. Por ejemplo, las versiones más recientes del Instituto Nacional del Cáncer (INC) [19], una ontología puede cómodamente encajar en torno a 400 MB de memoria (100 MB de

memoria sin anotaciones). Sin embargo, la API ha sido diseñada de manera que sea posible proporcionar otro mecanismo de almacenamiento para las ontologías. Por ejemplo, se conocen casos de uso para almacenamiento de ontologías en bases de datos relacionales o almacenes de triples. OWLAPI ha sido diseñado con estos tipos de casos de uso en mente, y es posible mezclar diferentes implementaciones de almacenamiento, de modo que al importar una ontología, podría contener ontologías en memoria, ontologías en almacenamiento secundario en forma de una base de datos personalizada y ontologías en almacenes de triples.
Mientras que la API no incluye estos mecanismos de almacenamiento alternativos de por sí, desarrolladores externos, han desarrollado este tipo de soluciones. Una solución ejemplar, llamada OWLDB, ha sido desarrollada por Henss [21]. Su solución almacena ontologías como una base de datos relacional, utilizando un mapeo de "nativo" de OWL (en oposición a un mapeo usando triples) consigue construir un esquema de base de datos personalizada.

3.5.1.6 Interfaces para razonadores

Una parte clave de trabajar con ontologías OWL es el razonamiento. Se utilizan razonadores para comprobar la coherencia de las ontologías, para comprobar si la una ontología contiene clases ilógicas, para comprobar que las jerarquías de propiedad y de clases son correctas y para comprobar si los axiomas se aplicaron correctamente en una ontología. OWLAPI tiene varios interfaces para apoyar la interacción con razonadores OWL. La interfaz principal es la OWLReasoner interfaz que proporciona métodos para llevar a cabo la mencionada tareas. Las interfaces para el razonador se han diseñado con el fin de facilitar su uso y de forma que los usuarios puedan ayudarse de su razonamiento progresivo.
La API permite que un razonador se configure de forma que detecta cambios en la ontología y ya sea inmediatamente procesa los cambios o los organiza en un búfer que después puedan ser procesados. Este diseño es apto para el escenario en el que se utiliza un razonador dentro de un editor de ontologías y, en algún momento, debe responder a las ediciones de la ontología (como será el caso de este proyecto), o situaciones en las que un razonador debe responder a los cambios de la ontología tal y como llegan.
En el momento de escribir esto, Fact++ [22], Pellet [22] y Hermit [13] son los razonadores que ofrecen soporte para OWLAPI siendo este último el más utilizado. Esto significa que también están disponibles como plugins para el editor de ontologías Protegé.

3.5.2 Jena

Jena[9] es un framework de software libre, desarrollado por HP Labs, para el desarrollo de aplicaciones Java relacionadas con la Web Semántica. Entre las características de Jena cabe destacar que permite gestionar todo tipo de ontologías, almacenarlas y realizar consultas contra

ellas. Soporta distintos lenguajes como RDF [9], DAML [24] y OWL [16], siendo independiente del lenguaje. Actualmente está disponible Jena 2.11 que incluye los siguientes componentes:

- API para RDF (Resource Description Framework).
- API de ontologías con soporte para OWL, DAML y RDF Schema.
- Lectura y escritura RDF en formato RDF/XML, N3 y N-Triples
- Motor de consultas SPARQL (ARQ) [26].
- Almacenamiento en memoria y almacenamiento persistente.
- Subsistema de razonamiento.

Su estructura es bastante completa consiguiendo abarcar una gran parte de lo que es la web semántica. Jena consta de cinco módulos diferentes como se puede comprobar en la Figura 13:

- Inference API o API de Inferencia: utilizado para inferir nuevo conocimiento sobre representaciones RDF.
- RDF API: para manipular los RDF graph (conjunto de triples RDF).
- Ontology API: para manejar ontologías representadas en RDF y OWL.
- Store API: que se encarga del almacenamiento de las ontologías.
- SPARQL API: para tratar a las ontologías como bases de datos y poder realizar consultas.

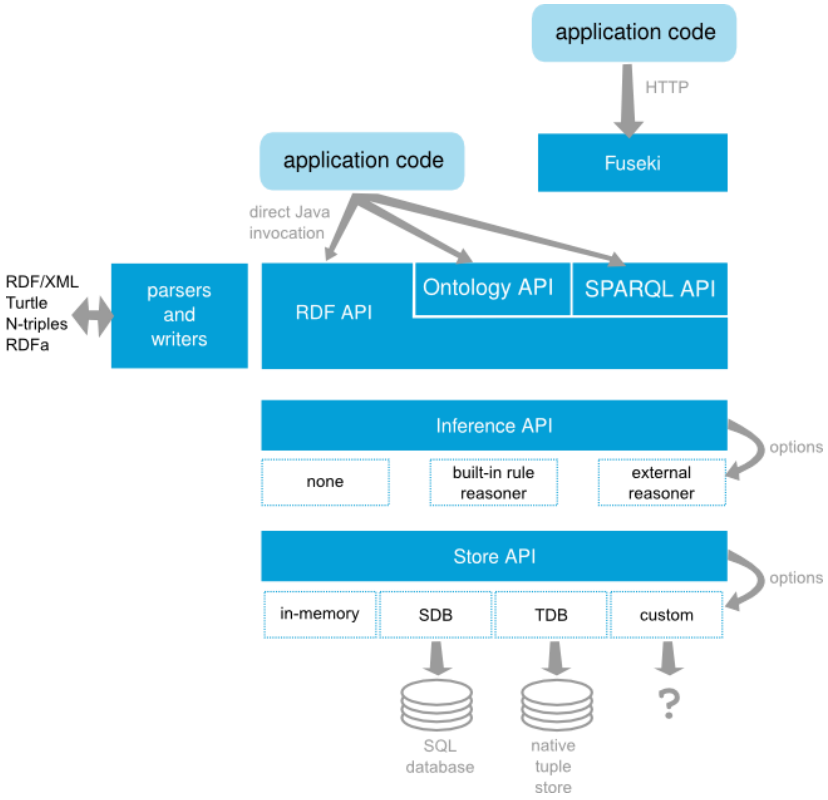

Figura 13: Estructura de Jena [25]

3.5.2.1 API para RDF

Permite crear y manipular modelos RDF desde una aplicación Java. La utilización de estos modelos es aceptada por la W3C. RDF es un lenguaje semejante a XML que permite la creación de información semántica utilizando la estructura XML. Define una estructura para implementar la semántica de la información identificando objetos y sus propiedades. Provee primitivas para utilizar expresiones que poseen (sujeto, predicado, objeto).

Otro aspecto a tener en cuenta es cómo el framework trabaja la manera del polimorfismo en Java. Jena resuelve el tema de los distintos niveles polimórficos de RDF y la abstracción de clases en Java (OntClass, Restriction, DataTypeProperty y ObjectProperty) considerando a estas abstracciones como una vista del recurso. Es decir, existe una relación uno a muchos, un recurso puede ser representado por varias vistas.

3.5.2.2 API para el almacenamiento

El almacenamiento en memoria es la forma clásica con la que suelen trabajar habitualmente las aplicaciones. En el caso de aplicaciones que involucran el manejo de ontologías, equivale a tener el modelo OWL (ontología) almacenado en memoria principal, como si se tratase de una variable de programa. Una forma de trabajar con ontologías en Jena es declarando una variable OntModel e ir construyendo el modelo en la propia aplicación (creación de clases, subclases, propiedades y restricciones). Con Jena también es posible cargar un modelo a partir de una fuente local, por ejemplo, un fichero que contenga una ontología definida en el lenguaje de marcado OWL.

El almacenamiento persistente surge ante la necesidad de guardar datos de programa de forma duradera para recuperarlos en otro momento. El término "persistencia" es sinónimo de "durabilidad" y "permanencia". Obviamente, el almacenamiento persistente en bases de datos supone grandes ventajas sobre el almacenamiento en memoria y el almacenamiento tradicional en el sistema de ficheros. Para ejecutar operaciones sobre bases de datos en una aplicación Java, es necesario usar el API de almacenamiento (Store API). Este API es una colección de interfaces Java y métodos de gestión de manejadores de conexión para cada modelo específico de base de datos. En este ámbito, Jena es capaz de trabajar con ontologías almacenadas de forma persistente en una base de datos.

3.5.2.3 Inferencia en Jena

Inferir consiste en deducir información adicional a partir de la existente. El código que realiza la tarea de inferir se le llama razonador (Reasoner). Jena incluye un conjunto básico de razonadores:

- RDFS Reasoner: que soporta RDF y RDFS

- OWL Reasoner: para realizar la inferencia en OWL aunque es incompleta.
- Transitive Reasoner: que es un razonador configurable.

Consultando la página web de Jena se observa en su estructura que se incluye la posibilidad de incluir motores de razonamiento externos. Sin embargo esto todavía no es cierto, es tan sólo una intención que muestran los desarrolladores de Jena.

3.6. SPARQL

3.6.1 Introducción

SPARQL (Simple Protocol and RDF Query Language) [12], es un lenguaje de recuperación para RDFs. Se trata de una recomendación del W3C que los estandarizó en 2008, para crear un lenguaje de consulta dentro de la Web semántica que está ya implementada en muchos lenguajes de programación y bases de datos como por ejemplo en Java donde se puede encontrar el API de SPARQL perteneciente a Jena. Este lenguaje de consulta permite centrarse en la información que se desea tener recopilada, sin tener en cuenta la tecnología de la base de datos o el formato utilizado para almacenar esos datos. Debido a que las consultas en el lenguaje SPARQL expresan objetivos de alto nivel, es fácil extenderlos a orígenes de datos inesperados, o incluso transferirlos a nuevas aplicaciones. Además, SPARQL ha sido diseñado para un uso a escala de la Web, así permite hacer consultas sobre orígenes de datos distribuidos, independientemente del formato. A la hora de recuperar información la creación de una sola consulta a través de diferentes almacenes es mejor que múltiples consultas.

SPARQL consiste en tres especificaciones complementarias, que contienen diferentes partes de su funcionalidad. SPARQL lo forman un lenguaje de consultas, un formato para las respuestas, y un medio para el transporte de consultas y respuestas:

- SPARQL Query Language [27]: Núcleo de SPARQL. Explica la sintaxis para la composición de sentencias y su concordancia.
- SPARQL Protocol [27]: Formato utilizado para la devolución de los resultados de las búsquedas (instrucciones SELECT o ASK), a partir de un esquema de XML.
- SPARQL Query XML Results Format [27]: Describe el acceso remoto de datos y la transmisión de consultas de los clientes a los procesadores. Utiliza WSDL para definir protocolos remotos para la consulta de bases de datos basadas en RDF.

3.6.2 Componentes

El lenguaje SPARQL posee tres componentes importantes: URIs, literales y variables, todas ellas procedentes del lenguaje RDF.

- URIs: sirven para especificar los recursos. Su descripción se encuentra en la RFC 3987.
- Literales: se describen como una cadena de caracteres encerradas entre " ", y que representan objetos constantes elementales.
- Variables: estas variables son globales, además deben de ser prefijadas por "?" o "$" seguidas de un nombre cualquiera en el que no puede contener estos caracteres. Se utilizan para asignar valores de manera temporal, para almacenar resultados o para utilizar filtros sobre ellas.

3.6.3 Sintaxis

La sintaxis de SPARQL es similar a la conocida SQL, añadiendo algunas modificaciones para facilitar el análisis sintáctico del lenguaje:

PREFIX prefijo1, prefijo2..., prefijoX

SELECT/CONSTRUCT var1, var2,...varX

WHERE {condiciones}

El primer bloque (PREFIX) incluye los prefijos que serán utilizados en la consulta, de modo que se asigna a una combinación de letras una dirección. El segundo bloque (SELECT/CONSTRUCT) indica las variables de las que se desea conservar su valor tras ejecutar la consulta, o mejor dicho los valores que se desea retornar. El tercer bloque (WHERE) es donde está la consulta en sí, incluye las condiciones que se tienen que dar en los RDF para que seleccione esa tripleta de valores como una coincidencia, y se almacenen las variables indicadas en el bloque anterior. El lenguaje SPARQL tiene cuatro tipos de operaciones sobre los RDFs, a continuación se pueden apreciar las cuatro con un ejemplo explicativo.

- SELECT es la instrucción más común. Devuelve las variables que se le indican para los casos en los que coinciden con el patrón de búsqueda.

```
PREFIX foaf: <http://sparql.com/foaf/0.1/>
SELECT ?nameX ?nameY ?nickY
WHERE
{ ?x foaf:knows ?y ;
foaf:name ?nameX .
?y foaf:name ?nameY .
OPTIONAL { ?y foaf:nick ?nickY }
}
```

Figura 14: Sintaxis query SELECT en SPARQL [12]

En la Figura 14, se observa que se declara un prefijo llamado “foaf” al que se le asigna una dirección web que hará de prefijo posteriormente. Mediante el SELECT se indica que se quiere saber el valor de las variables ?nameX, ?nameY y ?nickY. Dentro del bloque WHERE (encerrado mediante llaves), se observa las tres condiciones que se le imponen a los RDFs resultantes, las dos primeras indican que se busca un objeto cualquiera que tenga los atributos foaf:name y foaf:knows (aquí se puede apreciar cómo se utilizan los prefijos declarados previamente), y de los objetos que hayan sido devueltos como valor del atributo foaf:knows, se buscan aquellos que tengan nombre. En esta consulta aparece el bloque OPTIONAL dentro del WHERE, en este caso la función de OPTIONAL, es evitar que lo que se especifique dentro sea una condición, convirtiéndolo en algo opcional; que si aparece es añadido en los resultados.

- CONSTRUCT devuelve un grafo RDF construido por la sustitución de variables en un conjunto de tres plantillas.

```
PREFIX foaf: <http://sparql/foaf/0.1/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT { <http://sparql.org/person#Alice> vcard:FN ?name }
WHERE { ?x foaf:name ?name }
```

Figura 15: Sintaxis query CONSTRUCT en SPARQL [12]

- ASK devuelve un valor verdadero o falso indicando si se ha encontrado algún valor coincidente con los patrones de la consulta. Realmente es un SELECT simplificado ya que no devuelve los casos que han coincidido.

```
PREFIX foaf: <http://sparql/foaf/0.1/>
ASK { ?x foaf:name "Alice" }
```

Figura 16: Sintaxis query ASK en SPARQL [12]

- DESCRIBE devuelve un grafo RDF representativo sobre las relaciones entre los objetos que enlazan con el especificado en la instrucción, describiendo estos recursos encontrados.

```
DESCRIBE <http://example.org/>
```

Figura 17: Sintaxis query DESCRIBE en SPARQL [12]

Existe una cláusula adicional FILTER que permite filtrar las consultas utilizando para ello operadores con el fin de obtener resultados más exactos. Existen 4 tipos de operadores:

- Operadores lógicos:
    - A||B, A&&B, ¡A, (A)
    - De comparación: =, !=, <, >, <=, >=
- Operadores aritméticos.
    - Unarios +A, -A
    - Binarios (A op B)
- Operadores y funciones RDF.
    - Booleans. BOUND(A), isURI(A), isBLANK(A), isLITERAL(A)
    - String. STR(A), LANG(A), DATATYPE(A)
- Operadores para realizar STRINGS

3.6.4 Funcionamiento

Las consultas SPARQL cubren tres objetivos:

- Extraer información en forma de URIs y literales.
- Extraer sub-estructuras RDF.
- Construir nuevas estructuras RDF partiendo de resultados de consultas.

Un inconveniente para este lenguaje es la falta de posibles acciones sobre los datos, ya que únicamente pueden realizarse operaciones de lectura (de ahí que se incluya su uso junto a Jena), lo que limita al sistema a leer y comparar datos. En la Figura 18 se puede observar una estructura en formato RDF, sobre la que el sistema puede realizar consultas.

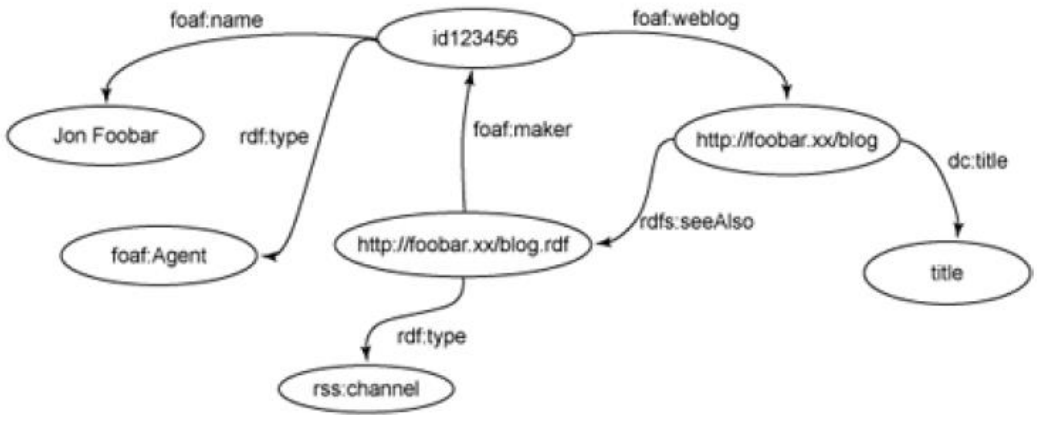

Figura 18: Ejemplo de Grafo RDF [12]

Una consulta simple sobre este grafo sería la mostrada en la Figura 19:

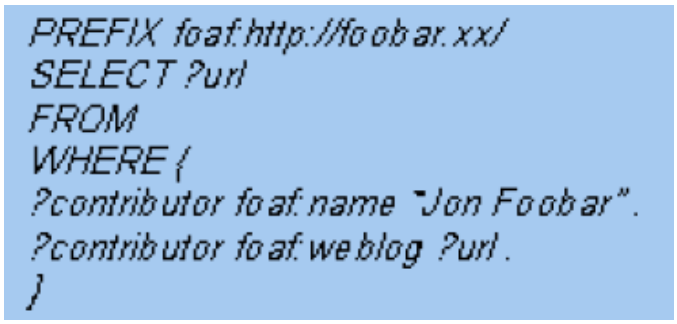

Figura 19: Query de ejemplo en SPARQL [12]

En ella se desea encontrar la URL de una persona llamada "Jon Foobar". Como se puede ver la manera de referirse al objeto que cumple la condición (en este caso que su nombre sea igual a "Jon Foobar"), ha de ser el mismo que del que se va a extraer el dato requerido (en este caso la dirección del Weblog)

En la primera línea encontramos la palabra clave PREFIX. Su función es equivalente a la declaración de namespaces en XML: asociar una URI a una etiqueta, que se usará en adelante para describir el namespace. Pueden definirse tantas de estas etiquetas en una misma consulta como se necesiten o deseen.

El comienzo de la consulta queda marcado por la palabra clave SELECT. Semejante a su uso en SQL, sirve para definir los datos que deben ser devueltos en la respuesta. En el ejemplo tan sólo la URL. La palabra clave FROM identifica los datos sobre los que se ejecutará la consulta. Una consulta puede incluir varios FROM. La palabra WHERE indica las restricciones de los RDF, que para este caso se obliga a que el objeto ?contributor tenga un atributo foaf:name, equivalente al literal "Jon Foobar", y se solicita el atributo foaf:weblog de los casos que coinciden.

3.7 Entornos de desarrollo: Protegé

Protegé [14] es un editor de ontologías y bases de conocimiento gratuito y abierto (actualmente se encuentra disponible la versión 4.0 en formato libre). Se encuentra basado en Java y soporta Frames, XML Schema, RDF y OWL y además cuenta con un ambiente "Plug-and-play". Sirve de apoyo de una sólida comunidad de desarrolladores y académicos, así como el gobierno y usuarios corporativos, los cuales están utilizando las soluciones de Protégé en áreas tan diversas como la biomedicina, la recogida de datos y modelado corporativo. Protégé al igual que otros sistemas basados en marcos describen ontologías declarativas, estableciendo explícitamente cual es la jerarquía de clases y a qué clases individuales pertenece. Protégé proporciona soporte para la construcción del marco basado en ontologías y es también una sólida plataforma para ontologías OWL. Estas son sus principales características:

• Editor incorporado en la parte superior de la fuente abierta OWL API Support para OWL 2.

• Entradas directas a memoria, conexión con Pellet, FaCT++ y Hermit.

• Soporte para editar múltiples ontologías.

• Permite la creación, importación y exportación de fichas configurables por el usuario.

• Su marco GUI es Configurable. Además de permitir la creación, importación y exportación de la configuración del usuario.

• Presenta múltiples vistas alternativas de la misma ontología.

• OWL API para OWL 2.0 provee un modelo eficiente en memoria.

• El marco plug-in es compatible con OSGi Equino (fácilmente extensible) [14].

• La aplicación genérica del marco está separada del Kit editor OWL.

• En la modularización hace un uso inteligente de los repositorios local / global para manejar las dependencias de importación. Además permite la carga de múltiples ontologías en un solo trabajo.

• Permite el cambio entre las ontologías de forma dinámica.

• Sugerencias UI para mostrar en las declaraciones hechas en la ontología.

• Refactorización: fusión de las ontologías y la eliminación de todas las importaciones redundantes.

• Refactorización: capacidad de mover axiomas entre ontologías. Permite Renombrar (incluyendo varias entidades), manipulación y separaciones de los diferentes axiomas; una rápida creación de clases definidas; transforma en diversas restricciones (incluyendo recubrimiento(covering));conversión de IDs a etiquetas de conversión y traslado de axiomas entre ontologías.

• Infiere axiomas que muestra en la mayoría de las vistas estándar.

• Interfaz directa con FaCT++ [22].

• Interfaz directa a Pellet reasoner [22].

• Interfaz directa a Hermit reasoner [13].

• Plug-ins de Reasoners.

Siguiendo con las características de OWL, las siguientes peculiaridades son importantes a la hora de resaltar en el aspecto de la edición de OWL.

• Prestación de conformidad de las entidades de la ontología, usando fragmentos URI o valores de anotación.

• Análisis de la descripción OWL (también admite nombres en las anotaciones).

• Incorporado el cambio de apoyo que permite deshacer los cambios y compuestos.

• Histórico de autocompletado y expresiones.

• Resaltado de sintaxis.

• Creación automática de documentos y etiquetas IDs para nuevas entidades.

• Normas de edición SWRL [14].

Con referencia a los plug-ins que existen hoy día se resaltan:

• Arquitectura altamente conectable con el apoyo de muchos de los diferentes tipos de Plug-in incluyendo vistas, menú de acciones, reasoners, preferencias y gestores circulares.

• Actualización automática de nuevos Plug-ins y versiones.

• Muchos Plug-ins disponibles incluyendo reasoners, matrices, secuencias de comandos, árboles existenciales, extracción de textos, explicaciones, procesamiento de la ontología, pruebas de marco, generación de lenguaje natural y más.

• Permite reconfigurar la interfaz de usuario añadiendo y eliminando tablas y vistas a través de sus pestañas, así como ver los menús.

Protégé se mantiene en constante actualización permitiendo utilizar un entorno cada vez más estable de cara a los usuarios y contribuyentes, ya que en los últimos años se ha producido una gran cantidad de cambios en el fondo de la API de OWL.

Capítulo 4

Diseño de la implementación propuesta

En este capítulo se describe el diseño de la ontología propuesta así como los dos tipos de implementaciones que se han realizado utilizando dos API distintas de Java, Jena y OWL API.

4.1 Estructura de la ontología a desarrollar

La principal idea ha sido conseguir una ontología básica con tres recursos clave: Paciente, Enfermo, Sano. Enfermo y Sano serán recursos que hereden de Paciente y se han establecido normas que diferencien entre un paciente sano y uno enfermo en función de unos parámetros básicos tales como la tensión arterial, la frecuencia cardiaca, la frecuencia respiratoria, el nivel de oxígeno en sangre y la temperatura corporal. De esta manera se consigue una forma muy sencilla y eficaz de diferenciar un paciente enfermo de un paciente sano al describir la ontología; en función de los valores que tomen los seis indicadores principales de salud.

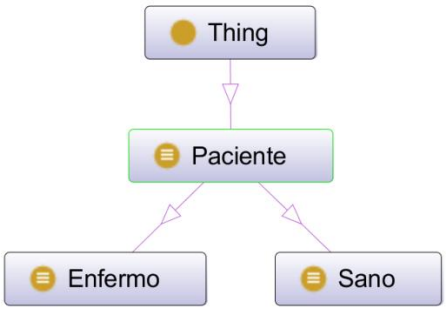

Figura 20: Dependencias de la ontología

Los valores elegidos para clasificar los pacientes son los siguientes:

	Enfermo	Sano
Tensión sistólica	Mayor que 140 mmHg	Resto de valores
Tensión diastólica	Mayor que 90 mmHg	Resto de valores
Frecuencia cardiaca	Mayor que 100 puls/min	Resto de valores
Frecuencia respiratoria	Mayor que 20 resp/min	Resto de valores
Nivel de oxígeno	Menor que 95 %	Resto de valores
Temperatura	Mayor que 37 ºC	Resto de valores

Figura 21: Clasificación enfermo/sano en la ontología

4.2 Descripción del recurso Paciente

El recurso Paciente desciende directamente del recurso Thing como se puede ver en la Figura 20. Pertenecientes al dominio del recurso Paciente se han incluido las seis propiedades del tipo DataProperty pertenecientes a los principales indicadores de salud del ser humano.

Las propiedades son las siguientes:

- *tensionSistólica*: de tipo funcional (cada Paciente solo podrá tener un valor de tensión sistólica) y cuyo rango son los enteros.
- *tensiónDiastólica*: de tipo funcional y cuyo rango son los enteros.
- *frecuenciaCardiaca*: de tipo funcional y cuyo rango son los enteros.
- *frecuenciaRespiratoria*: de tipo funcional y cuyo rango son los enteros.
- *nivelOxígeno*: de tipo funcional y cuyo rango son los enteros.
- *temperatura*: de tipo funcional y cuyo rango son los números decimales.

Para el desarrollo de esta ontología no se han desarrollado ningún tipo de propiedades de tipo ObjectProperties ya que no se ha considerado necesario y complicaría la ontología sin ofrecer ningún tipo de ventaja adicional en este caso.

4.3 Descripción del recurso Enfermo

El recurso Enfermo es una clase que hereda de Paciente a la que se le han añadido las restricciones de la Figura 22. Era importante que un razonador fuese capaz de clasificar a un Paciente sin ninguna ayuda así que se decidió utilizar el campo equivalentClass (explicado en el capítulo 3) de OWL para añadir las restricciones. Como se puede comprobar en la captura de Protegé, (entorno de desarrollo utilizado para la elaboración de la ontología) la sintaxis es casi explicativa para una persona con formación técnica:

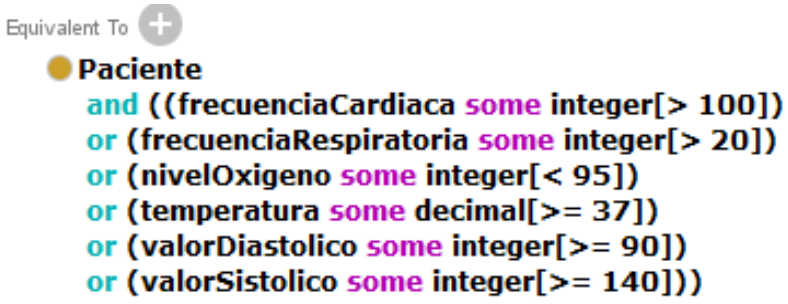

Figura 22: Descripción del recurso Enfermo

Si cualquiera de las propiedades de un recurso Paciente se incluye en los rangos establecidos del recurso Enfermo, un razonador bien construido será capaz de inferir que ese Paciente también pertenece al recurso Enfermo, así se podrá acceder a todo el grupo de pacientes que están

enfermos con tan sólo pedirle a la ontología que te devuelva todos los individuos pertenecientes al recurso Enfermo. Esa es una de las grandes ventajas de los lenguajes semánticos, permiten mucho dinamismo a la hora de clasificar recursos.

4.4 Descripción del recurso Sano

El recurso Sano como se puede observar en la Figura 20 también depende del recurso Paciente. En un primer momento se implementó como un recurso análogo al recurso Enfermo, es decir se usó el campo equivalentClass de OWL y se invirtieron las desigualdades de dicho recurso. Después de varias pruebas empíricas se comprobó que la forma más eficiente de describir el recurso Sano era decir que es un recurso de tipo Paciente y excluirlo del grupo de los Enfermos. De esta manera se ganaba mucho tiempo en el momento de la clasificación por parte del razonador ya que a la hora de clasificar los pacientes sólo tenía que realizar las comprobaciones de los pacientes enfermos y una pequeña comprobación más (exclusión del grupo) para clasificar a los sanos. Si observamos la captura de Protegé para definir el recurso Sano se puede apreciar de nuevo cómo la sintaxis es totalmente intuitiva demostrando la facilidad de clasificación del lenguaje semántico (ver Figura 23).

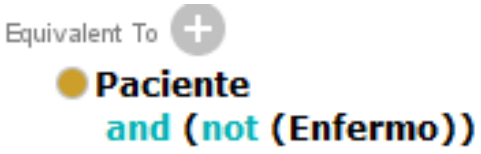

Figura 23: Descripción del recurso Sano

4.5 Estructura de las clases Java a desarrollar

Con la ontología creada el siguiente paso ha sido desarrollar clases Java que trabajasen con las APIs Jena y OWLAPI que fuesen capaces de trabajar con dicha ontología y conseguir clasificar los pacientes según su estado de salud. Con este fin se han desarrollado las clases PlantaHospitalJena y PlantaHospitalOWL con fines similares y con la principal diferencia de que cada una usa un API distinto para trabajar con la ontología. Como se puede presuponer PlanaHospitalJena utiliza Jena y PlantaHospitalOWL API utiliza OWLAPI. La utilidad de ambas clases es muy similar, pero como veremos más adelante la eficiencia a la hora de clasificar difiere bastante la una de la otra.

Las clases PlantaHospital necesitan trabajar con los datos de pacientes así que la primera necesidad que surgió con el diseño fue la creación de una clase Paciente que contuviese los datos que necesitase la ontología.

4.5.1 Estructura de la clase Paciente

La clase Paciente no es más que un contenedor de los datos del Paciente y el número de habitación donde se encuentra ingresado. No se almacena en ningún momento ningún tipo de registro de objetos de la clase Paciente ya que el encargado de ello será la ontología.

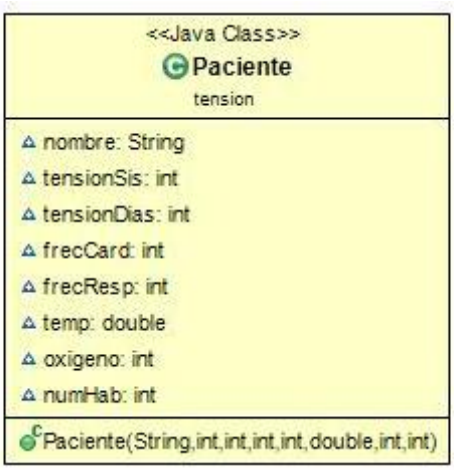

Figura 24: Diagrama UML de la clase Paciente

Se puede observar en la Figura 24 que la clase paciente consta del atributo nombre que será la URI a la que referencia en la ontología, de los atributos correspondientes a las propiedades de la ontología y del atributo numHab que es el número de habitación comentado anteriormente. El único método con el que cuenta es el constructor que será invocado por las clases PlantaHospital para crear pacientes.

4.5.2 Estructura de la clase PlantaHospital

La clase PlantaHospital es la clase padre de la que heredarán las clases PlantaHospitalJena y PlantaHospitalOWL anteriormente mencionadas. Se ha utilizado la herencia ya que comparten varios atributos tales como la las direcciones URI de la ontología y el buffer de lectura y escritura que se usa para cargar la ontología en el API correspondiente y añadir los pacientes necesarios en el archivo OWL.

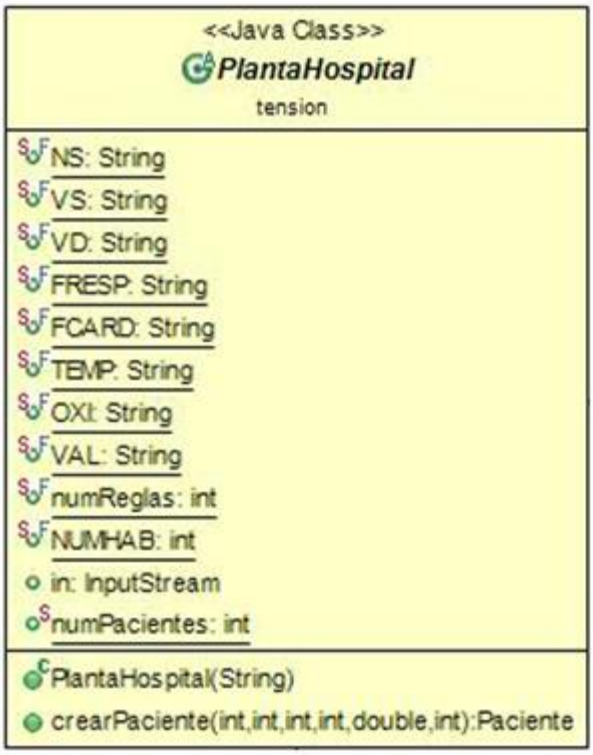

Figura 25: Diagrama UML de la clase PlantaHospital

Los atributos de tipo *static final* que se pueden observar en el diagrama de la Figura 25 tales como *NS*, *VS*(valor sistólico) o *VD*(valor diastólico) corresponden las URIS de las propiedades de la ontología, *NUMHAB* es el número de habitaciones de la planta y *numPacientes* el número de pacientes que se encuentran ingresados. El atributo *in* del tipo *InputStream* es el buffer de lectura y escritura que se utiliza para cargar la ontología en el API correspondiente. Dicho buffer se abrirá en el constructor de esta clase al que llamarán las clases hijas en la primera línea de sus propios constructores. El método *crearPaciente* simplemente devuelve un objeto de tipo Paciente con su número de habitación correspondiente.

4.5.3 Estructura de la clase PlantaHospitalJena

La clase PlanaHospitalJena hereda de la clase descrita en el apartado anterior PlantaHospital a la que se le añaden dos atributos específicos del API de Jena: *m* de tipo *OntModel* que será el modelo donde se cargará la ontología creada anteriormente e *inf* que es el modelo inferido, es decir, el modelo que gracias a un motor de razonamiento proporcionado por Jena distinguiría entre pacientes enfermos y sanos. Lamentablemente, el estado actual de desarrollo del API de Jena impide que los razonadores disponibles infieran conocimiento hasta el nivel demandado por nuestra ontología. Esta limitación es la que motivó el uso del API alternativo (OWLAPI) que si permite el uso de un razonador externo. Dado que Jena no dispone de motores de razonamiento potentes para diferenciar el estado de salud de los pacientes se optó por otra herramienta facilitada por dicho API: las consultas SPARQL. Estas consultas

(explicadas en el apartado 3.6) no utilizan la lógica definida en la ontología. Para poder obtener el estado de los pacientes, se han utilizado filtros en las consultas en los que se han tenido que volver a definir los valores que delimitan la salud del paciente que ya se habían definido en la ontología.

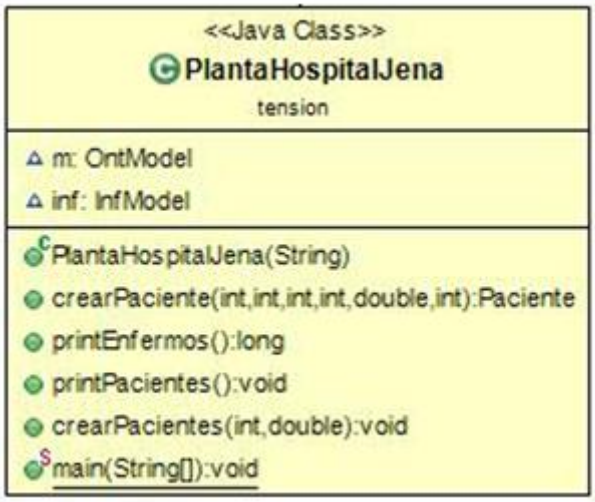

Figura 26: Diagrama UML de la clase PlantaHospitalJena

El método crearPaciente() introduce un individuo de clase paciente en la ontología y le asigna un nombre y una habitación. El método *crearPacientes(int numPac, double pro)* introduce en la ontología *numPac* pacientes con una probabilidad *pro* de que estén enfermos, este método se ha creado principalmente para realizar con más facilidad las pruebas necesarias de medición del tiempo de CPU y uso de memoria RAM que se utilizan en la clasificación de los pacientes. p*rintEnfermos() y printPacientes()* realizan la consulta en SPARQL. La primera le pide a la ontología todos los pacientes que estén fuera de los límites de salud y la segunda simplemente pregunta por todos los pacientes.

4.5.4 Estructura de la clase PlantaHospitalOWL

Como ya se ha comentado en los apartados anteriores, PlantaHospitalOWL utiliza la OWLAPI para comunicarse con la ontología. La razón de utilizar dicha API es que en este caso si permite utilizar motores de razonamiento externos lo cual sí que permite clasificar a los pacientes sin tener que incluir ningún tipo de herramienta ni control adicional, toda la clasificación la hace el razonador. Se decidió utilizar el motor Hermit [13] ya que es el más utilizado y el que mejor integración tiene con OWLAPI [16].

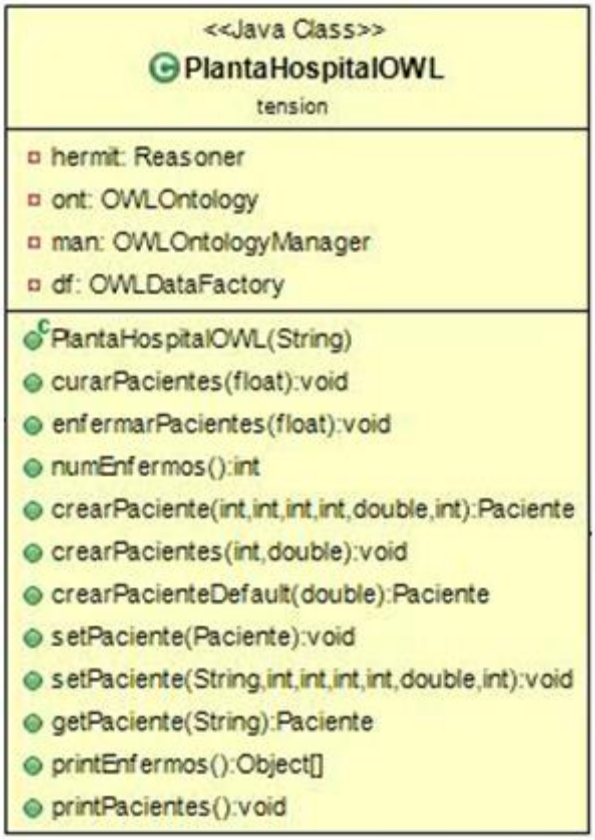

Figura 27: Diagrama UML clase PlantaHospitalOWL

El atributo *hertmit* de tipo *Reasoner* es el motor de razonamiento que se ha descrito en el párrafo anterior, *ont* de tipo *OWLOntology* es el modelo ontológico es donde se cargará la ontología, *man*de tipo *OWLOntologyManager* es el objeto que introducirá las variaciones dentro de la ontología, y *df* de tipo *OWLDataFactory* será donde se guarden los individuos pertenecientes a la ontología. Para esta última clase se han definido más métodos ya que será la clase que trabajará con la interfaz gráfica. Como en todas las clases tenemos en constructor *PlntaHospitalOWL(String)* que llama al constructor de la clase padre(abre el buffer), inicia el razonador y carga la ontología. También contamos con los métodos *printPacientes()* y *printEnfermos().* En este caso se han mejorado las prestaciones de *printEnfermos()* y devuelve en un array de objetos el nombre de los pacientes que están enfermos, de esta manera, la interfaz será capaz de localizar y señalar los enfermos gráficamente. Además de los métodos *crearPaciente()* y *crearPacientes()* se ha definido otro método llamado *crearPacienteDefault(double)* cuyo parámetro es la probabilidad de que dicho paciente pertenezca al grupo de los enfermos. Para ayudar al uso de la interfaz también se han implementado los métodos *getPaciente()* y *setPaciente()* que permiten acceder y modificar los valores de un paciente a partir de su nombre.

4.5.5 Estructura de la clase Interfaz

La clase Interfaz es la clase donde se ha implementado la interfaz gráfica que trabajará con la PlantaHospitalOWL para mostrar de una manera visual el funcionamiento de la ontología en una aplicación sencilla.

Figura 28: Diagrama UML de la clase Interfaz

Todos los atributos de tipo *ImageIcon* son imágenes con las que trabajará la interfaz a la hora de pintar las habitaciones si se da de alta un paciente o si enferma así como para enseñar una foto de perfil por defecto si se quiere ver el perfil de un paciente. Los atributos *tiempo*, *tiempoenfermar* y *tiempocurar* son parámetros para la simulación que lo que hacen es variar el la frecuencia con la que entran a funcionar los tres hilos de ejecución en paralelo con los que

trabaja esta interfaz. El primero la frecuencia en segundos de la comprobación del estado de salud de los pacientes, el segundo es la frecuencia en segundos del hilo que se encarga de enfermar con cierta probabilidad a los pacientes, y el tercero es la frecuencia con la que trabaja el hilo que se encarga de curar a los pacientes con cierta probabilidad.

El estado de la interfaz nada más arrancar la aplicación se puede observar en la Figura 29.

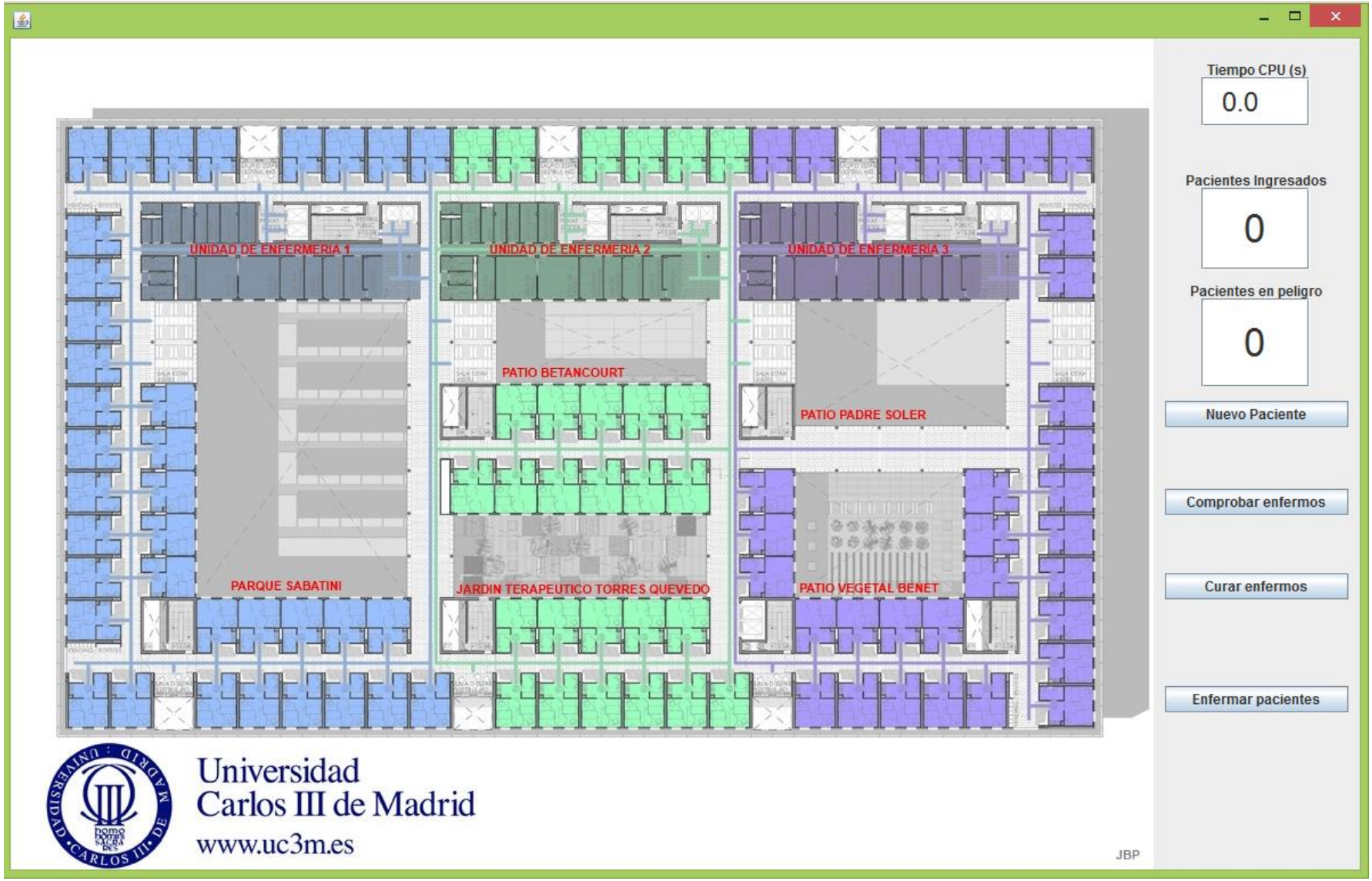

Figura 29: Interfaz gráfica en estado inicial

Se puede comprobar que el hospital está vacío, a la derecha tendremos los botones para dar de alta a pacientes nuevos con parámetros por defecto. Una vez se dé de alta el primer paciente, los hilos de comprobación empezarán a funcionar y a variar el estado de salud según unas probabilidades definidas en el código. El resultado de llenar el hospital se ve en la Figura 30.

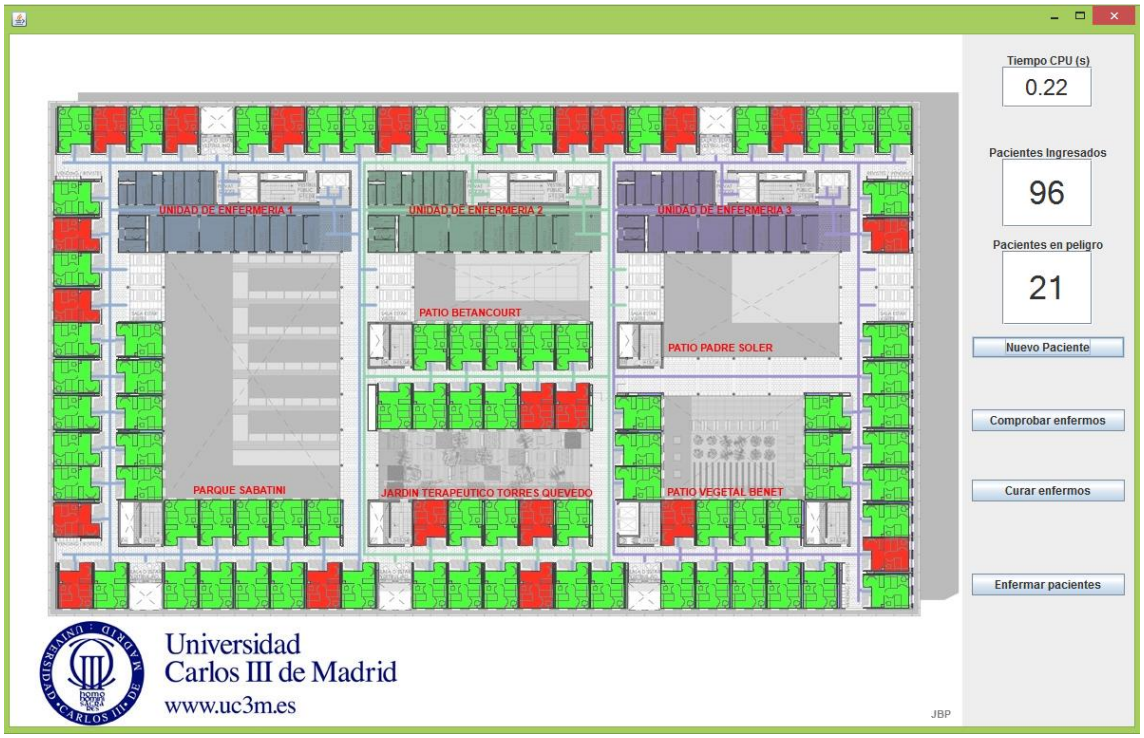

Figura 30: Interfaz gráfica con el hospital completo

Se puede comprobar en el cuadrado de arriba a la derecha el tiempo de CPU que ha tomado hacer la última comprobación del estado de los pacientes. Además se muestran el número de pacientes ingresados y el número de pacientes enfermos que hay ahora mismo en el hospital. Se han incluido además a modo de prueba, tres botones para acelerar o frenar los hilos que se utilizan para enfermar, curar o comprobar, de esta manera si pinchamos con el ratón en el botón "Enfermar pacientes" varias veces seguidas nos encontraremos con la situación de la Figura 31.

Figura 31: Interfaz con hilo de Enfermar Pacientes acelerado

Por otro lado si pulsamos el botón de curar pacientes la imagen es completamente distinta (ver Figura 32).

Figura 32: Interfaz con hilo de curar pacientes acelerado

Se ha incluido además la posibilidad de poder ver los datos de un paciente en tiempo real, basta tan sólo con pinchar con el ratón encima de la habitación que se quiere comprobar (Figura 33).

Figura 33: Perfiles genéricos de pacientes

También se puede comprobar el estado del paciente posicionando al ratón encima de la habitación de interés (Figura 34).

Figura 34: Detalle al posar el ratón encima de una habitación

Se han tenido en cuenta además errores típicos de este tipo de programas como por ejemplo intentar llenar el hospital con más clientes de los que se permiten mostrando ventanas de aviso de este estilo (ver Figura 35).

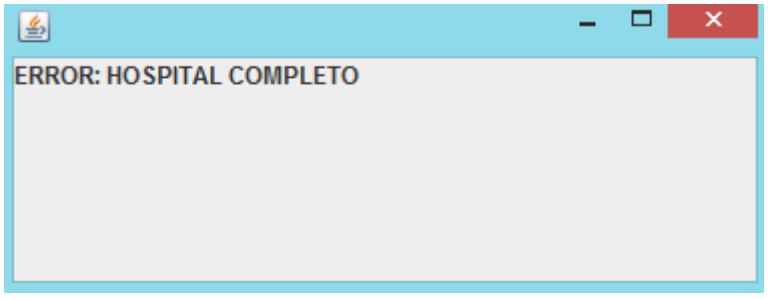

Figura 35: Mensaje de Hospital completo

4.5.6 Estructura general de la solución implementada

Se han desarrollado unos diagramas explicativos para facilitar la comprensión del trabajo desarrollado:

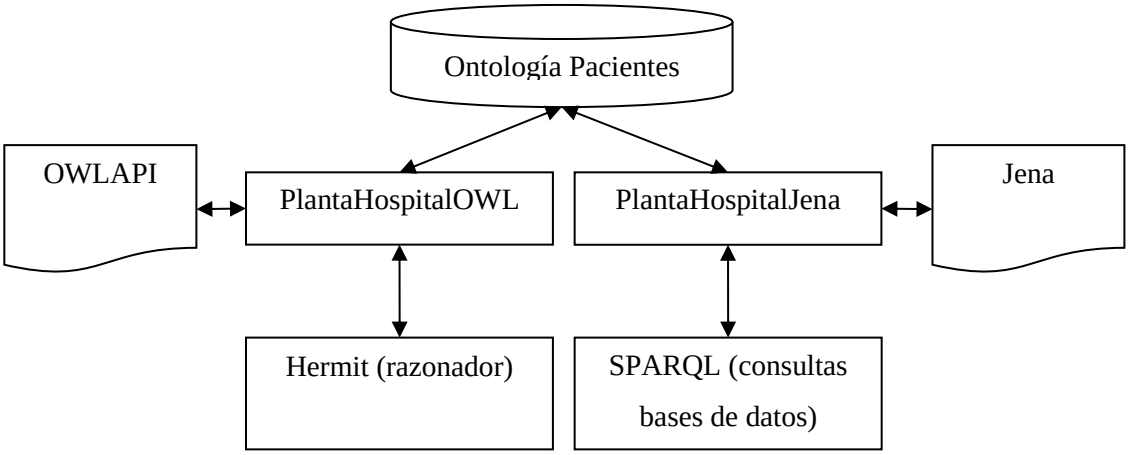

Figura 36: Dependencias de las clases desarrolladas

En la Figura 36 se aprecia la diferencia clara entre las clases PlantaHospitalOWL y PlantaHospitalJena. Como ya se ha explicado anteriormente, cada una utiliza un API distinto, pero además las formas de clasificar a los pacientes son también muy dispares. PlantaHospitalOWL utiliza las definiciones de la ontología y acude al motor de razonamiento (Hermit) para que clasifique cada uno de los pacientes. Por otro lado PlantaHospitalJena genera una query (consulta) en lenguaje SPARQL en el que indica los valores que definen a un enfermo para ser capaz de clasificarlo. Esto hace que PlantaHospitalJena no utilice el potencial de la ontología ya que en ningún momento infiere conocimiento a partir de las restricciones enunciadas en ella.

Por todo ello se decidió que la interfaz sería diseñada exclusivamente para PlantaHospitalOWL (ver Figura 37).

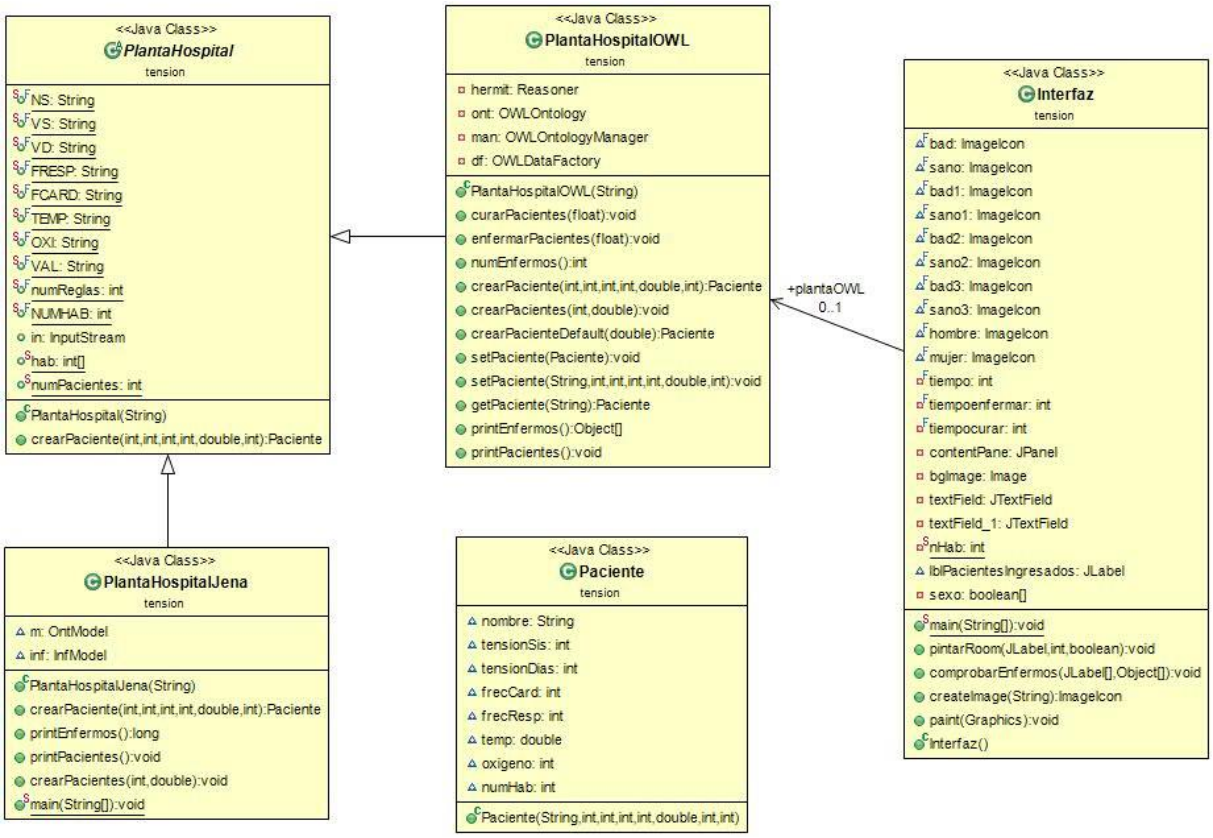

Figura 37: Diagrama de clases de la solución completa

En la Figura 37 se puede observar la dependencia (herencia) de las clases PlantaHospitalJena y PlantaHospitalOWL con la clase padre PlantaHospital así como el uso que hace la clase Interfaz de un objeto de la clase PlantaHopitalOWL. Con la utilización de los métodos desarrollados la interfaz es capaz de simular el funcionamiento de una planta de hospital.

Capítulo 5

Evaluación comparativa

En el capítulo anterior se ha explicado la diferencia surgida a la hora de clasificar pacientes con Jena y con OWLAPI. Es por eso que este capítulo se centra principalmente en mostrar las diferencias de rendimiento, en cuanto al tiempo consumido en la CPU y el uso de memoria RAM. Para ello se ha utilizado un ordenador con un microprocesador Intel Core i7 a 2,8 GHz con 8 GB de memoria RAM

Este estudio comparativo se ha estructurado en variación a dos variables: el número de individuos de tipo paciente que se dan de alta en el hospital y el número de reglas que se han incluido en la ontología. Se han desarrollado ontologías análogas a la presentada en el capítulo 4 con la única variación del número reglas adscritas a dicha ontología. El número de reglas que se han utilizado es el siguiente: 1, 2, 3, 4, 5, 10, 25, 50. Para cada una de esas ontologías se ha calculado el tiempo que tarda (para cada una de las dos soluciones desarrolladas) en clasificar a cierto número de pacientes, desde un individuo hasta 1000. A pesar de que Java se trata de un entorno con recolector de basura, se ha monitorizado la memoria RAM utilizada durante los procesos anteriormente discretos con la herramienta VisualVM de Java [28].

5.1 Caso 1. Ontología de una regla

Para el primer caso se ha elegido la ontología más sencilla, una única regla que diferencie entre un paciente enfermo y un paciente sano.

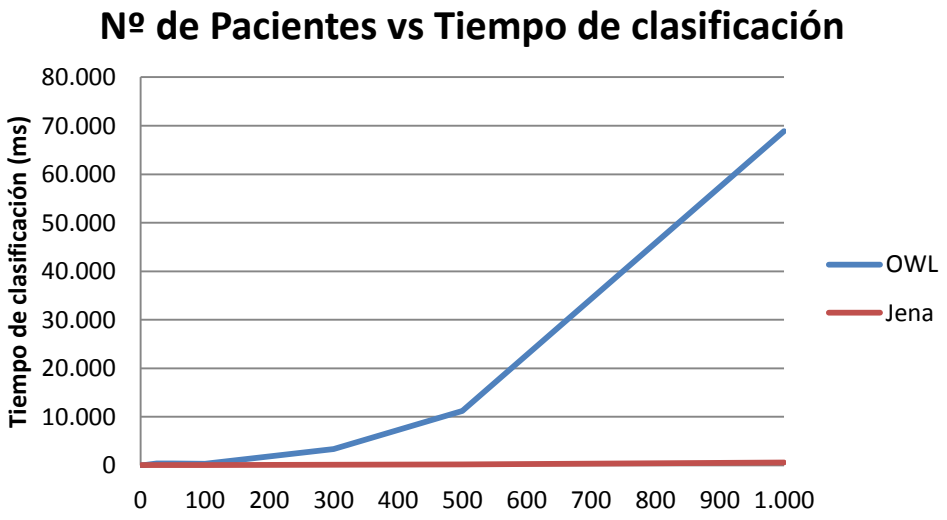

Figura 38: Tiempo de clasificación vs nº de pacientes Caso 1

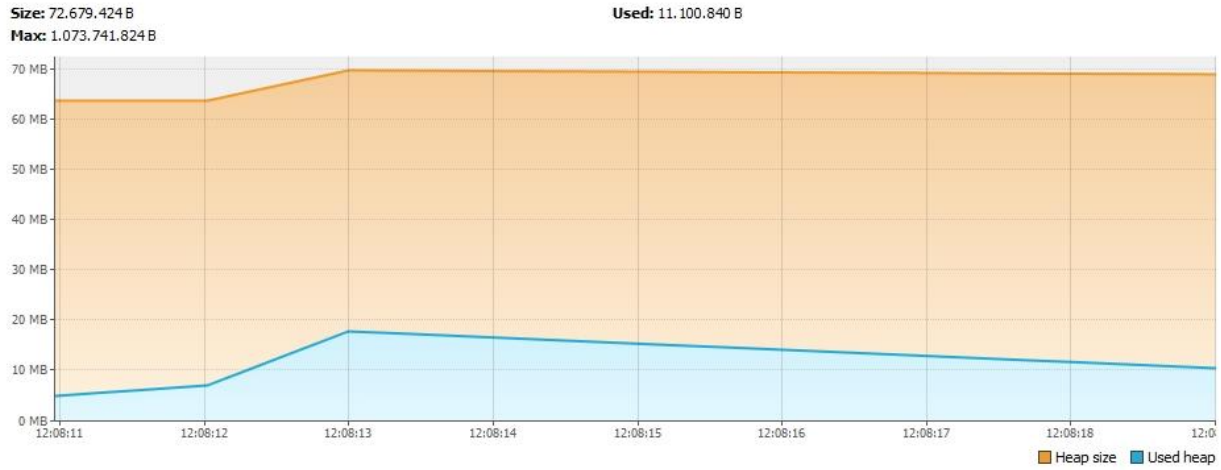

Figura 39: Uso memoria RAM Caso 1 Jena

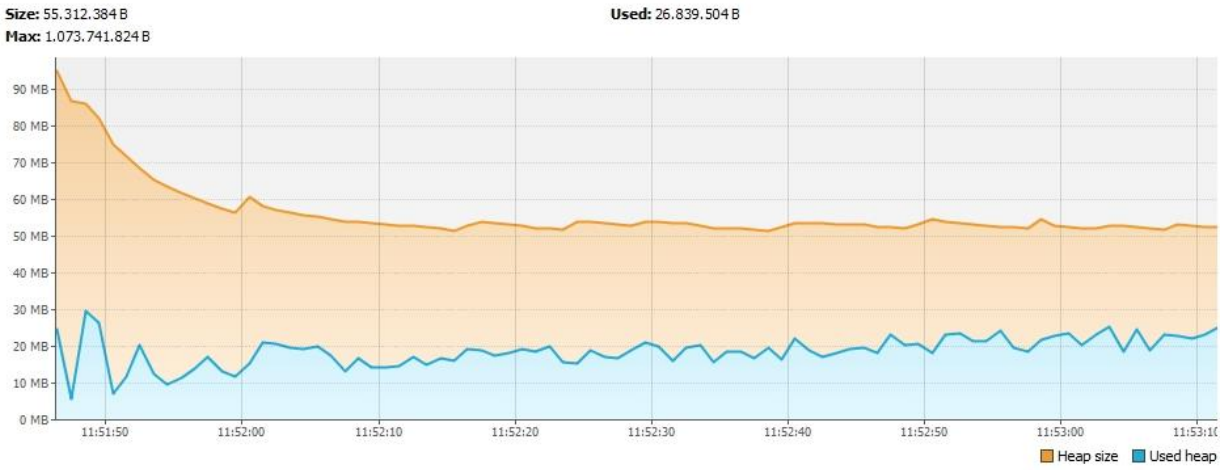

Figura 40: Uso memoria RAM caso1 OWLAPI

En la Figura 38 se puede apreciar como para los 100 primeros pacientes el tiempo de clasificación es muy similar tanto para el caso de Jena como para el caso de OWLAPI, no llegando ni siquiera a 1 segundo de duración. Sin embargo a partir de los 100 clientes el rendimiento de las consultas de SPARQL de Jena se mantiene con una tendencia lineal y una pendiente muy pequeña, mientras que OWLAPI toma una tendencia exponencial que hace que para clasificar 1000 pacientes Jena tarde menos de 1 segundo y OWLAPI con su razonador tarde 70 segundos. En cuanto al uso de memoria se puede observar que Jena tiene una media de 11 MB mientras que OWLAPI usa alrededor del doble durante la mayoría del proceso de clasificación.

5.2 Caso 2. Ontología de dos reglas

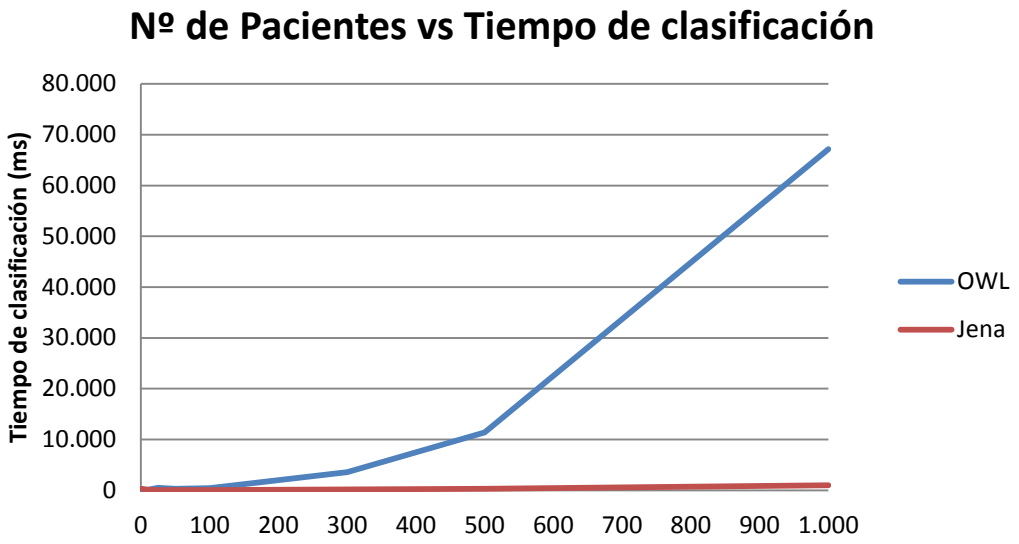

Figura 41: Tiempo de clasificación vs nº de pacientes Caso 2

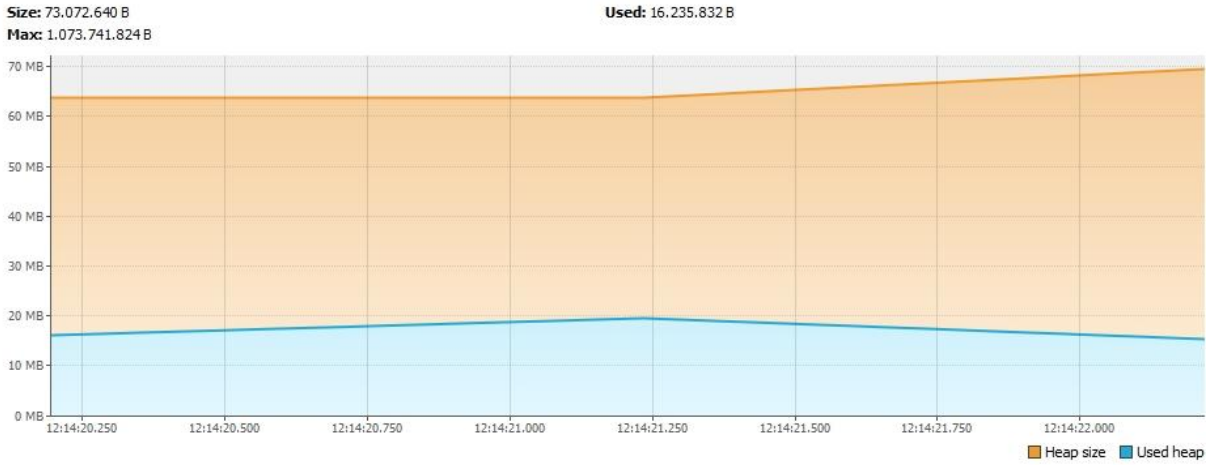

Figura 42: Uso de memoria RAM caso 2 Jena

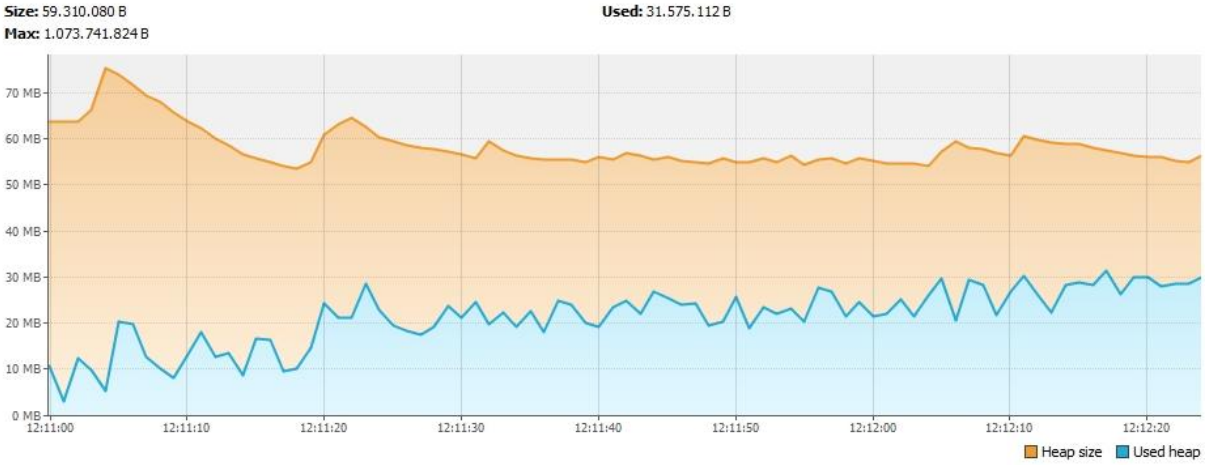

Figura 43: Uso de memoria RAM caso 2 OWLAPI

En el segundo caso apenas se aprecian cambios en cuanto a las graficas de tiempo respecto al de una sola regla. El tiempo para clasificar a 1000 pacientes continua cercano los 70 segundos en

OWLAPI mientras que en Jena sigue sin llegar al segundo. En el uso de memoria RAM si que se nota cierta subida, en Jena se llega hasta los 15 MB de media mientras que OWLAPI usa 25 MB.

5.3 Caso 3. Ontología de tres reglas

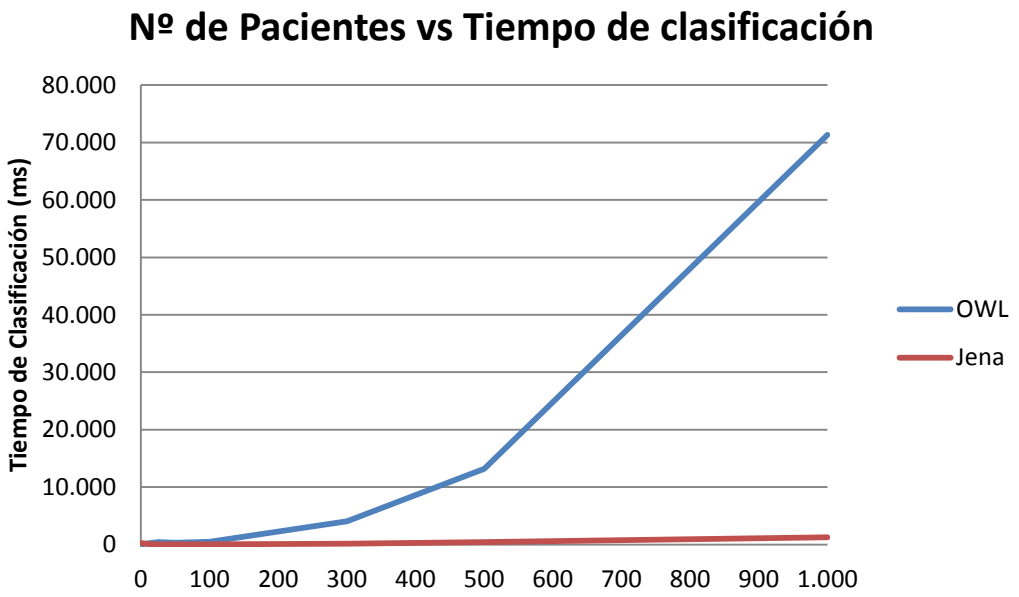

Figura 44: Tiempo de clasificación vs nº de pacientes Caso 3

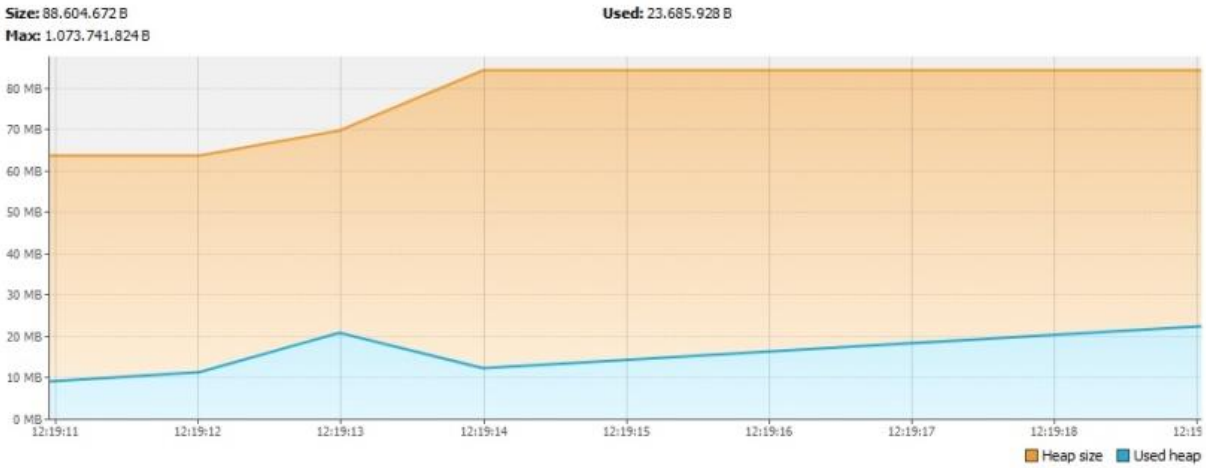

Figura 45: Uso de memoria RAM caso 3 Jena

Figura 46: Uso de memoria RAM caso 3 OWLAPI

En el tercer caso, usando una ontología de tres reglas, la diferencia sigue sin ser notable en cuanto al tiempo de clasificación. OWLAPI sigue con una tendencia muy similar a los dos casos anteriores mientras que se aprecia una pequeña subida de pendiente en Jena que ahora si que llega a estar por encima del segundo a la hora de clasificar 1000 pacientes (1,27 segundos). En cuanto al uso de memoria RAM tampoco se observa demasiada subida, la media de OWLAPI está en 27 MB mientras que la de Jena se encuentra en los 17 MB.

5.4 Caso 4. Ontología de cuatro reglas

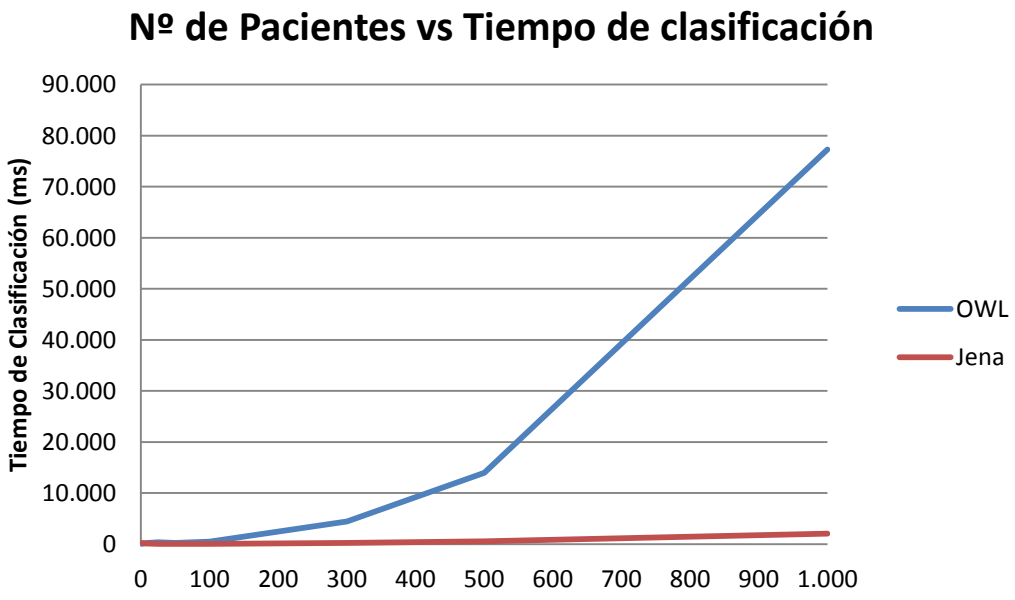

Figura 47: Tiempo de clasificación vs nº de pacientes Caso 4

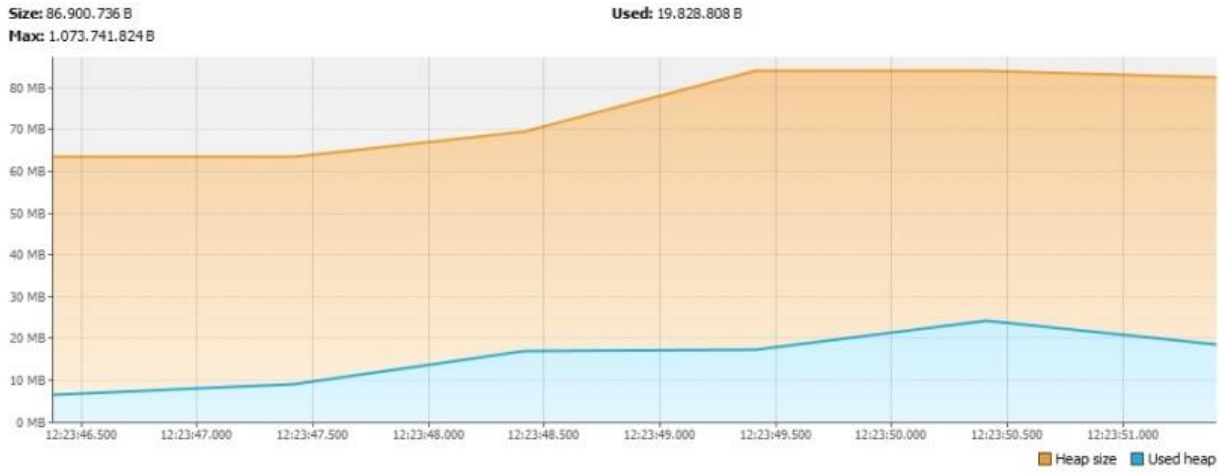

Figura 48: Uso de memoria RAM caso 4 Jena

Figura 49: Uso de memoria RAM caso 4 OWLAPI

En el cuarto caso, el uso de cuatro reglas en la ontología depara una subida de tiempo notable en ambas soluciones. OWLAPI sube hasta los 77 segundos para la clasificación de 1000 pacientes, mientras que Jena casi duplica su valor y se sitúa en 2,1 segundos. La tendencia de Jena sigue siendo lineal pero se empieza a apreciar una subida. La tendencia de OWLAPI no ha cambiado y es exponencial, El uso de memoria RAM para OWLAPI es de 30 MB y en Jena de 19 MB, un crecimiento casi inapreciable.

5.5 Caso 5. Ontología de cinco reglas

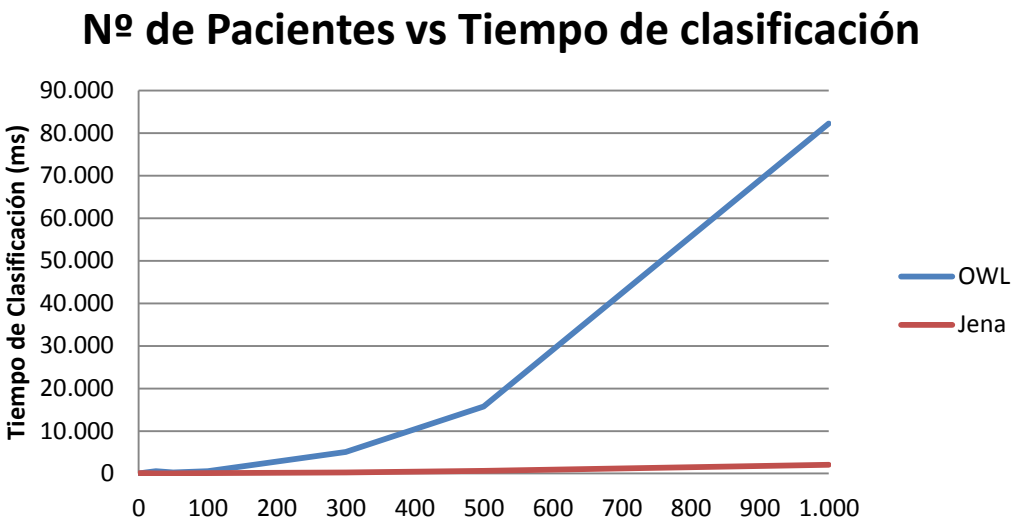

Figura 50: Tiempo de clasificación vs nº de pacientes Caso 5

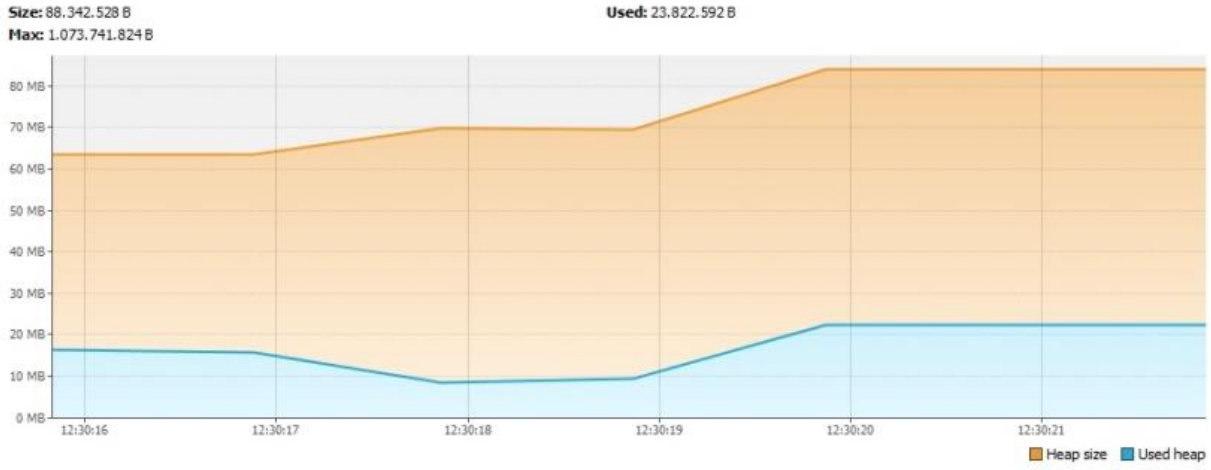

Figura 51: Uso de memoria RAM caso 5 Jena

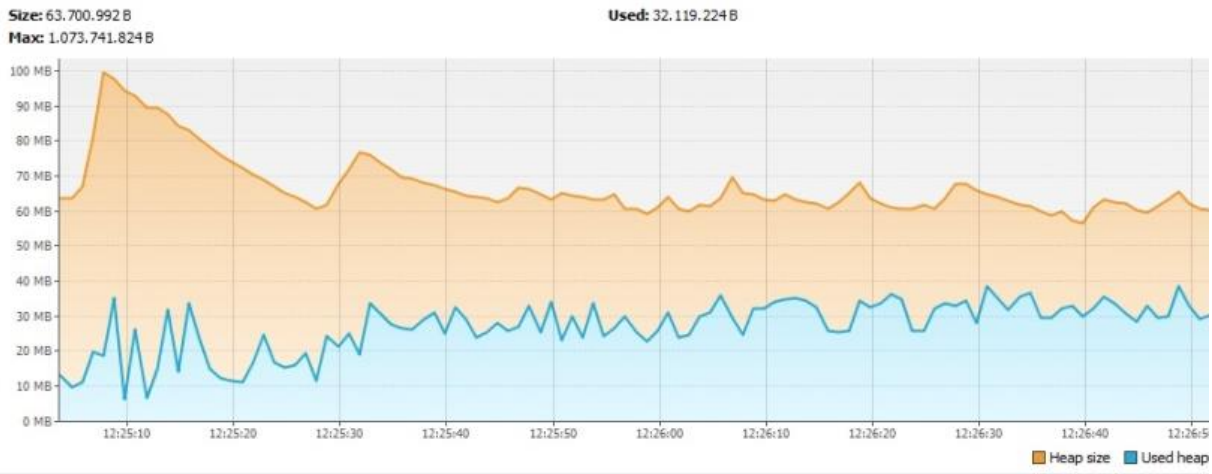

Figura 52: Uso de memoria RAM caso 5 OWLAPI

El paso de cuatro reglas a cinco reglas en cuanto a eficiencia es casi imperceptible. El tiempo de clasificación en Jena sigue en los 2,1 segundos mientras que en OWL ha aumentado un poco hasta los 81 segundos. El uso de memoria RAM sigue cerca de los 30 MB para OWLAPI y en

20 MB para Jena. Al observar la poca variación que se veía en las gráficas se decidió pasar a usar una mayor cantidad de reglas y es por lo que los casos 6, 7 y 8 incluyen 10, 25 y 50 reglas respectivamente.

5.6 Caso 6. Ontología de diez reglas

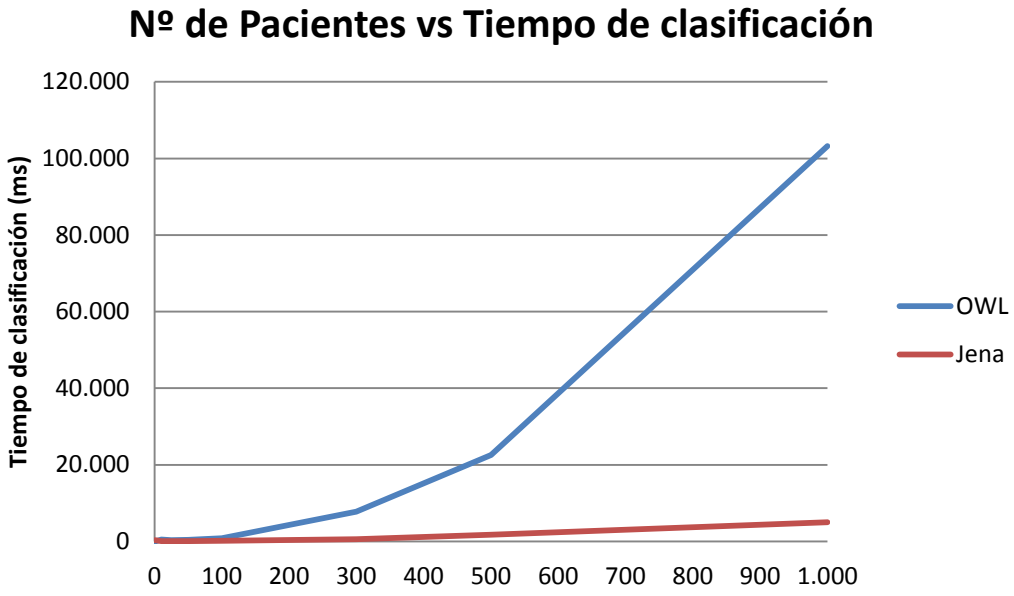

Figura 53: Tiempo de clasificación vs nº de pacientes Caso 6

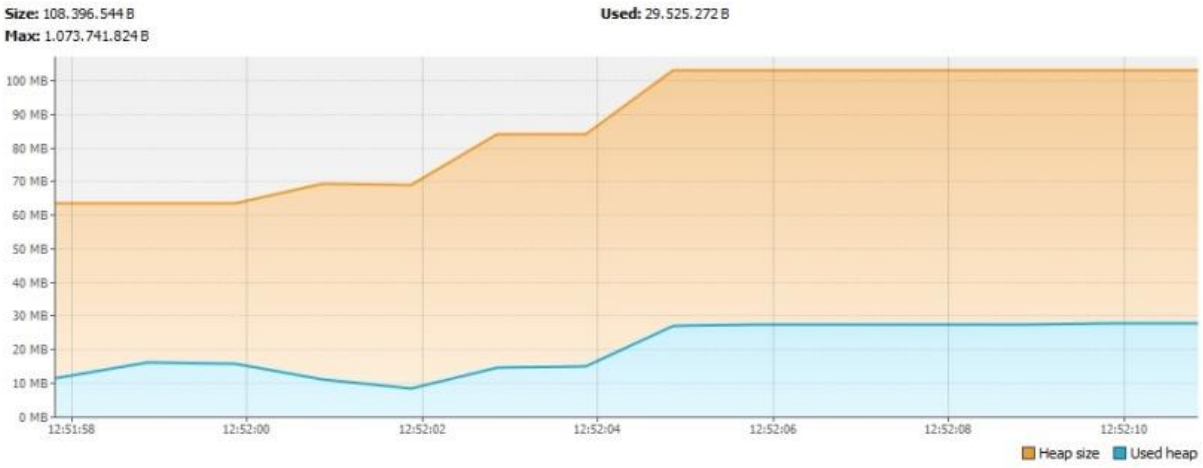

Figura 54: Uso de memoria RAM caso 6 Jena

Figura 55: Uso de memoria RAM caso 6 OWLAPI

Con el cambio de cinco a diez reglas en la ontología, la diferencia ya es más notable. OWLAPI sigue ascendiendo hasta los 105 segundos para clasificar 1000 pacientes, mientras que Jena ha sufrido una subida relativa mucho más destacable, llegando a los 5 segundos para clasificar el mismo número de pacientes. En cuanto al uso de memoria RAM siguen ascendiendo a un ritmo constante, OWLAPI llega a los 40 MB y Jena se queda en los 25 MB de media.

5.7 Caso 7. Ontología de veinticinco reglas

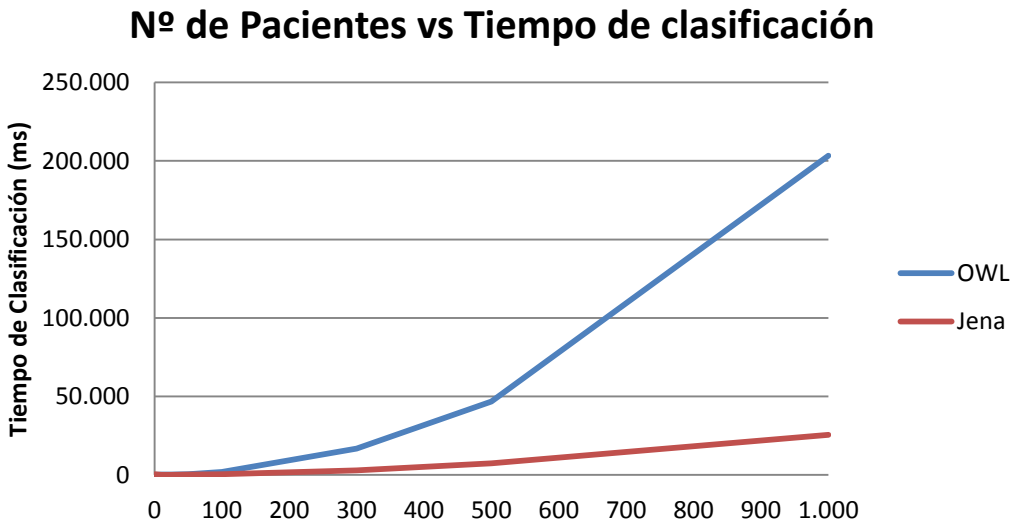

Figura 56: Tiempo de clasificación vs nº de pacientes Caso 7

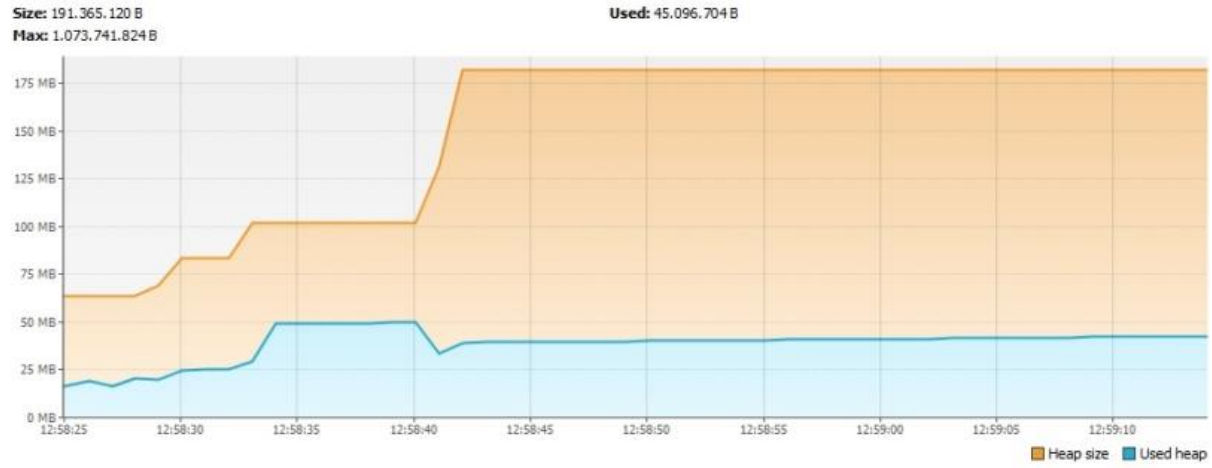

Figura 57: Uso de memoria RAM caso 7 Jena

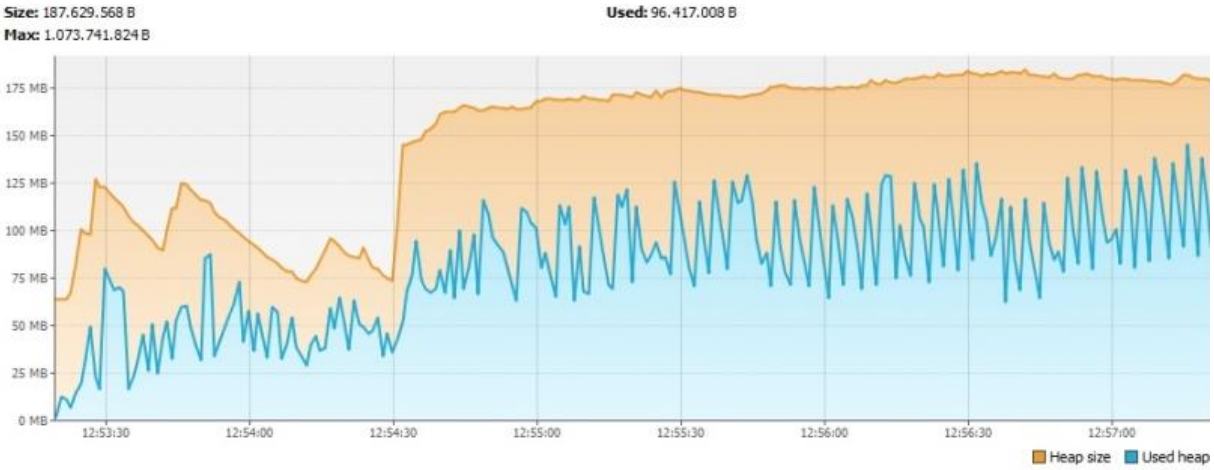

Figura 58: Uso de memoria RAM caso 7 OWLAPI

En el caso 7, por fin se puede observar como la tendencia de Jena empieza a tomar una forma exponencial. El tiempo de clasificación de 1000 pacientes ha ascendido hasta los 25 segundos mientras que en OWLAPI ha llegado hasta los 200 segundos para clasificar el mismo número de pacientes. En el uso de memoria RAM también se aprecia un incremento evidente pasando a los 40 MB de media en Jena y a los 90 MB en OWLAPI.

5.8 Caso 8. Ontología de cincuenta reglas

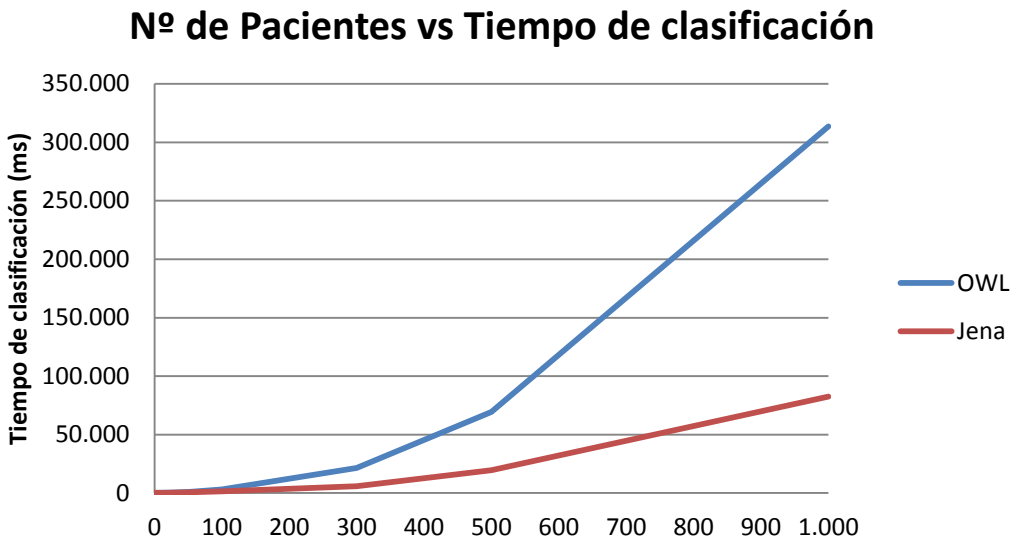

Figura 59: Tiempo de clasificación vs nº de pacientes Caso 8

Figura 60: Uso de memoria RAM caso 8 Jena

Figura 61: Uso de memoria RAM caso 8 OWLAPI

En el último caso estudiado, se ha utilizado una ontología con cincuenta propiedades y una restricción por cada una de ellas. Llegados a este punto se observa en la Figura 59 como Jena

crece también definitivamente de forma exponencial al tener que soportar filtros realmente grandes. Tarda 82 segundos en clasificar a 1000 pacientes mientas que OWLAPI lo consigue en 313 segundos. Al evaluar el uso de memoria RAM en las Figuras 60 y 61, se aprecia que será más alto en OWLAPI, ascendiendo hasta los 220 MB de media mientras que Jena se establece en unos 150 MB.

5.9 Conclusiones de la comparación entre Jena y OWLAPI

Después de realizar todas las pruebas descritas anteriormente se ha podido comprobar el rendimiento de los dos parsers para la web semántica (Jena y OWLAPI) y se han podido extraer diferentes conclusiones.

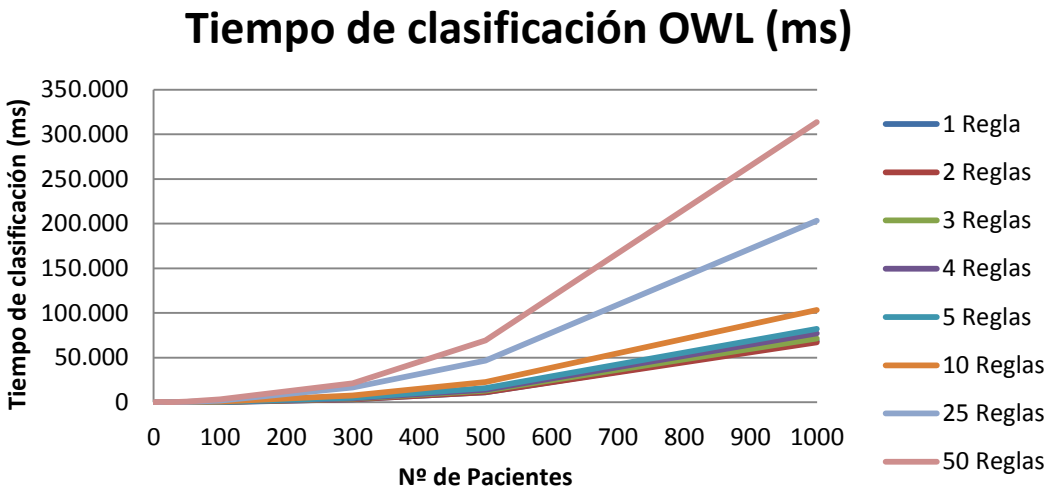

Figura 62: Evolución del tiempo de clasificación en OWLAPI según número de reglas de la ontología

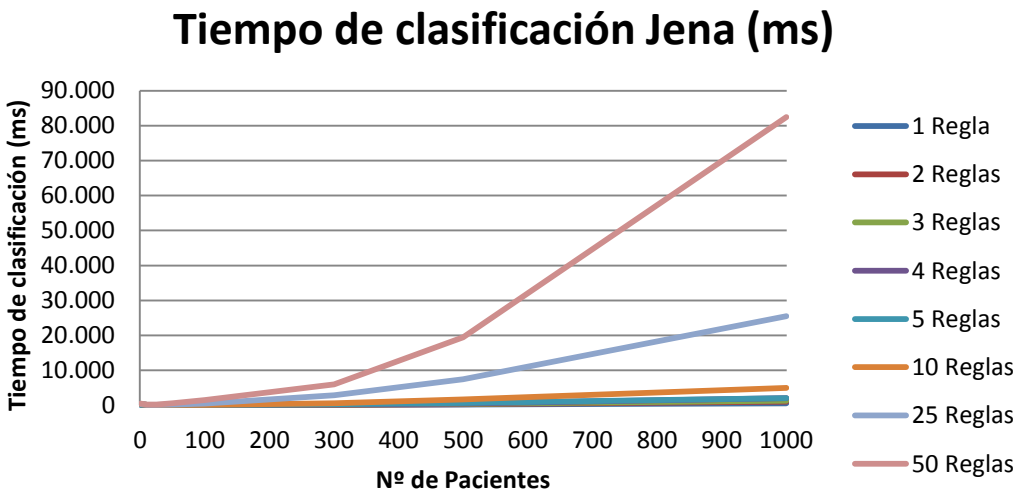

Figura 63: Evolución del tiempo de clasificación en Jena según número de reglas de la ontología

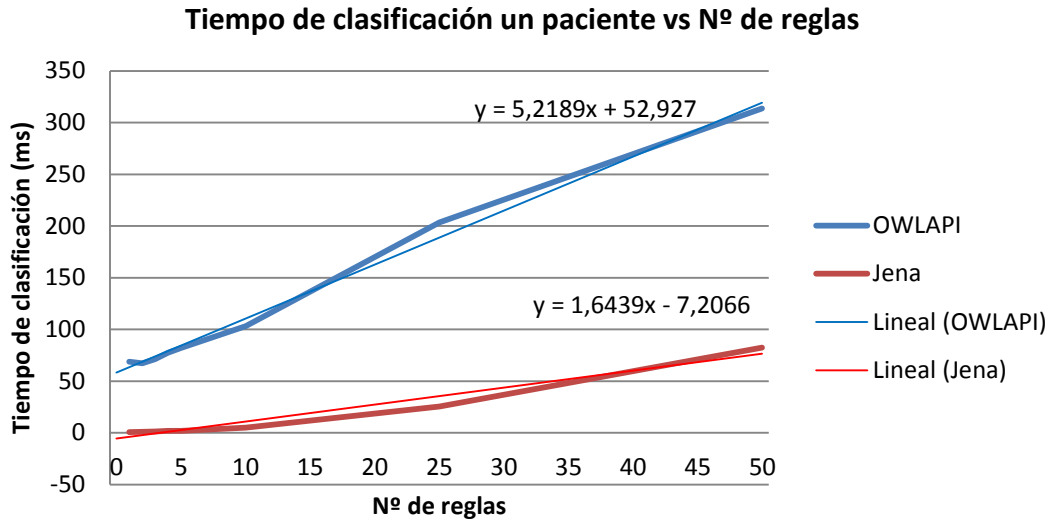

Figura 64: Tiempo de clasificación de un paciente frente al número de reglas para Jena y OWLAPI

El tiempo de clasificación de un paciente está directamente relacionado al número de reglas existentes en la ontología para ambas tecnologías. Sin embargo tal y como se puede apreciar en las Figuras 62, 63 y 64, OWLAPI necesita de mucho más tiempo para clasificar a un paciente. Esta diferencia de eficiencia en la clasificación, se hace más grande con el aumento del número de reglas. Esto se debe a que a la hora de realizar la clasificación el razonador de OWLAPI recorre la estructura de árbol de cada uno de los pacientes comprobando todas y cada una de las reglas ontológicas. Por otro lado, la consulta SPARQL de Jena tan sólo se limita a realizar un sencillo filtro en el total de pacientes por lo que el tiempo necesario es mucho menor.

Al analizarse los datos de la Figura 64 y estudiar la tendencia lineal de ambas, se llega a la conclusión de que por cada regla añadida a la ontología se añaden cerca de 5 milisegundos a la hora de clasificar a un paciente en OWLAPI mientras que en el caso de Jena tan sólo se añaden 1,6 milisegundos por cada regla.

En cuanto al uso de la memoria RAM, de nuevo es Jena la que sale vencedora. En la Figura 65 se puede observar la evolución del uso de la memoria RAM en función del número de reglas utilizadas y se aprecia una clara diferencia entre Jena y OWLAPI. Jena utiliza como máximo 150 MB para clasificar a los pacientes mientras que OWLAPI llega a usar 350 MB de pico.

El estudio de la tendencia lineal en este caso no no es aplicable pues ambas funciones siguen una tendencia exponencial aunque claramente el valor de la exponencial de Jena será menor que el de OWLAPI.

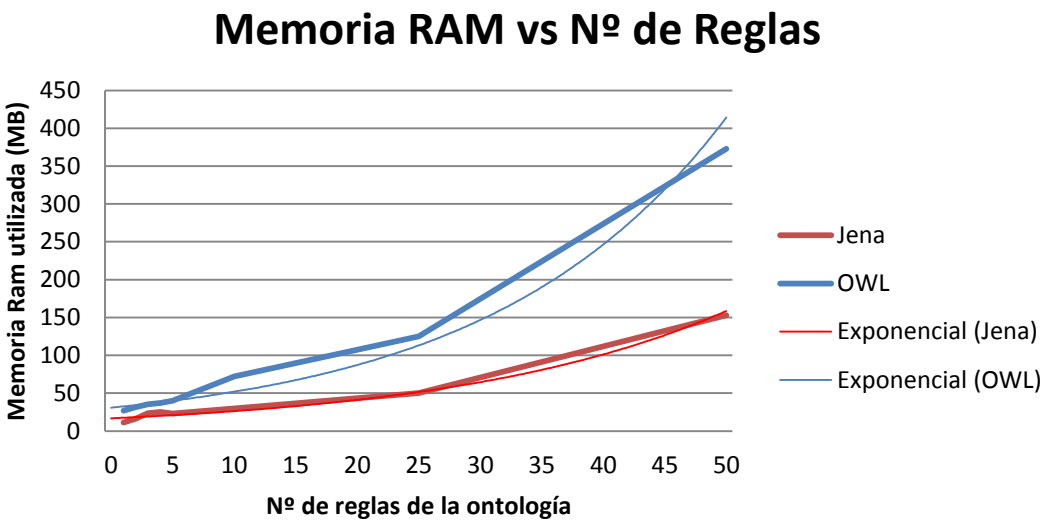

Figura 65: Evolución del uso de memoria RAM frente al número de reglas de la ontología

Cabe destacar un fenómeno sorprendente a la hora del estudio del uso de la memoria RAM, y es que basta con mirar cada uno de los ocho casos estudiados para notar una diferencia muy significativa.

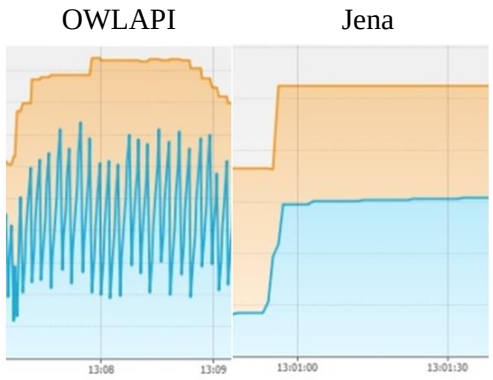

Figura 66: Uso de memoria RAM en OWLAPI y Jena

Mientras se está realizando una consulta en SPARQL no se permite la entrada en la CPU al recolector de basura de Java, sin embargo cuando se utiliza el razonador de OWLAPI sí que puede entrar a usar la CPU. Esa es la causa de que al observar el uso de memoria RAM en OWLAPI veamos un diente de sierra y en SPARQL se vea una línea casi plana como se puede apreciar en la Figura 66.

Capítulo 6

Conclusiones y líneas futuras de trabajo

El objetivo de este proyecto era comprobar la utilidad de las ontologías para la monitorización de los pacientes en tiempo real y después de este estudio ha quedado claro que su funcionamiento es más que aceptable.

Por otra parte del estudio comparativo se extraen diferentes conclusiones. Usar Jena y las consultas SPARQL es significativamente más eficiente que usar OWLAPI y el razonador Hermit. El uso de memoria es menor y el tiempo de CPU también para todos los casos estudiados. Sin embargo a la hora de producir cambios en la ontología, esos cambios son transparentes para OWLAPI ya que no hay que modificar nada de su implementación, pero en Jena habría que actualizar los filtros de las consultas SPARQL lo que hace perder en dinamismo a la hora de comparar ambos agentes ontológicos.

También es destacable como el cambio de tendencia en el tiempo de clasificación en el caso de Jena ya que al subir el número de reglas a 50, el comportamiento lineal que tenía su tiempo de CPU pasa a ser exponencial.

Si comparamos además el ratio que ocupa cada paciente en la ontología con lo que ocupa en una base de datos normal sin ningún tipo de regla nos encontramos que para un caso base de 5 propiedades por paciente, una base de datos normal ocupa 112 bytes por paciente y en la ontología es casi 8 veces mayor, 801 bytes por paciente.

Como líneas futuras de trabajo, sería interesante desarrollar ontologías con mayor número de reglas para comprobar si en algún momento el tiempo de búsqueda es mayor en Jena que en OWLAPI.

También se debería intentar desarrollar un motor de razonamiento lo suficientemente potente en Jena que permitiese inferir conocimiento con este agente ontológico y así hacer una comparación en cuanto a la eficiencia de los motores.

Por otro lado sería interesante, extender la ontología no solo distinguir entre enfermo y sano, sino ser capaz de dar posibles enfermedades o sugerir medicamentos que sean capaces de paliar los síntomas del enfermo.

Apéndice A

Presupuesto

En este apéndice se analiza el desarrollo de este proyecto en términos económicos.

A.1 Tareas

En esta primera sección se muestra una estimación de la duración de las distintas tareas llevadas a cabo durante el desarrollo de este proyecto. Dichas tareas no se han realizado de forma continua ya que el desarrollo del proyecto se ha compaginado con estudios y becas.
En la tabla 1 se resumen las tareas realizadas junto con el número de horas estimadas para su realización.

Tarea	Nº de horas
Investigación documentación y análisis	50
Configuración del entorno de desarrollo (Protegé Eclipse)	20
Toma de contacto con OWLAPI y Jena	80
Desarrollo de la ontología	100
Desarrollo de PlantaHospitalJena	100
Desarrollo de PlantaHospitalOWL	200
Desarrollo de la interfaz gráfica	90
Integración del sistema completo	30
Pruebas y validación del sistema	100
Redacción de la memoria	160

Tabla 1: Tareas del proyecto y horas empleadas

A..2. Costes

Como se ha visto en la tabla A.1, la duración del proyecto ha sido de 930 horas, que asumiendo 8 horas de trabajo al día equivalen a 117 días laborables aproximadamente con lo que podemos asumir que si el proyecto lo realizara una persona a tiempo completo se habría podido terminar aproximadamente en cuatro meses.

A.2.1. Personal

Para la realización del proyecto se ha necesitado cubrir las tareas que realizarían los especialistas que se detallan en la tabla 2, junto con el precio de la mano de obra por hora. El jefe de proyecto sería el tutor del mismo.

Cargo	Coste (€/hora)
Analista	35
Diseñador	40
Gestión de calidad y pruebas	35
Jefe de proyecto	45
Programador	25

Tabla 2: Salarios de especialistas

Si estos costes se aplican al número de horas dedicado por cada especialista en función de las actividades y las tareas encomendadas, se puede calcular el coste total del personal requerido para este proyecto, a través de la tabla 3. El coste del personal asciende a 29.875€.

Tarea	Analista	Diseñador	Gestión de calidad	Jefe de proyecto	Programador
Investigación documentación y análisis	40			10	
Configuración del entorno de desarrollo (Protegé Eclipse)					20
Toma de contacto con OWLAPI y Jena		20		10	50
Desarrollo de la ontología		30	10	10	50
Desarrollo de PlantaHospitalJena		25	5	10	60
Desarrollo de PlantaHospitalOWL		25	20	10	145
Desarrollo de la interfaz gráfica		60	10	5	15
Integración del sistema completo		5	10	5	10
Pruebas y validación del sistema			50	10	40
Redacción de la memoria	40	20	20	20	60
Suma de horas	80	185	125	90	450
Coste total	**29.875€**				

Tabla 3: Coste de personal

A.2.2. Material

En este apartado se detalla el coste de los materiales empleados durante la realización del proyecto. Estos costes se pueden dividir en equipo y licencias.

A.2.2.1. Equipo

El equipo utilizado consta de dos componentes principales:
Ordenador personal con monitor de 27", procesador de cuatro núcleos y 8GB de memoria RAM. Valorado en 1.000€.

A.2.2.2. Licencias

En la tabla 4 se analiza el coste de las licencias que se han necesitado en la elaboración del proyecto.

Cargo	Licencias	Coste/licencia	Coste
Windows 8	1	150	150
Eclipse IDE	1	0	0
Protegé	1	0	0
Microsoft Office	1	130	130
Total	**280 €**		

Tabla 4: Coste de licencias

Es coste total en licencias es de 280 €

A.2.3. Costes indirectos

En la tabla 5 se resumen el resto de costes, derivados del uso de instalaciones durante los 4 meses que duraría el proyecto

Concepto	Coste (€)
Alquiler del local	1200
Luz y agua	200
Servicio de limpieza	150
Teléfono fijo y conexión a internet	200
Seguro a todo riesgo	120
Total	**1.870 €**

Tabla 5: Costes indirectos

A.3. Resumen

Teniendo en cuenta todos los costes relatados anteriormente, el coste total necesario para realizar el proyecto asciende a C, tal y como se muestra en la tabla 6.

Concepto	Coste (€)
Mano de obra	29.875
Equipo	1.000
Licencias	280
Costes indirectos	1.870
Total antes de impuestos	**32.125 €**
Total (21% IVA)	**38.871,25 €**

Tabla 6: Coste total

Teniendo en cuenta los costes desglosados en los apartados anteriores, el presupuesto total de este proyecto asciende a la cantidad de **TREINTA Y OCHO MIL OCHOCIENTOS SETENTA Y UNO CON VEINTICINCO** euros.

Leganés 29 de septiembre de 2013

El ingeniero proyectista

Fdo. Jose Bañón Peñuelas

Apéndice B

Manual para crear una ontología

Este apéndice se ha extraido de la página de la universidad de Standford y adaptado a este texto. En él, se explica todos y cada uno de los pasos necesarios para crear una ontología. Se ha incluido para aclarar la estructura de la ontología desarrollada y para que sirva de ayuda en caso de que algún lector decida implementar su propia ontología.

B.1 Una simple metodología de ingeniería del conocimiento

Para desarrollar una ontología, no existe ni una sola forma o ni una sola metodología "correcta". Aquí abordamos los puntos generales que deben ser tomados en consideración y ofrecemos uno de los procedimientos posibles para desarrollar una ontología. Describimos un enfoque iterativo en el desarrollo de la ontología: comenzamos por abordar la ontología de manera frontal. A continuación volvemos sobre la ontología, que consideramos en proceso de evolución, afinándola y completándola con detalles. A lo largo de este proceso discutimos las decisiones de modelización que toma el diseñador, así como los pros, los contras y las implicaciones de diferentes soluciones.
Inicialmente, queremos enfatizar algunas reglas fundamentales en el diseño de ontologías a las cuales nos referiremos varias veces. Esas reglas pueden parecer algo dogmáticas. Ellas pueden ayudar, sin embargo, para tomar decisiones de diseño en muchos casos.

No hay una forma correcta de modelar un dominio - siempre hay alternativas viables. La mejor solución casi siempre depende de la aplicación que tienes en mente y las extensiones que anticipas.

El desarrollo de ontologías es un proceso necesariamente iterativo.

Los conceptos en la ontología deben ser cercanos a los objetos (físicos o lógicos) y relaciones en tu dominio de interés. Esos son muy probablemente sustantivos (objetos) o verbos (relaciones) en oraciones que describen tu dominio.

Es decir, decidir para qué vamos a usar la ontología y cuán detallada o general será la ontología guiará a muchas de las decisiones de modelamiento a lo largo del camino. Entre las varias alternativas viables, necesitaremos determinar cuál funcionará mejor para la tarea proyectada, cuál será más intuitiva, más extensible y más mantenible. Necesitamos también recordar que una ontología es un modelo de la realidad del mundo y los conceptos en la ontología deben reflejar esta realidad. Después de que hayamos

definido una versión inicial de la ontología, podemos evaluarla y depurarla usándola en aplicaciones o métodos que resuelvan problemas o discutiéndola con expertos en el área. En consecuencia, casi seguramente necesitaremos revisar la ontología inicial. Este proceso de diseño iterativo probablemente continuara a través del ciclo de vida entero de la ontología.

B.1.1 Paso 1. Determinar el domino y alcance de la ontología

Sugerimos comenzar el desarrollo de una ontología definiendo su dominio y alcance. Es decir, responder a varias preguntas básicas:

- ¿Cuál es el dominio que la ontología cubrirá?
- ¿Para qué usaremos la ontología?
- ¿Para qué tipos de preguntas la información en la ontología deberá proveer respuestas?
- ¿Quién usará y mantendrá la ontología?

Las respuestas a esas preguntas pueden cambiar durante el proceso del diseño de la ontología, pero en cualquier momento dado ellas ayudarán a limitar el alcance del modelo.

Consideremos la ontología de vinos y alimentos que se introdujo antes. El dominio de la ontología es la representación de alimentos y vinos. Planeamos usar esta ontología en las aplicaciones que sugieran buenas combinaciones de vinos y alimentos.

Naturalmente, los conceptos que describen diferentes tipos de vinos, tipos principales de alimentos, la noción de una buena combinación de vino y alimento y la mala combinación figurarán en nuestra ontología. Al mismo tiempo, es improbable que la ontología incluya conceptos para gestionar inventarios en un establecimiento vinícola o empleados en un restaurante aunque esos conceptos están de alguna manera relacionados a las nociones de vino y alimento.

Si la ontología que estamos diseñando será usada para ayudar en el procesamiento de lenguaje natural de artículos en las tiendas de vino, sería importante incluir sinónimos e información de las varias clases de palabras a las cuales una palabra puede ser asignada para los conceptos de la ontología. Si la ontología será usada para ayudar a los clientes de un restaurante a decidir qué vino ordenar, necesitamos incluir información del precio de venta al por menor. Si es usada por compradores de vino que almacenan el vino en bodegas, el precio de venta al por mayor y la disponibilidad serán necesarios. Si la gente que mantendrá la ontología describe el dominio en un lenguaje que es diferente del lenguaje que usan los usuarios de la ontología, tendremos que proveer el mapeo entre los lenguajes.

Preguntas de competencia

Una de las formas de determinar el alcance de la ontología es bosquejando una lista de preguntas que la base de conocimientos basada en la ontología debería ser capaz de responder, preguntas de competencia [30]. Esas preguntas servirán después como prueba de control de calidad: ¿La ontología contiene suficiente información para responder esos tipos de preguntas? ¿Las respuestas requieren un nivel particular de detalle o representación de un área particular? Las preguntas de competencia son solamente un bosquejo y no necesitan ser exhaustivas.

En el dominio de los vinos y alimentos, las siguientes preguntas son posibles preguntas de competencia:

- ¿Qué características debo considerar cuando elijo un vino?
- ¿Bordeaux es un vino rojo o blanco?
- ¿El Cabernet Sauvignon va bien con comida de mar?
- ¿Cuál es la mejor elección de vino para acompañar carne asada?
- ¿Qué características de un vino afectan su idoneidad con un pescado?
- ¿El cuerpo o aroma de un vino específico cambia con su año de cosecha?
- ¿Cuáles fueron buenas cosechas para el Napa Zinfandel?

Juzgando a partir de esta lista de preguntas, la ontología incluirá la información de varias características de vinos y tipos de vinos, años de cosechas (buenos y malos), clasificación de alimentos que importan para elegir un vino apropiado, combinaciones recomendadas de vinos y comidas.

B.1.2 Paso 2. Considerar la reutilización de ontologías existentes

Casi siempre vale la pena considerar lo que otra persona ha hecho y verificar si podemos refinar y extender recursos existentes para nuestro dominio y tarea particular. Reusar ontologías existentes puede ser un requerimiento si nuestro sistema necesita interactuar con otras aplicaciones que ya se han dedicado a ontologías particulares o vocabularios controlados. Muchas ontologías ya están disponibles en forma electrónica y pueden ser importadas dentro de un entorno de desarrollo de ontologías que estás usando. El formalismo en el cual una ontología esta expresado a menudo no interesa, puesto que muchos sistemas de representación de conocimiento pueden importar y exportar ontologías. Aun si el sistema de

representación de conocimiento no puede funcionar directamente con un formalismo particular, la tarea de traducir una ontología a partir de un formalismo a otro no es usualmente difícil.

Hay bibliotecas de ontologías reusables en la Web y en la literatura. Por ejemplo, podemos usar la biblioteca de ontologías Ontolingua [31] o la biblioteca de ontologías DAML [32]. También hay un cierto número de ontologías comerciales públicamente disponibles como www.unspsc.org, www.rosettanet.org o www.dmoz.org.

Por ejemplo, es posible que una base de conocimientos sobre vinos Franceses exista. Si podemos importar esta base de conocimiento y la ontología sobre la cual está basada, tendremos no solamente la clasificación de vinos Franceses sino también el primer paso hacia la clasificación de características de vinos usadas para distinguir y describir los vinos. Es posible que listas con las propiedades de los vinos estén disponibles en sitios Web comerciales tales como www.wines.com que los clientes consideren útiles para comprar vinos.

En esta guía, sin embargo, asumiremos que no existe ninguna ontología relevante y comenzaremos la ontología desde el principio.

B.1.3 Paso 3. Enumerar términos importantes para la ontología

Es útil escribir una lista con todos los términos con los que quisiéramos hacer enunciados o dar explicación a un usuario. ¿Cuáles con los términos de los cuales quisiéramos hablar?

¿Qué propiedades tienen esos términos? Por ejemplo, términos importantes relativos a los vinos incluirán vino, cepaje, establecimiento vinícola, localidad, color del vino, cuerpo, sabor, contenido de azúcar; diferentes tipos de alimentos, tales como pescado y carne roja; subtipos de vino tales como vino blanco, etc. Inicialmente, es importante obtener una lista integral de términos sin preocuparse del recubrimiento entre los conceptos que representan, relaciones entre los términos, o cualquier propiedad que los conceptos puedan tener, o si los conceptos son clases o slots.

Los siguientes dos pasos (desarrollando la jerarquía de clases y definiendo las propiedades de los conceptos (slots)) están estrechamente relacionadas. Es difícil hacer primero uno de ellos y luego hacer el otro. Típicamente, creamos unas cuantas definiciones de los conceptos en la jerarquía y luego continuamos describiendo las propiedades de esos conceptos y así sucesivamente. Esos dos pasos son también los más importantes en el proceso de diseño de la ontología. Los describiremos brevemente y dedicaremos las siguientes dos secciones a discutir los asuntos más complicados que necesitan ser considerados, peligros comunes, decisiones a tomar, etc.

B.1.4 Paso 4. Definir las clases y la jerarquía de clases

Hay varios posibles enfoques para desarrollar una jerarquía de clases (Uschold and Gruninger 1996):

Un proceso de desarrollo *top-down* comienza con la definición de los conceptos más generales en el dominio la subsecuente especialización de los conceptos. Por ejemplo, podemos comenzar creando clases para los conceptos generales de Vino y Alimentos. Luego especializamos la clase Vino creando algunas de sus subclases: Vino blanco, Vino rojo, Vino rosado. Podemos posteriormente categorizar la clase Vino rojo en, por ejemplo, Syrah, Borgoña, Cabernet Sauvignon, etc.

Un proceso de desarrollo *bottom-up* comienza con la definición de las clases más específicas, las hojas de la jerarquía, con el subsecuente agrupamiento de esas clases en conceptos más generales. Por ejemplo, comenzamos definiendo clases para los vinos Pauillac y Margaux. Luego creamos una superclase común para esas dos clases (Medoc) la cual a su vez es una subclase de Bordeaux.

Un proceso de desarrollo combinado es el resultado de una combinación de los enfoques *top-down* y *bottom-up*: primero definimos los conceptos más sobresalientes y luego los generalizamos y especializamos apropiadamente. Podríamos comenzar con unos cuantos conceptos de nivel superior como Vino, y unos conceptos específicos, como Margaux. Podemos luego relacionarlos en un concepto de nivel medio, tal como Medoc. Podríamos luego desear generar todas las clases de vino regional de Francia, generando en consecuencia un cierto número de conceptos de nivel medio.
La Figura 67 muestra una posible descomposición entre los diferentes niveles de generalidad. Ninguno de esos tres métodos es inherentemente mejor que cualquiera de los otros. El enfoque a tomar depende fuertemente de la visión personal del dominio. Si un desarrollador tiene una visión sistemática top-down del dominio, entonces será más fácil usar el enfoque top-down. El enfoque combinado es a menudo el más fácil para muchos desarrolladores de ontologías, puesto que los "conceptos del medio" tienden a ser conceptos más descriptivos en el dominio [33].
Si tiendes a pensar en vinos distinguiendo primero la clasificación más general, entonces el enfoque top-down podría funcionar mejor para ti. Si prefieres comenzar listando ejemplos específicos, el enfoque bottom-up podría ser el más apropiado.
Sea cual sea el enfoque que elijamos, usualmente comenzaremos definiendo las clases. De la lista creada en el Paso 3, seleccionamos los términos que describen objetos que tienen existencia independiente en lugar de términos que describen esos objetos.

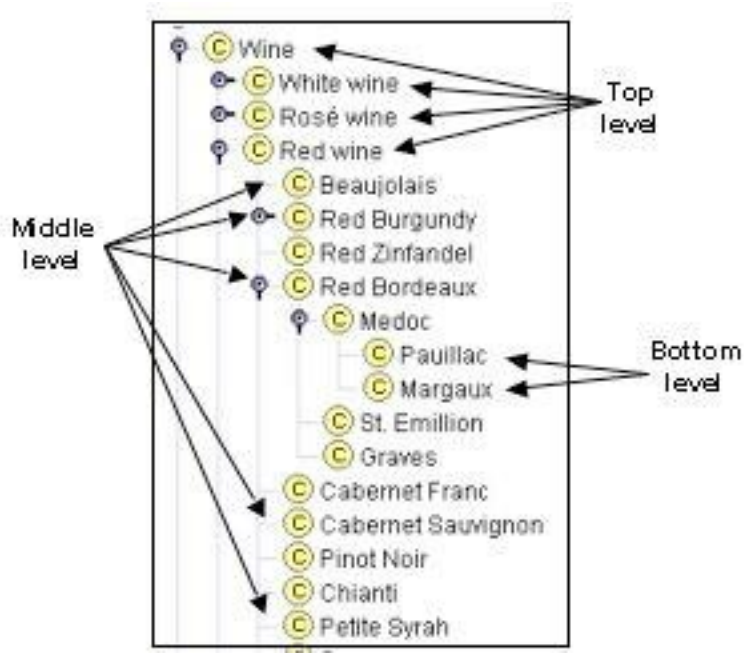

Figura 67: Diferentes niveles de la taxonomía de los Vinos

Esos términos serán las clases de la ontología y llegarán a ser anclas en la jerarquía de clases. Organizamos las clases en una taxonomía jerárquica preguntando si siendo una instancia de una clase, el objeto necesariamente será (i.e., por definición) una instancia de alguna otra clase.

Si una clase A es una superclase de la clase B, entonces cada instancia de B lo es también de A.

En otras palabras, la clase B representa un concepto que es un "tipo de" A.
Por ejemplo, cada vino Pinot Noir es necesariamente un vino rojo. Por lo tanto, la clase Pinot Noir es una subclase de la clase Vino Rojo.
La Figura 2 muestra una parte de la jerarquía de clases de la ontología de Vinos. La sección 4 contiene una discusión detallada de algunos aspectos a considerar cuando se está definiendo una jerarquía de clases.

B.1.5 Paso 5. Definir las propiedades de las clases: slots

Las clases aisladas no proveerán suficiente información para responder las preguntas de competencia del Paso 1. Una vez que hemos definido algunas de las clases, debemos describir la estructura interna de los conceptos.
Ya hemos seleccionado clases de la lista de términos creada en el Paso 3. La mayoría de los términos restantes son muy probablemente propiedades de esas clases. Esos términos incluyen, por ejemplo, un color de vino, cuerpo, sabor, contenido de azúcar y localización de un establecimiento vinícola.

Para cada propiedad en la lista, debemos determinar qué clase es descrita por la propiedad. Esas propiedades se convierten en slots adosados a las clases. De esta forma, la clase Vino tendrá los siguientes slots: color, sabor, y azúcar. Y la clase Establecimiento vinícola tendrá un slot localización.
En general, hay varios tipos de propiedades de objeto que pueden llegar a ser slots en una ontología:

- propiedades "intrínsecas" tales como el sabor de un vino;
- propiedades "extrínsecas" tales como el nombre de un vino, y el área de donde proviene;
- partes, si el objeto es estructurado; pueden ser "partes" físicas y abstractas (ej., los platos de una comida)
- relaciones con otros individuos; éstas son las relaciones entre miembros individuales de una clase y otros ítems (ej., el productor de vino, representando una relación entre un vino y un establecimiento vinícola, y la uva con la cual el vino está producido.)

De esta forma, además de las propiedades que hemos identificado previamente, necesitamos añadir los siguientes slots a la clase Vino: nombre, área, productor, cepaje. La

Figura 68 muestra los slots para la clase Vino. Todas las subclases de una clase heredan los slots de esa clase. Por ejemplo, todos los slots de la clase Vino serán heredados por todas las subclases de Vino, que incluyen Vino Rojo y Vino Blanco. Agregaremos un slot adicional, nivel de tanino (bajo, moderado, o alto), a la clase Vino Rojo. El slot nivel de tanino será heredado por todas las clases que representan vinos rojos (tales como Bordeaux y Beaujolais).
Un slot deberá estar adosado a la clase más general que pueda tener esa propiedad. Por ejemplo, el cuerpo y color de un vino deberán estar adosados a la clase Vino, puesto que ésta es la clase más general cuyas instancias tendrán un cuerpo y un color.

Template Slots

Name	Type	Cardinality	Other Facets
body	Symbol	single	allowed-values={FULL,MEDIUM,LIGHT}
color	Symbol	single	allowed-values={RED,ROSÉ,WHITE}
flavor	Symbol	single	allowed-values={DELICATE,MODERATE,STRONG}
grape	Instance	multiple	classes={Wine grape}
maker	Instance	single	classes={Winery}
name	String	single	
sugar	Symbol	single	allowed-values={DRY,SWEET,OFF-DRY}

Figura 68: Slots de la clase Vino

B.1.6 Paso 6. Definir las facetas de los slots

Los slots puedes tener diferentes facetas que describen el tipo de valor, valores admitidos, el número de los valores (cardinalidad), y otras características de los valores que los slots pueden tomar. Por ejemplo, el valor del slot nombre (como en "el nombre de un vino") es una cadena de caracteres. Es decir, nombre es un slot con String como tipo de valor. El slot produce (como en "un establecimiento vinícola produce tales vinos") puede tener valores múltiples y los valores son instancias de la clase Vino. Es decir, produce es un slot con Instance como tipo de valor y Vino como clase admitida.
Describiremos ahora varias facetas comunes.

Cardinalidad del slot

La cardinalidad de un slot define cuantos valores un slot puede tener. Algunos sistemas solamente distinguen entre cardinalidad simple (admitiendo a lo sumo un valor) y cardinalidad múltiple (admitiendo cualquier cantidad de valores). Los vinos producidos por un establecimiento vinícola particular completan el slot produce que es de cardinalidad múltiple para la clase Establecimiento vinícola.
Algunos sistemas admiten la especificación de una cardinalidad mínima y máxima para describir la cantidad de valores de un slot con más precisión. Una cardinalidad mínima N significa que un slot debe tener al menos N valores. Por ejemplo, el slot cepaje de un Vino tiene una cardinalidad mínima de 1: cada vino esta hecho de al menos una variedad de uva. Una cardinalidad máxima M siginifica que un slot puede tener a lo sumo M valores. La cardinalidad máxima para el slot cepaje para vinos de simple variedad de uva es 1: esos vinos son hechos de solamente una variedad de uva. Algunas veces puede ser útil fijar la máxima cardinalidad en 0. Esta definición indicaría que el slot no puede tener ningún valor particular para una subclase particular.

Tipo de valor de los slots

Una faceta tipo de valor describe qué tipos de valores pueden llenar el slot. Aquí está una lista de los tipos de valores más comunes:

String es el tipo de valor más simple el cual es usado por slots tales como nombre: el valor es una simple cadena de caracteres

Number (algunas veces los tipos de valores Float e Integer son usados por ser más específicos) describe slots con valores numéricos. Por ejemplo, el precio que un vino puede tener es un tipo de valor Float.

Los slots del tipo *Boolean* son simples banderas si/no. Por ejemplo, si elegimos no representar vinos espumantes como una clase separada, que el vino sea espumante o no, puede ser representado como un valor de un slot Boolean: si el valor es "true" ("si") el vino es espumante y si el valor es "false" ("no") el vino no es espumante.

Los slots del tipo *Enumerated* especifican una lista específica de valores admitidos para el slot. Por ejemplo, podemos especificar que slot sabor puede tomar uno de los siguientes valores posibles: fuerte, moderado y delicado. En Protégé-2000 los slots enumerados son del tipo Symbol.

Los slots del tipo *Instance* admiten la definición de relaciones entre individuos. Los slots con tipo de valor Instance deben también definir una lista de clases admitidas de las cuales las instancias pueden provenir. Por ejemplo, el slot produce de la clase Establecimiento vinícola puede tener instancias de la clase Vino como sus valores.

La Figura 69 muestra la definición del slot produce en la clase Establecimiento vinícola (Winery). El slot tiene cardinalidad múltiple, Instance como tipo de valor, y la clase Vino (Wine) como clase admitida para sus valores.

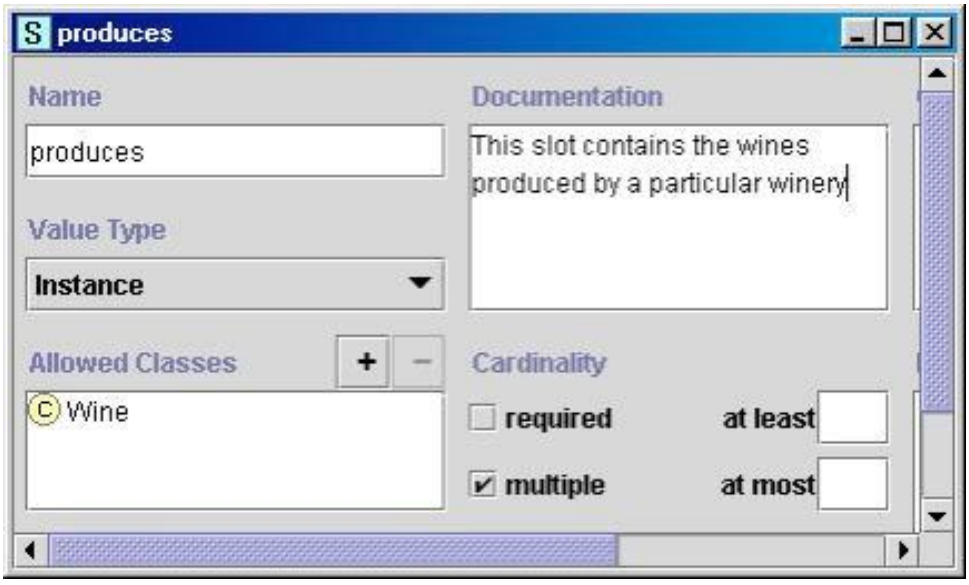

Figura 69: Definición del slot "produces"

Dominio y rango de un slot

Las clases admitidas para los slots de tipo Instance son a menudo llamadas rango de un slot. En el ejemplo de la Figura 4, la clase Vino es el rango del slot produce. Algunos sistemas permiten restringir el rango de un slot cuando el slot esta adosado a una clase particular.

Las clases a las cuales un slot está adosado o las clases cuyas propiedades son descritas por un slot son llamadas dominio del slot. La clase Establecimiento vinícola es el dominio del slot produce. En los sistemas en los cuales adosamos slots a las clases, las clases a las cuales el slot es adosado usualmente constituye el dominio del slot. No hay necesidad de especificar el dominio separadamente.
Las reglas básicas para determinar un dominio y un rango de un slot son similares:

Cuando se define un dominio o rango de un slot, se debe encontrar las clases o clase más generales que puedan ser respectivamente el dominio o rango de los slots. Por otro lado, no definir un dominio ni rango que sea demasiado general: todas las clases en el dominio de un slot deben ser descritas por el slot y las instancias de todas las clases en el rango de un slot deben poder ser rellenos potenciales del slot. No elegir una clase demasiado general para el rango (i.e., es inútil de crear un rango COSA (THING)) pero es posible elegir una clase que cubre todos los valores de relleno.
En términos más específicos:

- Si una lista de clases que definen un rango o un dominio de un slot incluyen una clase y sus subclases, remover la subclase.

Si el rango de un slot contiene la clase Vino y la clase Vino Rojo, podemos remover Vino Rojo del rango porque no añade nueva información: El Vino Rojo es una subclase de Vino y por lo tanto el rango del slot ya lo incluye implícitamente como también todas las otras subclases de la clase Vino.

- Si una lista de clases que definen un rango o dominio de un slot contiene todas las subclases de la clase A, pero no la clase A es sí, el rango debería contener solamente la clase A y no las subclases.

En lugar de definir el rango del slot para incluir Vino Rojo, Vino Blanco y Vino Rosado (enumeración de todas las subclases directas de Vino), podemos limitar el rango a la clase Vino como tal.

- Si una lista de clases que definen un rango o dominio de un slot contiene unas cuantas subclases de la clase A, considerar si la clase A daría una definición de rango más apropiada.

En sistemas en los cuales adosar un slot a una clase es lo mismo que agregar la clase al dominio del slot, las mismas reglas se aplican al adosado del slot: Por un lado, debemos tratar de hacerla tan general como sea posible. Por otro lado, debemos asegurar que cada clase a la cual adosamos el slot pueda en efecto tener la propiedad que el slot representa. Podemos adosar el slot del nivel de tanino a cada clase que representa a los vinos rojos (ej., Bordeaux, Merlot, Beaujolais, etc.). Sin embargo, puesto que todos los

vinos rojos tienen la propiedad nivel de tanino, deberíamos adosar en su lugar el slot a esta clase más general de Vinos Rojos. Si adicionalmente, generalizamos el dominio del slot del nivel de tanino (adosándolo en su lugar a la clase Vino) no sería correcto puesto que no usamos el nivel de tanino para describir por ejemplo a los vinos blancos.

B.1.7 Paso 7. Crear instancias

El último paso consiste en crear instancias individuales de clases en la jerarquía. La definición de una instancia individual de una clase requiere (1) elegir una clase, (2) crear una instancia individual de la clase y (3) rellenar los valores del slot. Por ejemplo, podemos crear una instancia individual ChateauMorgonBeaujolais para representar un tipo específico de vino Beaujolais. ChateauMorgonBeaujolais es una instancia de la clase Beaujolais que representa a todos los vinos Beaujolais. Esta instancia tiene definidos los siguientes valores de slot (ver
Figura 70).

- Cuerpo: Ligero
- Color: Rojo
- Aroma: Delicado
- Nivel de tanino: Bajo
- Cepaje: Gamay (instancia de la clase Uva)
- Productor: ChateauMorgon (instancia de la clase Establecimiento vinícola)
- Región: Beaujolais (instancia de la clase Región-Vino)
- Azúcar: Seco

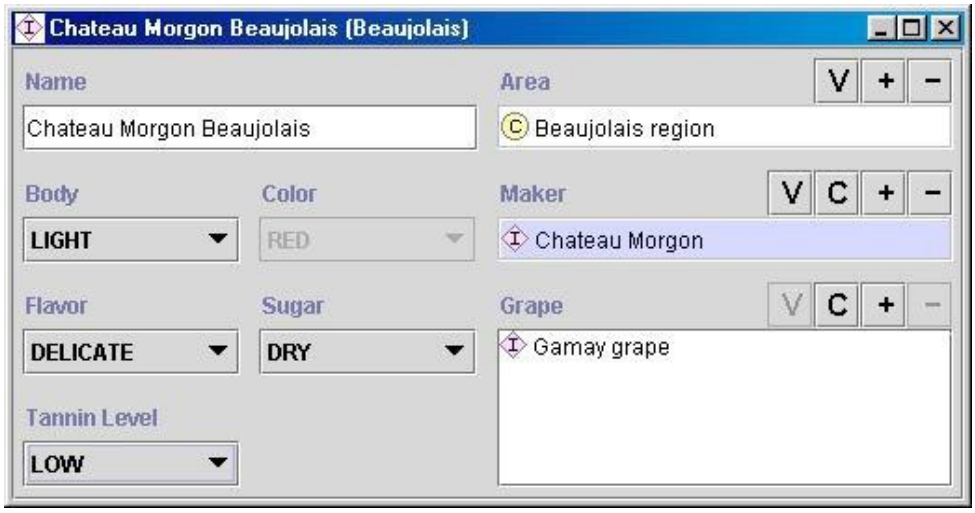

Figura 70: Definición de una instancia de la clase Beaujolais

B.2 Definición de las clases y de la jerarquía de clases

Esta sección discute aspectos en los cuales hay que tener cuidado y errores que son fáciles de cometer cuando se definen clases y jerarquías de clases. Como lo mencionamos antes, no hay una jerarquía de clases correcta para un dominio dado. La jerarquía depende de los posibles usos de la ontología, el nivel de detalle que es necesario para la aplicación, preferencias personales, y algunas veces requerimientos de compatibilidad con otros modelos. Sin embargo, discutiremos varias recomendaciones para tenerlas en cuenta cuando se desarrolle una jerarquía de clases. Después de haber definido un número considerable de nuevas clases, es útil detenerse y verificar si la jerarquía emergente va de acuerdo a esas recomendaciones.

B.2.1 Asegurarse que la jerarquía de clases es correcta

Una relación "is-a"

La jerarquía de clases representa una relación "is-a" ("es-un, es-una"): una clase A es una subclase de B si cada instancia de B es también una instancia de A. Por ejemplo, Chardonnay es una subclase de Vino Blanco. Otra forma de pensar en la relación taxonómica es viéndola como una relación "kind-of" ("tipo-de"): Chardonnay es un tipo de Vino Blanco. Un avión comercial es un tipo de avión. Carne es un tipo de alimento. En definitiva, una subclase de una clase representa un concepto que es un "tipo de" concepto que la superclase representa.

Por otro lado un simple vino no es una subclase de todos los vinos. Un error común de modelamiento es el de incluir una versión singular y plural del mismo concepto en la jerarquía haciendo esta anterior una subclase de la última. Por ejemplo, está mal definir una clase Vinos y una clase Vino como una subclase de Vinos. Cuando piensas en la jerarquía como representación de la relación "kind-of" ("tipo-de"), el error de modelamiento se hace claro: un simple Vino no es un tipo de Vinos. La mejor forma de evitar ese tipo de error es utilizando siempre la forma singular o plural al nombrar las clases (ver la Sección 6 sobre la discusión del nombrado de conceptos).

Transitividad de las relaciones jerárquicas

Una relación de subclase es transitiva:

- Si B es una subclase de A y C es una subclase de B, entonces C es una subclase de A

Por ejemplo, podemos definir la clase Vino, y luego definir la clase Vino Blanco como una subclase de Vino. Luego definimos una clase Chardonnay como una subclase de Vino Blanco. La transitividad de la relación de subclase significa que la clase Chardonnay es también una subclase de Vino. Algunas veces hacemos la distinción entre subclases directas y subclases indirectas. Una **subclase directa** es la subclase "más cercana" de la clase: no hay clases entre la clase y sus subclases directas en la jerarquía. Es decir, no hay otras clases en la jerarquía entre la clase y su superclase directa. En nuestro ejemplo, Chardonnay es una subclase directa de Vino Blanco y no es una subclase directa de Vino.

Evolución de una jerarquía de clases

Mantener una jerarquía consistente de clases puede llegar a ser desafiante a media que el dominio evoluciona. Por ejemplo, por muchos años, todos los vinos Zinfandel fueron rojos. Por lo tanto, definimos una clase de vinos Zinfandel como una subclase de la clase Vino Rojo. Algunas veces, sin embargo, los productores comenzaron a presionar las uvas y extraer inmediatamente los elementos de las uvas que producen color, modificando en consecuencia el color del vino resultante. Por lo tanto, obtenemos "zinfandel blanco" cuyo color es rosado. Ahora necesitamos dividir la clase Zinfandel en dos clases de zinfandel (Zinfandel Blanco y Zinfandel Rojo) y clasificarlos como subclases de Vino Rosado y Vino Rojo respectivamente.

Las clases y sus nombres

Es importante distinguir entre una clase y su nombre, las clases representan conceptos en el dominio y no las palabras que denotan esos conceptos.

El nombre de una clase puede cambiar si elegimos una terminología diferente, pero el término como tal representa la realidad objetiva en el mundo. Por ejemplo, podemos crear una clase Camarón y luego renombrarla a Gamba (la clase aun representa el mismo concepto). Combinaciones apropiadas de vinos que hacen referencia a platos con camarones deben hacer referencia a platos con gambas. En términos más prácticos, la siguiente regla siempre debe ser seguida:

Los sinónimos para el mismo concepto no representan clases diferentes.

Los sinónimos son solo nombres diferentes para un concepto o término. Por lo tanto, no deberíamos tener una clase llamada Camarón y una clase llamada Gamba, y posiblemente una clase llamada Crevette. En su lugar, hay una clase, llamada Camarón o Gamba. Muchos sistemas admiten la asociación de una lista de sinónimos, traducciones, o nombres de presentación con una clase. Si un sistema no permite estas asociaciones, los sinónimos siempre podrían ser listados en la documentación de la clase.

Evitar ciclos en las clases

Debemos evitar ciclos en la jerarquía de clases. Se dice que hay un ciclo en una jerarquía cuando una clase A tiene una subclase B y al mismo tiempo B es una superclase de A. Crear un ciclo como ese en un jerarquía equivale a declarar que las clases A y B son equivalentes: todas las instancias de A son instancias de B y todas las instancias de B son también instancias de A.

En efecto, puesto que B es una subclase de A, todas las instancias de B deben ser instancias de la clase A. Puesto que A es una subclase de B, todas las instancias de A deben también ser instancias de la clase B.

B.2.2 Análisis de clases hermanas en la jerarquía de clases

Clases hermanas en una jerarquía de clases

Las clases hermanas en una jerarquía son clases que son subclases directas de la misma clase. Todas las clases hermanas en una jerarquía (excepto para las que están al nivel de la raíz) deben estar al mismo nivel de generalidad.

Por ejemplo, Vino Blanco y Chardonnay no deberían ser clases de la misma clase (digamos Vino). Vino Blanco es un concepto más general que Chardonnay. Las clases hermanas deben representar conceptos que caen “en la misma línea” de la misma forma que las secciones de un mismo nivel en un libro están al mismo nivel de generalidad. En ese sentido, los requerimientos para una jerarquía de clases son similares a los requerimientos para una estructuración de un libro. Sin embargo, los conceptos en la raíz

de la jerarquía (los cuales son a menudo representados como subclases directas de alguna clase muy general, como Thing (Cosa)) representan divisiones principales del dominio y no tienen que ser conceptos similares.

¿Cuán mucho es demasiado y cuán poco es insuficiente? No hay reglas que digan el número de subclases directas que una clase debería tener. Sin embargo, varias ontologías bien estructuradas tienen entre dos y una docena de subclases directas. Por lo tanto, consideremos las siguientes reglas:

Si una clase tiene solamente una subclase directa, puede existir un problema de modelamiento o sino la ontología no está completa. Si hay más de una docena de subclases para una clase dada, entonces categorías intermedias adicionales pueden ser necesarias.

La primera de las dos reglas es similar a la regla de composición tipográfica en la que las listas con viñetas nunca deberían tener solamente una viñeta. Por ejemplo, la mayoría de los vinos rojos de Borgoña son vinos Côtes d'Or. Supongamos que queremos representar solamente este tipo de mayoritario de vinos de Borgoña. Podríamos crear una clase Borgoña Rojo y luego una simple subclase Côtes d'Or. Sin embargo, si en nuestra representación los vinos rojo de Borgoña y Côtes d'Or son esencialmente equivalentes (todos los vinos rojos de Borgoña son Côtes d'Or y todos los vinos Côtes d'Or son rojos de Borgoña), la creación de la clase Côtes d'Or no es necesaria ya que no adiciona nueva información a la representación. Si deberíamos incluir los vinos Côtes Chalonnaise, los cuales son vinos de Borgoña más baratos de la región sur de Côtes d'Or, entonces crearíamos dos subclases de la clase Borgoña: Côtes d'Or y Côtes Chalonnaise (Figura 71).
Supongamos ahora que listamos todos los tipos de vinos como subclases directas de la clase
Vino. Esta lista entonces incluiría tipos más generales de vinos tales como Beaujolais y Bordeaux, como también tipos más específicos de vinos tales como Paulliac y Margaux (Figura 72).

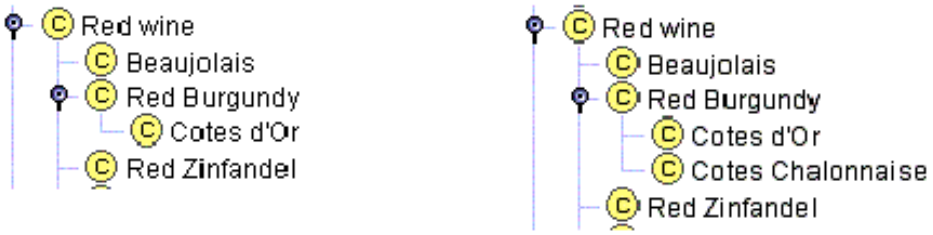

Figura 71: Subclases de la clase VinoRojo

La clase Vino tiene varias subclases directas y, de hecho, para que la ontología refleje los diferentes tipos de vino en una manera más organizada, Medoc debería ser una subclase de Bordeaux y Côtes d'Or

debería ser una subclase de Borgoña. Tener tales categorías intermedias como Vino Rojo y Vino Blanco también reflejaría el modelo conceptual del dominio de vinos que mucha gente tiene (Figura 7).
Sin embargo, si no existen clases naturales para agrupar los conceptos en la larga lista de clases hermanas, no hay la necesidad de crear clases artificiales (dejar las clases en la forma que están). Después de todo, la ontología es un reflejo del mundo real y si no existen categorizaciones en el mundo real, entonces la ontología debería reflejar eso.

B.2.3 Herencia múltiple

La mayoría de los sistemas de representación de conocimiento admiten herencia múltiple en la jerarquía de clases: una clase puede ser subclase de varias clases. Supongamos que deseamos crear una clase separada de vinos de sobremesa, la clase Vino de Sobremesa. El vino Porto es al mismo tiempo vino rojo y vino de sobremesa. Por lo tanto, definimos una clase Porto con dos superclases: Vino Rojo y Vino de Sobremesa. Todas las instancias de la clase Porto serán instancias de la clase Vino Rojo y de la clase Vino de Sobremesa. La clase Porto heredará sus slots y sus facetas de sus dos superclases. De esta forma, ésta heredará el valor DULCE para el slot Azúcar de la clase Vino de Sobremesa y el slot nivel de tanino y el valor para su slot color de la clase Vino Rojo.

B.2.4 ¿Cuándo introducir una nueva clase?

Una de las más difíciles decisiones de tomar durante el modelamiento es cuándo introducir una nueva clase o cuándo representar una desemejanza a través de diferentes valores de propiedades. Es difícil navegar en una jerarquía extremadamente anidada con varias clases raras y en una jerarquía muy plana que tiene pocas clases con mucha información codificada en los slots. Sin embargo, no es fácil encontrar el balance apropiado.
Hay varias reglas de base que ayudan a decidir cuándo introducir nuevas clases en la jerarquía.

Las subclases de una clase usualmente (1) tienen propiedades adicionales que la superclase no tiene, o (2) diferentes restricciones de las de las superclase, o (3) participan en relaciones diferentes que la superclases.

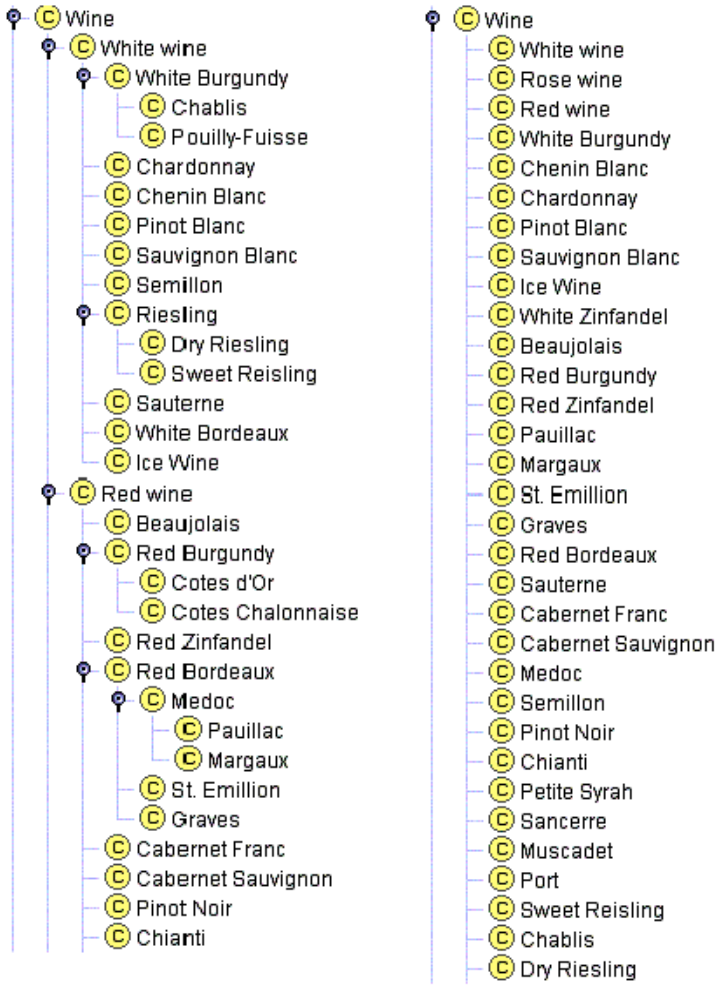

Figura 72: Categorización de vinos

Los vinos rojos pueden tener diferentes niveles de tanino, sin embargo esta propiedad no es usada para describir los vinos en general. El valor para el slot azúcar del Vino de Sobremesa es DULCE, sin embargo no es el caso para la superclase de la clase Vino de Sobremesa. Los vinos Pinot Noir pueden servirse con mariscos mientras que otros vinos rojos no. En otras palabras, introducimos una nueva clase en la jerarquía usualmente solo cuando hay algo hay algo que podamos decir acerca de esta clase que no podamos decir acerca de la superclase. En la práctica, cada subclase debe tener nuevos slots añadidos a ésta, o tener nuevos valores definidos para el slot, o sustituir (override) algunas facetas de los slots heredados. Sin embargo, puede ser útil crear nuevas clases aun cuando no introduzcan nuevas propiedades.

Las clases en terminologías jerárquicas no necesitan introducir nuevas propiedades. Por ejemplo, algunas ontologías incluyen grandes jerarquías de referencia con términos comunes usados en el dominio. Por ejemplo, una ontología que es subyacente a un sistema de registro electrónico médico

puede incluir una clasificación de varias enfermedades. Esta clasificación puede ser solo eso: una jerarquía de términos sin propiedades (o con el mismo conjunto de propiedades). En ese caso, es aún útil organizar los términos en la jerarquía en lugar de en una lista plana porque (1) permitirá una exploración y navegación más fácil y (2) facilitará al médico la fácil elección de un nivel de generalidad del término que es apropiado para la situación. Otra razón para introducir nuevas clases sin nuevas propiedades es para modelar conceptos entre los cuales los expertos del dominio comúnmente hacen una distinción aun cuando no hayamos decidido modelar la distinción en sí. Puesto que usamos las ontologías para facilitar la comunicación entre los expertos de un dominio y entre ellos mismos y los sistemas basados en conocimiento, deseamos reflejar en la ontología la visión del experto sobre el dominio.

Finalmente, no deberíamos crear subclases de una clase para cada restricción adicional. Por ejemplo, introdujimos las clases Vino Rojo, Vino Blanco, y Vino Rosado porque esta distinción es natural en el mundo de los vinos. No introdujimos clases para los vinos delicado, moderado, etc. Cuando definimos una jerarquía de clases, nuestra meta es de encontrar un balance entre crear nuevas clases útiles para la organización de clases y crear demasiadas clases.

B.2.5 ¿Una nueva clase o un valor de propiedad?

Cuando modelamos un dominio, a menudo necesitamos decidir si modelar una distinción específica (como vino blanco, rojo o rosado) como un valor de propiedad o como un conjunto de clases, nuevamente, depende del alcance del dominio y de la tarea en mano.

¿Creamos una clase Vino Blanco o simplemente creamos una clase Vino y llenamos diferentes valores para el slot color?. La respuesta usualmente está en el alcance que hemos definido para la ontología. ¿Qué tan importante es el concepto Vino Blanco en nuestro dominio? Si los vinos tienen solamente importancia marginal en el dominio y si siendo blanco o no el vino no tiene ninguna implicación particular en sus relaciones con otros objetos, entonces no deberíamos introducir una clase separada para los vino blancos. Para un modelo de dominio usado en una factoría que produce etiquetas de vinos, las reglas de las etiquetas de vino de cualquier color son las mismas y la distinción no es importante. Por el contrario, para la representación de vinos, alimentos y sus combinaciones apropiadas, un vino rojo es muy diferente de un vino blanco: está emparejada con diferentes alimentos, tiene diferentes propiedades, y así sucesivamente. De manera similar, el color del vino es importante para la base de conocimientos de vinos que podríamos usar para determinar el orden de los elementos a considerar en la degustación del vino. De esta manera, creamos una clase separada para Vino Blanco.

Si los conceptos con diferentes valores de slot se vuelven restricciones para diferentes slots en otras clases, entonces debemos crear una nueva clase para esta distinción. Caso contrario, representamos la distinción en un valor de slot.

De manera similar, nuestra ontología de vinos tiene clases tales como Rojo Merlot y Blanco Merlot, en lugar de una simple clase para todos los vinos Merlot: los Merlots rojos y los Merlots blancos son realmente vinos diferentes (aunque producidos del mismo cepaje) y si estamos desarrollando una ontología detallada de vinos, esta distinción es importante.

Si la distinción es importante en el dominio y pensamos en los objetos con diferentes valores para la distinción como diferentes tipos de objetos, entonces deberíamos crear una nueva clase para la distinción.

Considerar potenciales instancias individuales de una clase puede ser también útil al decidir si se introduce una nueva clase o no.

Una clase a la cual una instancia individual pertenece no debería cambiar a menudo. Usualmente, cuando usamos propiedades extrínsecas en lugar de intrínsecas de los conceptos para diferenciarlos entre las clases, las instancias de esas clases tendrán que migrar a menudo de una clase a otra. Por ejemplo, Vino Enfriado no debería ser una clase en una ontología que describe las botellas de vino en un restaurante. La propiedad enfriado debería simplemente ser un atributo del vino en una botella puesto que una instancia de Vino Enfriado puede fácilmente dejar de ser una instancia de esta clase y llegar a ser instancia de esta clase de nuevo.

Usualmente, los números, colores, localizaciones son valores de slots y no conducen a la creación de nuevas clases. En el caso del vino, sin embargo, existe una notable excepción puesto que el color del vino es primordial para la descripción del vino.

Tomemos otro ejemplo, consideremos una ontología de la anatomía humana. Cuando representamos las costillas, ¿creamos clases para “1ra costilla izquierda”, “2da costilla izquierda” y así sucesivamente? O ¿creamos una clase Costilla con slots para el orden y la posición lateral (izquierda-derecha)?. Si la información de cada costilla que representamos en la ontología es significativamente diferente, entonces deberíamos de hecho crear una clase para cada una de las costillas. Es decir, si deseamos representar detalles de la información de adyacencia y localización (la cual es diferente para cada costilla) como también funciones específicas que cada costilla juega y órganos que protege, entonces nos interesa las clases. Si estamos modelando la anatomía a un nivel ligeramente leve de generalidad y todas las costillas son muy similares en nuestras aplicaciones potenciales (se trata de ver cuál costilla está rota en los rayos X sin implicaciones en las otras partes del cuerpo), entonces podríamos simplificar nuestra jerarquía y tener simplemente la clase Costilla, con dos slots: posición lateral y orden.

B.2.6 ¿Una instancia o una clase?

Decidir si un concepto particular es una clase en la ontología o una instancia individual depende de cuáles son las aplicaciones potenciales de la ontología. Decidir dónde las clases terminan y las instancias comienzan, empieza por la decisión de cuál es el nivel más bajo de granularidad en la representación. El nivel de granularidad es a su vez determinado por una aplicación potencial de la ontología. En otras palabras, '¿Cuáles son los ítems más específicos que representaremos en la base de conocimientos? Volviendo a las preguntas de competencia que hemos identificado en el Paso 1 de la Sección 3, los conceptos más específicos que constituirán respuestas a esas preguntas serán muy buenos candidatos para ser individuos en la base de conocimientos.

La instancias individuales son los conceptos más específicos representados en una base de conocimientos. Por ejemplo, si solo hablaremos del emparejado de vinos con alimentos, no estaremos interesados en específicas botellas físicas de vino. Por lo tanto, términos como Sterling Vineyards Merlot serán probablemente los términos más específicos que usemos. De este modo, Sterling Vineyards Merlot será una instancia en la base de conocimientos.
Por otro lado, si deseamos mantener un inventario de los vinos del restaurante además de la
base de conocimientos de buenas parejas vino-alimento, las botellas individuales de cada vino llegarán a ser instancias individuales en nuestra base de conocimientos.
De manera similar, si deseamos registrar las diferentes propiedades de cada cosecha específica de los Sterling Vineyards Merlot, entonces la cosecha específica de vino es una instancia en la base de conocimientos y Sterling Vineyards Merlot es una clase que contiene instancias para todas sus cosechas.
Otra regla puede "desplazar" algunas instancias individuales al conjunto de clases:

- Si los conceptos forman una jerarquía natural, entonces deberíamos representarlos como clases.

Consideremos las regiones de producción de vino. Inicialmente, podemos definir regiones principales producción de vino, tales como Francia, Estados Unidos, Alemania, y así sucesivamente, como clases, y regiones específicas de producción de vino dentro esas grandes regiones como instancias. Por ejemplo, la región de Borgoña es una instancia de la clase de Región Francesa. Sin embargo, también quisiéramos decir que la Región Côtes d'Or es una Región de Borgoña. En consecuencia, la Región de Borgoña debe ser una clase (para tener subclases o instancias). Sin embargo, parece arbitrario hacer que la Región de Borgoña sea una clase y que la Región Côtes d'Or sea una instancia de la Región de Borgoña: es muy difícil distinguir claramente qué regiones son clases y cuáles son instancias. Por lo tanto, definimos todas las regiones de producción de vino como clases. Protégé-2000 permite a los usuarios especificar algunas clases como Abstract (abstractas), indicando que la clase no puede tener ninguna instancia directa. En nuestro caso, todas las clases de las regiones de producción de vino son abstractas.

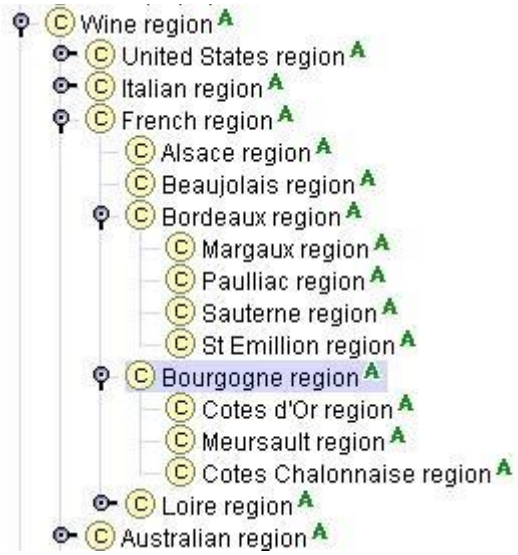

Figura 73: Jerarquía de las regiones de producción de vino

La misma jerarquía de clases sería incorrecta si omitimos la palabra "región" de los nombres de las clases. No podemos decir que la clase Alsacia (Alsace) es una subclase de la clase Francia: Alsacia no es un tipo de Francia. Sin embargo, la región de Alsacia es un tipo de región de Francia.
Solamente las clases pueden ser dispuestas en una jerarquía (los sistemas de representación de conocimiento no tienen la noción de sub-instancia). Por lo tanto, si existe una jerarquía natural entre los términos, como en las jerarquías terminológicas, debemos definir esos términos como clases aunque no tengan ninguna instancia propia.

B.2.7 Limitación del alcance

Como nota final sobre la definición de una jerarquía de clases, el siguiente conjunto de reglas es siempre útil para decidir si una ontología está completa:

La ontología no debería contener toda la información posible del dominio: no necesitas especializar (o generalizar) más de lo que necesitas para tu aplicación (como máximo un nivel extra de cada lado).

En nuestro ejemplo de vinos y alimentos, no necesitamos saber qué papel es usado para las etiquetas o cómo cocinar gambas.
De manera similar, la ontología no debe contener todas las posibles propiedades de las clases ni las distinciones entre ellas en la jerarquía.

En nuestra ontología, sin duda no incluiremos todas las propiedades que un vino o alimento puedan tener. Representamos las propiedades más sobresalientes de las clases de ítems en nuestra ontología. Aunque los libros de vinos nos dirán el tamaño de las uvas, no incluimos ese conocimiento. De manera similar, no hemos agregado todas las relaciones que podríamos imaginar entre todos los términos de nuestro sistema. Por ejemplo, no incluimos las relaciones tales como vino favorito o alimento preferido en la ontología para tener una representación más completa de todas las interconexiones entre los términos que hemos definido.

Las últimas reglas también se aplican para establecer relaciones entre conceptos que ya los hemos incluido en la ontología. Consideremos una ontología que describe experimentos biológicos. La ontología probablemente contendrá un concepto de Organismos Biológicos. También contendrá el concepto de un Experimentador que ejecuta un experimento (con su nombre, afiliación, etc.). Es cierto que un experimentador es una persona, y como persona, también es casualmente un organismo biológico. Sin embargo, probablemente no debamos incorporar esta distinción en la ontología: para los propósitos de esta representación un experimentador no es un organismo biológico y probablemente nunca ejecutemos experimentos en los experimentadores como tal. Si estuviésemos representando todo lo que podamos decir de las clases en la ontología, un Experimentador llegaría a ser una subclase de Organismo Biológico. Sin embargo, no necesitamos incluir este conocimiento para las aplicaciones previsibles. De hecho, la inclusión de este tipo de clasificación adicional para las clases existentes en realidad es perjudicial: una instancia de un Experimentador tendrá slots para peso, edad, especie, y otros datos pertenecientes a un organismo biológico, pero absolutamente irrelevantes en el contexto de la descripción de un experimento. Sin embargo, debemos registrar tal decisión de diseño en la documentación para el beneficio de los usuarios que mirarán esta ontología y que no estarán enterados de la aplicación que teníamos en mente.

B.2.8 Subclases disjuntas

Muchos sistemas nos permiten especificar explícitamente que varias clases sean disjuntas. La clases son disjuntas si no pueden tener ninguna instancia en común. Por ejemplo, las clases Vino de Sobremesa y Vino Blanco de nuestra ontología no son disjuntas: hay muchos vinos que son instancias de ambos. La instancia Rothermel Trochenbierenauslese Riesling de la clase Riesling Dulce es un ejemplo de ello. Al mismo tiempo, las clases Vino Rojo y Vino Blanco son disjuntas: ningún vino puede ser simultáneamente rojo y blanco. La especificación de clases disjuntas permite al sistema de validar la ontología de mejor manera. Si declaramos que las clases Vino Rojo y Vino Blanco son disjuntas y posteriormente creamos una clase que es una subclase de Riesling (una subclase de Vino Blanco) y Porto (una subclase de Vino Rojo), el sistema indicará que hay un error de modelamiento.

B.3 Definición de las propiedades

En esta sección discutimos muchos más detalles a tener en cuenta cuando definimos los slots de una ontología (Paso 5 y Paso 6). Principalmente, discutiremos sobre los roles inversos y valores por defecto de un slot.

B.3.1 Slots inversos

Un valor de un slot puede depender de un otro valor de otro slot Por ejemplo, si un vino fue producido por un establecimiento vinícola, entonces el establecimiento vinícola produce ese vino. Esas dos relaciones, productor y produce, son llamadas relaciones inversas. Almacenar la información "en ambos sentidos" es redundante. Cuando sabemos que un vino es producido por un establecimiento vinícola, una aplicación que usa una base de conocimientos puede siempre inferir el valor de la relación inversa: el establecimiento vinícola produce el vino. Sin embargo, desde la perspectiva de adquisición de conocimiento, es conveniente tener ambas piezas de la información disponibles explícitamente. Este enfoque permite a los usuarios introducir vinos en algunos casos y establecimientos vinícolas en otros. El sistema de adquisición de conocimiento podría entonces completar automáticamente el valor de la relación inversa asegurando la consistencia de la base de conocimientos.

Nuestro ejemplo tiene un par de slots inversos: el slot productor de la clase Vino y el slot produce de la clase Establecimiento Vinícola. Cuando un usuario crea una instancia de la clase Vino e introduce el valor para el slot productor, el sistema automáticamente añade la instancia recién creada al slot produce de la instancia correspondiente Establecimiento Vinícola. Por ejemplo, cuando decimos que el Sterling Merlot es producido por el establecimiento vinícola Viñedos Sterling, el sistema automáticamente agregaría Sterling Merlot a la lista de vinos que el establecimiento vinícola Viñedos Sterling (Sterling Vineyard) produce (Figura 74).

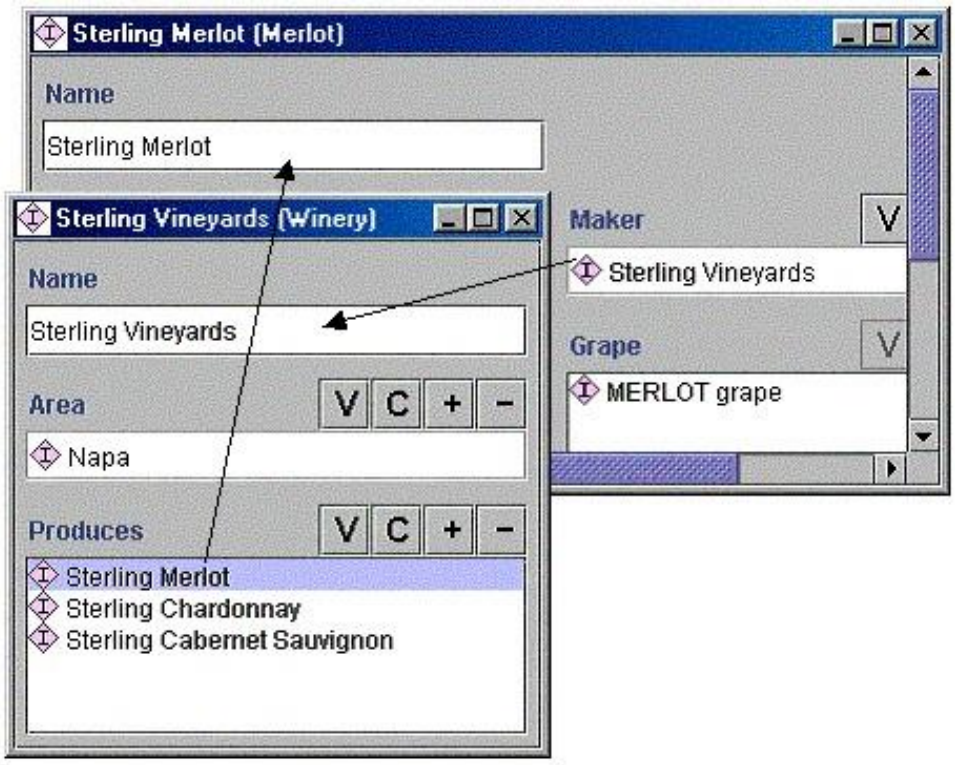

Figura 74: Instancias de slots inversos

B.3.2 Valores por defecto

Muchos sistemas basados en marcos permiten la especificación de valores por defecto para los slots. Si un valor particular de un slot es el mismo para la mayoría de las instancias de una clase, podemos entonces definir como el **valor por defecto** para el slot. Entonces, cuando cada nueva instancia de una clase que contenga este slot sea creada, el sistema lo llenará con el valor por defecto automáticamente. Luego podemos cambiar ese valor a otro valor que las facetas lo permitan. Es decir, los valores por defecto están ahí por comodidad: no imponen ninguna nueva restricción sobre el modelo ni cambian el modelo en alguna forma.

Por ejemplo, si la mayoría de los vinos de los cuales vamos a discutir son vinos de cuerpo completo, podemos tener "completo" como valor por defecto para el cuerpo del vino. Entonces, a menos que indiquemos otra cosa, todos los vinos que definamos serán de cuerpo completo.

Notar que esto es diferente de los **valores de los slots**. Los valores de los slots no pueden ser cambiados. Por ejemplo, podemos decir que el slot azúcar tiene el valor DULCE para la clase Vinos de Sobremesa. Entonces, todas las subclases e instancias de la clase Vino de Sobremesa tendrán el valor DULCE para el slot azúcar. Este valor no puede ser cambiado en ninguna de las subclases ni instancias de la clase.

B.4 ¿Qué está en un nombre?

La definición de convenios sobre los nombres de los conceptos en una ontología y su uso estricto, no solamente que la ontología sea fácil de entender sino también ayuda a evitar algunos errores comunes de modelamiento. Existen muchas alternativas para nombrar los conceptos. A menudo no hay ninguna razón particular para elegir una u otra alternativa. Sin embargo, necesitamos definir un convenio de nombrado de clases y slots, y adherirnos estrictamente a éste. La siguientes características de un sistema de representación de conocimiento afectan la elección de convenios de nombrado:

- ¿Tiene el sistema el mismo espacio de nombres para las clases, slots e instancias? Es decir, ¿permite el sistema tener una clase y un slot con el mismo nombre (tales como la clase establecimiento vinícola y el slot establecimiento vinícola)?

- ¿El sistema diferencia entre mayúsculas y minúsculas? Es decir, ¿el sistema trata los nombres de manera diferente si están escritos en mayúsculas o minúsculas (tales como Establecimiento Vinícola y establecimiento vinícola)?

- ¿Qué delimitadores permite el sistema en los nombres? Es decir, ¿los nombres pueden contener espacios, comas, asteriscos y así por el estilo?

Protégé-2000, por ejemplo, mantiene un solo espacio de nombres para todos sus marcos. Diferencia entre mayúsculas y minúsculas. En consecuencia, no podemos tener una clase establecimiento vinícola y un slot establecimiento vinícola. Sin embargo, podemos tener una clase Establecimiento Vinícola (mayúsculas) y un slot establecimiento vinícola. CLASSIC, por otro lado, no diferencia entre mayúsculas y minúsculas y mantiene diferentes espacios de nombres para las clases, slots e individuos. De esa manera, desde la perspectiva del sistema, no hay problema en nombrar la clase y el slot Establecimiento Vinícola.

B.4.1 Mayúsculas/minúsculas y delimitadores

Primero, podemos mejorar enormemente la legibilidad de una ontología si usamos de manera consistente los convenios sobre mayúsculas y minúsculas para los nombres de conceptos. Por ejemplo, es práctica común el escribir en mayúsculas los nombres de las clases y usar minúsculas en los nombres de los slots (asumiendo que el sistema diferencia entre mayúsculas y minúsculas). Cuando el nombre de un concepto contiene más de una palabra (tal como Plato de comida) necesitamos delimitar las palabras. Aquí hay algunas posibles opciones:

- Usar espacios: Plato de comida (muchos sistemas, incluyendo, Protégé-2000, permiten espacios en los nombres de conceptos).

- Escribir todas las palabras juntas y poner en mayúsculas cada nueva palabra: PlatoDeComida.

- Usar un subrayado o guion u otro delimitador en el nombre: Plato_ De_ Comida, Plato_ de_ comida, Plato_De_Comida, Plato_de_comida. (Si usas delimitadores, necesitarás también decidir si cada nueva palabra debe comenzar con una letra en mayúsculas).

Si el sistema de representación permite espacios en los nombres, su uso sería la solución más intuitiva para muchos desarrolladores de ontologías. Es, sin embargo, importante considerar otros sistemas con los cuales tu sistema podrá interactuar. Si esos sistemas no usan espacios o si tu medio de presentación no maneja bien los espacios, sería útil usar otro método.

B.4.2 Singular o Plural

Un nombre de una clase representa una colección de objetos. Por ejemplo, la clase Vino representa a todos los vinos. Por lo tanto, sería más natural para algunos diseñadores nombrar la clase Vinos en lugar de Vino. Ninguna alternativa es mejor o peor que otra (aunque la forma singular para las clases es usada más a menudo en la práctica). Sin embargo, cualquiera sea la elección, debe ser consistente a lo largo de toda la ontología. Algunos sistemas solicitan a sus usuarios declarar por adelantado si ellos usarán singular o plural para los nombres de los conceptos y no permiten desviarse de esa elección.

El uso de la misma forma todo el tiempo también previene al diseñador de cometer errores de modelamiento como crear una clase Vinos y luego crear una clase Vino como su subclase.

B.4.3 Convenios: prefijos y sufijos

Algunas metodologías de bases de conocimiento sugieren usar convenios sobre el uso de prefijos y sufijos en los nombres para distinguir entre clases y slots. Dos prácticas comunes son: añadir tiene- como prefijo o el sufijo -de a los nombres de los slots. De esta forma, nuestro slots serán tiene-productor y tiene-establecimiento vinícola si elegimos el convenio que sugiere el uso de tiene-. Si elegimos usar el convenio que sugiere que se emplee -de, los slots serán productor-de y establecimiento vinícola-de. Este enfoque permite que los usuarios vean el término para poder determinar inmediatamente si el término es una clase o slot. Sin embargo, los nombres de los términos llegan a ser ligeramente largos.

B.4.4 Otras consideraciones de nombrado

Aquí están unas cuantas cosas más a considerar cuando se definan los convenios de nombrado:

- No agregar cadenas tales como “clase”, “propiedad”, “slot”, y así sucesivamente a los nombres de conceptos. Siempre está claro a partir del contexto si el concepto es una clase o un slot. Además, si usas diferentes convenios de nombrado para las clases y slots (por ejemplo, mayúsculas y minúsculas respectivamente), el nombre como tal sería indicativo de qué es el concepto.

- Normalmente es buena idea evitar abreviaciones en los nombres de conceptos (es decir, usar Cabernet Sauvignon en lugar de Cab).

- Los nombres de las subclases directas de una clase deberían todas incluir o no incluir el nombre de la superclase. Por ejemplo, si estamos creando dos subclases de la clase Vino para representar vinos rojos y vinos blancos, los dos nombres de las subclases deberían ser Vino Rojo y Vino Blanco, y no así Vino Rojo y Blanco.

B.5 Conclusiones

En esta guía, hemos descrito una metodología de desarrollo de ontologías para sistemas declarativos basados en marcos. Listamos los pasos del proceso de desarrollo de ontologías e hicimos referencia a los aspectos complejos de la definición de jerarquías de clases y propiedades de clases e instancias. Sin embargo, después de seguir todas las reglas y sugerencias, una de las cosas más importantes que recordar es la siguiente: **no hay una sola ontología correcta para un dominio dado**. El diseño de ontologías es un proceso creativo y dos ontologías diseñadas por personas diferentes no serán iguales. Las aplicaciones potenciales de la ontología y el entendimiento y aspecto del dominio desde el punto de vista del diseñador afectarán sin duda las opciones de diseño de la ontología. “No se sabe si algo es bueno hasta que se pone a prueba”: podemos evaluar la calidad de nuestra ontología solamente usándola en aplicaciones para las cuales la diseñamos.

Apéndice C

Archivo OWL de la ontología del proyecto

El siguiente código es el correspondiente a la ontología descrita en el capítulo 3 de esta memoria. Se diferencian claramente tres apartados. En el primero se pueden observar los prefijos de todas las URIS que utiliza la ontología. El segundo comienza en *Data properties*, e incluye la definición ontológica de todas y cada una de las propiedades explicadas anteriormente. Se observa como cada pequeño bloque equivale a una propiedad, en la que primero se indica su correspondiente URI con el uso de los prefijos, luego se define el tipo, se declara como propiedad funcional (cada paciente solo puede tener una única propiedad del mismo tipo, es decir no puede tener dos valores distintos de frecuencia cardiaca) y se declaran el rango y el dominio. El tercer bloque, corresponde a las clases o recursos, obviamente el más largo y complicado es la definición del recurso enfermo ya que contiene todas las restricciones que se describieron en el capítulo 3. Una vez descrito el recurso Enfermo, se implementan también las descripciones del recurso Paciente y del recurso Sano que son mucho más sencillas. Con pequeñas nociones de HTML o XML, el código es bastante intuitivo:

```
?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY www2 "http://www.monitorizaciontensiones.com/" >
    <!ENTITY www "http://www.monitorizaciontensiones.com/#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>


<rdf:RDF xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://www.w3.org/2002/07/owl"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:www="&www2;#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:www2="http://www.monitorizaciontensiones.com/">
    <Ontology rdf:about="http://www.monitorizaciontensiones.com/"/>
    <!--

////////////////////////////////////////////////////////////////////////
//////////
    //
    // Data properties
    //
////////////////////////////////////////////////////////////////////////
//////////
     -->
 <!-- http://www.monitorizaciontensiones.com/#frecuenciaCardiaca -->
```

```
    <DatatypeProperty rdf:about="&www2;#frecuenciaCardiaca">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="&www2;#Paciente"/>
        <rdfs:range rdf:resource="&xsd;integer"/>
    </DatatypeProperty>

       <!--http://www.monitorizaciontensiones.com/#frecuenciaRespiratoria -->

    <DatatypeProperty rdf:about="&www2;#frecuenciaRespiratoria">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="&www2;#Paciente"/>
        <rdfs:range rdf:resource="&xsd;integer"/>
    </DatatypeProperty>

  <!-- http://www.monitorizaciontensiones.com/#nivelOxigeno -->

    <DatatypeProperty rdf:about="&www2;#nivelOxigeno">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="&www2;#Paciente"/>
        <rdfs:range rdf:resource="&xsd;integer"/>
    </DatatypeProperty>


    <!-- http://www.monitorizaciontensiones.com/#temperatura -->

    <DatatypeProperty rdf:about="&www2;#temperatura">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="&www2;#Paciente"/>
        <rdfs:range rdf:resource="&xsd;decimal"/>
    </DatatypeProperty>

    <!-- http://www.monitorizaciontensiones.com/#valorDiastolico -->

    <DatatypeProperty rdf:about="&www2;#valorDiastolico">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="&www2;#Paciente"/>
        <rdfs:range rdf:resource="&xsd;integer"/>
    </DatatypeProperty>

    <!-- http://www.monitorizaciontensiones.com/#valorSistolico -->

    <DatatypeProperty rdf:about="&www2;#valorSistolico">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="&www2;#Paciente"/>
        <rdfs:range rdf:resource="&xsd;integer"/>
    </DatatypeProperty>

    <!--

///////////////////////////////////////////////////////////////////////////////
//////////
    //
    // Classes
    //

///////////////////////////////////////////////////////////////////////////////
//////////
     -->

    <!-- http://www.monitorizaciontensiones.com/#Enfermo -->
```

```
    <Class rdf:about="&www2;#Enfermo">
        <equivalentClass>
            <Class>
                <intersectionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="&www2;#Paciente"/>
                    <Class>
                        <unionOf rdf:parseType="Collection">
                            <Restriction>
                                <onProperty
rdf:resource="&www2;#frecuenciaCardiaca"/>
                                <someValuesFrom>
                                    <rdfs:Datatype>
                                        <onDatatype
rdf:resource="&xsd;integer"/>
                                        <withRestrictions
rdf:parseType="Collection">
                                            <rdf:Description>
                                                <xsd:minExclusive
rdf:datatype="&xsd;integer">100</xsd:minExclusive>
                                            </rdf:Description>
                                        </withRestrictions>
                                    </rdfs:Datatype>
                                </someValuesFrom>
                            </Restriction>
                            <Restriction>
                                <onProperty
rdf:resource="&www2;#frecuenciaRespiratoria"/>
                                <someValuesFrom>
                                    <rdfs:Datatype>
                                        <onDatatype
rdf:resource="&xsd;integer"/>
                                        <withRestrictions
rdf:parseType="Collection">
                                            <rdf:Description>
                                                <xsd:minExclusive
rdf:datatype="&xsd;integer">20</xsd:minExclusive>
                                            </rdf:Description>
                                        </withRestrictions>
                                    </rdfs:Datatype>
                                </someValuesFrom>
                            </Restriction>
                            <Restriction>
                                <onProperty
rdf:resource="&www2;#nivelOxigeno"/>
                                <someValuesFrom>
                                    <rdfs:Datatype>
                                        <onDatatype
rdf:resource="&xsd;integer"/>
                                        <withRestrictions
rdf:parseType="Collection">
                                            <rdf:Description>
                                                <xsd:maxExclusive
rdf:datatype="&xsd;integer">95</xsd:maxExclusive>
                                            </rdf:Description>
                                        </withRestrictions>
                                    </rdfs:Datatype>
                                </someValuesFrom>
                            </Restriction>
                            <Restriction>
```

```
                                <onProperty
rdf:resource="&www2;#temperatura"/>
                                <someValuesFrom>
                                    <rdfs:Datatype>
                                        <onDatatype
rdf:resource="&xsd;decimal"/>
                                        <withRestrictions
rdf:parseType="Collection">
                                            <rdf:Description>
                                                <xsd:minInclusive
rdf:datatype="&xsd;integer">37</xsd:minInclusive>
                                            </rdf:Description>
                                        </withRestrictions>
                                    </rdfs:Datatype>
                                </someValuesFrom>
                            </Restriction>
                            <Restriction>
                                <onProperty
rdf:resource="&www2;#valorDiastolico"/>
                                <someValuesFrom>
                                    <rdfs:Datatype>
                                        <onDatatype
rdf:resource="&xsd;integer"/>
                                        <withRestrictions
rdf:parseType="Collection">
                                            <rdf:Description>
                                                <xsd:minInclusive
rdf:datatype="&xsd;integer">90</xsd:minInclusive>
                                            </rdf:Description>
                                        </withRestrictions>
                                    </rdfs:Datatype>
                                </someValuesFrom>
                            </Restriction>
                            <Restriction>
                                <onProperty
rdf:resource="&www2;#valorSistolico"/>
                                <someValuesFrom>
                                    <rdfs:Datatype>
                                        <onDatatype
rdf:resource="&xsd;integer"/>
                                        <withRestrictions
rdf:parseType="Collection">
                                            <rdf:Description>
                                                <xsd:minInclusive
rdf:datatype="&xsd;integer">140</xsd:minInclusive>
                                            </rdf:Description>
                                        </withRestrictions>
                                    </rdfs:Datatype>
                                </someValuesFrom>
                            </Restriction>
                        </unionOf>
                    </Class>
                </intersectionOf>
            </Class>
        </equivalentClass>
        <rdfs:subClassOf rdf:resource="&www2;#Paciente"/>
        <disjointWith rdf:resource="&www2;#Sano"/>
    </Class>
```

```
    <!-- http://www.monitorizaciontensiones.com/#Paciente -->

    <Class rdf:about="&www2;#Paciente">
        <equivalentClass>
            <Restriction>
                <onProperty rdf:resource="&www2;#valorSistolico"/>
                <someValuesFrom rdf:resource="&xsd;integer"/>
            </Restriction>
        </equivalentClass>
    </Class>



    <!-- http://www.monitorizaciontensiones.com/#Sano -->

    <Class rdf:about="&www2;#Sano">
        <equivalentClass>
            <Class>
                <intersectionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="&www2;#Paciente"/>
                    <Class>
                        <complementOf rdf:resource="&www2;#Enfermo"/>
                    </Class>
                </intersectionOf>
            </Class>
        </equivalentClass>
        <rdfs:subClassOf rdf:resource="&www2;#Paciente"/>
    </Class>



</rdf:RDF>



<!-- Generated by the OWL API (version 3.4.2) http://owlapi.sourceforge.net -
->
```

Apéndice D

Fragmentos destacables del código desarrollado

En este apartado de la memoria, se mostrarán las partes más significativas del código Java desarrollado y se explicará el significado del mismo. Tal y cómo se ha comentado en el capítulo 4, se desarrollarán las clases PlantaHospital y sus dos clases hijas PlantaHospitalWOL y PlantaHospitalJena.

D.1 Planta Hospital

PlantaHospital.java, es una clase de la que heredarán las dos implementaciones que usen Jena y OWLAPI, es por ello que lo primero que se ha hecho es definir los atributos que serán comunes a todas ellas tales como las IRIs de todas las propiedades y recursos, así como el número de habitaciones que tendrá la planta del hospital. Lo más importante de la clase PlantaHospital es su constructor el cual se encarga de abrir el buffer de lectura para cargar la ontología. Este buffer se utiliza en las dos implementaciones de las clases hijas es por ello que lo primero que harán dichos constructores será llamar al constructor de la clase PlantaHospital mediante la sentencia super().

```
public abstract class PlantaHospital {


//Definición de los atributos de la clase PlantaHospital

public static final String NS="http://www.monitorizaciontensiones.com/";
public static final String VS="#valorSistolico";
public static final String VD="#valorDiastolico";
public static final String FRESP="#frecuenciaRespiratoria";
public static final String FCARD="#frecuenciaCardiaca";
public static final String TEMP="#temperatura";
public static final String OXI="#nivelOxigeno";
public static final String VAL="valor";
public static final int numReglas=6;
public static final int NUMHAB = 96;
public InputStream in;
public static int []hab;
public int numPacientes;

//Constructor de la clase PlantaHospital

public PlantaHospital(String _in)throws MalformedURLException, OWLException,
    InstantiationException, IllegalAccessException, ClassNotFoundException{
            in=FileManager.get().open(_in);
            if(in==null) throw new IllegalArgumentException("Error al leer");
            numPacientes=0;
      }
```

D.2 Planta Hospital Jena

D.2.1 Constructor

```
public class PlantaHospitalJena extends PlantaHospital {
	//Definición de los atributos
	OntModel m;
	InfModel inf;
	//Constructor de la clase PLantaHospitalJena
	public PlantaHospitalJena(String s) throws MalformedURLException,
InstantiationException, IllegalAccessException, ClassNotFoundException,
OWLException{
		super(s);


	m =
ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM_MICRO_RULE_INF);
	m.read(in,"");
	//Sentencias para abrir con razonador OWL y desarrollar modelo inferido
	Reasoner res=ReasonerRegistry.getOWLReasoner();
	inf =ModelFactory.createInfModel(res,m);

	try{
		in.close();
		}catch(Exception e){System.out.println(e);}
	}
}
```

Cómo se ha explicado en el capítulo 3, Jena no es capaz todavía de inferir conocimiento, a pesar de ello, el código está implementado para que pueda funcionar el modelo inferido una vez Jena sea capaz de utilizarlo. Es por ello que Jena consta de dos atributos además de los que hereda, *OntModel m* e *InfModel inf*. *OntModel m* será el modelo ontológico mientras que *inf* será el modelo inferido. El constructor como se ha comentado en el apartado anterior, lo primero que hace es llamar al constructor de la clase padre, de esta manera abre el buffer de lectura de la ontología con la dirección que se ha indicado en s que será la ruta donde se encuentra almacenada la ontología explicada en el apéndice C. Una vez abierto el buffer, crea el modelo ontológico y carga la ontología en memoria a través del método read(). Por otro lado intenta desarrollar la inferencia utilizando un razonador y un modelo inferido, pero como ya se ha explicado, estos métodos todavía no son capaces de aportar valor a la clase. Una vez se ha cargado la ontología, cierra el buffer y termina el código del constructor.

D.2.2 crearPaciente()

```
public  Paciente crearPaciente(int tensionSistolica, int tensionDiastolica,
int fc, int fr, double t, int ox){
	  String nombre="Paciente"+numPacientes;
	    OntClass pacientes= m.getOntClass(NS+"#Paciente");
	    Property p= m.getProperty(NS+VS);
	    Property p2= m.getProperty(NS+VD);
	    Property p3= m.getProperty(NS+FCARD);
	    Property p4= m.getProperty(NS+FRESP);
	    Property p5= m.getProperty(NS+TEMP);
	    Property p6= m.getProperty(NS+OXI);
```

```
            Individual i=m.createIndividual(NS+"#"+nombre,pacientes);
            i.addProperty(p, tensionSistolica+"", XSDDatatype.XSDinteger);
            i.addProperty(p2, tensionDiastolica+"", XSDDatatype.XSDinteger);
            i.addProperty(p3, fc+"", XSDDatatype.XSDinteger);
            i.addProperty(p4, fr+"", XSDDatatype.XSDinteger);
            i.addProperty(p5, t+"", XSDDatatype.XSDdecimal);
            i.addProperty(p6, ox+"", XSDDatatype.XSDinteger);
            return super.crearPaciente(tensionSistolica,tensionDiastolica,
fc, fr,t,ox);
   }
```

El método crearPaciente(), sirve, como su propio nombre indica, para introducir un paciente en la ontología. Para ello se le da un nombre por defecto al paciente (“PacienteXX”) y se le piden al modelo las clases y las propiedades que se van a utilizar mediante los métodos getProperty y getOntClass. Una vez tenemos la clase paciente y las propiedades correspondientes a los indicadores de salud, podemos crear el individuo en el modelo m mediante la sentencia m.createIndividual() y se le van añadiendo las propiedades correspondientes pasadas por parámetro usando el método addProperty(). Una vez se ha añadido el paciente y todas sus propiedades, se llama al método de la clase padre que simplemente actualiza el número de pacientes (el atributo numPacientes) y devuelve un objeto de la clase paciente con las características que se le han pasado por parámetro.

D.2.3 printEnfermos()

```
public long printEnfermos(){

         long t0=System.currentTimeMillis();
            String query="PREFIX enf:
<http://www.monitorizaciontensiones.com/#>" +
SELECT  ?Paciente ?valorDiastolico ?valorSistolico ?frecuenciaCardiaca
?frecuenciaRespiratoria ?temperatura ?nivelOxigeno" +
"WHERE { ?Paciente enf:valorSistolico ?valorSistolico." +
            "?Paciente enf:valorDiastolico ?valorDiastolico." +
            "?Paciente enf:frecuenciaCardiaca ?frecuenciaCardiaca." +
            "?Paciente enf:frecuenciaRespiratoria ?frecuenciaRespiratoria." +
            "?Paciente enf:temperatura ?temperatura." +
            "?Paciente enf:nivelOxigeno ?nivelOxigeno." +
            "FILTER(?valorSistolico>=140 || ?valorDiastolico >=90 ||
?frecuenciaCardiaca >100 || ?frecuenciaRespiratoria >20 || ?temperatura >=37
||?nivelOxigeno <95)}";
            Query q=QueryFactory.create(query);
            // Ejecuto la query y obtengo los resultados
            QueryExecution qe = QueryExecutionFactory.create(query, m);
            ResultSet results = qe.execSelect();
            System.out.println("Los enfermos con hipertensión son:");
            // Muestro los resultados
            ResultSetFormatter.out(System.out, results,q);
            qe.close();

            long t=System.currentTimeMillis();

            return (t-t0);

}
```

El método printEnfermos() utiliza el lenguaje SPARQL para comunicarse con la ontología. Es por eso que lo primero que se hace en este método es construir una consulta en SPARQL que será el *String query*. En ella se piden todos los datos de los paciente (valor diastólico, valor sistólico frecuencia cardiaca…) pero se le añade un filtro. Este filtro lo que hace es pedirle que solo muestre los pacientes que estén dentro de los valores de los enfermos, es decir con valor sistólico mayor que 140 o con frecuencia cardiaca mayor que 100 etc… Una vez que la query está bien construida (sintaxis explicada en el capítulo 3), se crea la query mediante el método estático créate() se ejecuta mediante el método execSelect() y se muestran los resultados con ResultsSetFormatter.out().

Hay que destacar también que en este método se mide el tiempo que se tarda en clasificar a los enfermos, se pide el tiempo del sistema a la entrada y salida del método mediante la sentencia System.CurrentTimeMillis() y se devuelve la resta de ambas de manera que obtenemos el tiempo de clasificación para cada ejecución del método.

D.3 Planta Hospital OWL

D.3.1 Constructor

```
public class PlantaHospitalOWL extends PlantaHospital {
      //Definición de los atributos
      private Reasoner hermit;
      private OWLOntology ont;
      private OWLOntologyManager man;
      private OWLDataFactory df;

      //Constructor de la clase PLantaHospitalJena
      public PlantaHospitalOWL(String _in)throws MalformedURLException,
OWLException,
    InstantiationException, IllegalAccessException, ClassNotFoundException {
            super(_in);
            man = OWLManager.createOWLOntologyManager();
            ont=man.loadOntologyFromOntologyDocument(in);

            try{
                  in.close();
                  }catch(Exception e){System.out.println(e);}
            df = man.getOWLDataFactory();
            hermit=new Reasoner(ont);

      }
```

Salta a la vista que OWLAPI es más complicado que Jena. Para empezar el número de atributos que se utilizan para manejar la ontología son 4 en vez de los 2 que se utilizaban en Jena. En este caso tendremos el razonador *hermit*, la ontología en si misma *OWLOntology ont*, el objeto que gestiona la ontología *OWLOntologyManager man* y el objeto que guarda todos los datos *OWLDataFactory df*. El constructor de nuevo lo primero que hace es abrir el buffer de lectura, crea una ontología y carga la ontología del buffer mediante el método loadOntologyFromOntologyDocument(). Acto seguido cierra el buffer de lectura, carga los datos de la ontología en *df* e inicializa el razonador.

D.3.2 crearPaciente()

```
public Paciente crearPaciente(int valorSistolico, int valorDiastolico, int
fc, int fr, double t, int ox){
            String nombre="Paciente"+numPacientes;
            //Creo los tipos de datos que voy a utilizar
              OWLDatatype integerDatatype =
df.getOWLDatatype(OWL2Datatype.XSD_INTEGER.getIRI());
               OWLDatatype decimalDatatype =
df.getOWLDatatype(OWL2Datatype.XSD_DECIMAL.getIRI());

              //creo los valores
            OWLLiteral sisto = df.getOWLLiteral(valorSistolico+"",
integerDatatype);
            OWLLiteral diasto = df.getOWLLiteral(valorDiastolico+"",
integerDatatype);
            OWLLiteral fcard = df.getOWLLiteral(fc+"", integerDatatype);
            OWLLiteral fresp = df.getOWLLiteral(fr+"", integerDatatype);
            OWLLiteral temp = df.getOWLLiteral(t+"", decimalDatatype);
            OWLLiteral oxi = df.getOWLLiteral(ox+"", integerDatatype);

            //creo el individuo
            OWLIndividual
individuo=df.getOWLNamedIndividual(IRI.create(NS+"#"+nombre));
            //importo los tipos de propiedad
            OWLDataProperty valorSistolico1=
df.getOWLDataProperty(IRI.create(NS+VS));
            OWLDataProperty valorDiastolico1=
df.getOWLDataProperty(IRI.create(NS+VD));
            OWLDataProperty fcard1=
df.getOWLDataProperty(IRI.create(NS+FCARD));
            OWLDataProperty fresp1=
df.getOWLDataProperty(IRI.create(NS+FRESP));
            OWLDataProperty temp1=
df.getOWLDataProperty(IRI.create(NS+TEMP));
            OWLDataProperty oxi1= df.getOWLDataProperty(IRI.create(NS+OXI));

            //añado las propiedades al individuo
            OWLAxiom
assrtion1=df.getOWLDataPropertyAssertionAxiom(valorSistolico1,
individuo,sisto);
            OWLAxiom
assrtion2=df.getOWLDataPropertyAssertionAxiom(valorDiastolico1,
individuo,diasto);
            OWLAxiom assrtion3=df.getOWLDataPropertyAssertionAxiom(fcard1,
individuo,fcard);
            OWLAxiom assrtion4=df.getOWLDataPropertyAssertionAxiom(fresp1,
individuo,fresp);
            OWLAxiom assrtion5=df.getOWLDataPropertyAssertionAxiom(temp1,
individuo,temp);
            OWLAxiom assrtion6=df.getOWLDataPropertyAssertionAxiom(oxi1,
individuo,oxi);

            //añado a la ontología
            AddAxiom addAxiomChange = new AddAxiom(ont,assrtion1);
            AddAxiom addAxiomChange2 = new AddAxiom(ont,assrtion2);
            AddAxiom addAxiomChange3 = new AddAxiom(ont,assrtion3);
            AddAxiom addAxiomChange4 = new AddAxiom(ont,assrtion4);
            AddAxiom addAxiomChange5 = new AddAxiom(ont,assrtion5);
            AddAxiom addAxiomChange6 = new AddAxiom(ont,assrtion6);
```

```
            //aplico los cambios
            man.applyChange(addAxiomChange);
            man.applyChange(addAxiomChange2);
            man.applyChange(addAxiomChange3);
            man.applyChange(addAxiomChange4);
            man.applyChange(addAxiomChange5);
            man.applyChange(addAxiomChange6);

            OWLClass paciente = df.getOWLClass(IRI.create(NS +
"#Paciente"));

            OWLClassAssertionAxiom ax =
df.getOWLClassAssertionAxiom(paciente,
                    individuo);

            man.addAxiom(ont, ax);
            return super.crearPaciente(valorSistolico,valorDiastolico, fc,
fr,t,ox);

      }
```

El método crearPaciente() guarda algunas semejanzas con el mismo método de la clase PlantaHospitalJena, pero añade ciertas complicaciones que Jena no tiene. De nuevo lo primero de todo es asignarle un nombre al paciente, después se obtienen los tipos de datos que se utilizarán en las propiedades, en este caso enteros y decimales. El siguiente paso es crear los *Literals* de manera que la ontología entienda los valores de los datos que se le van a introducir, (este paso por ejemplo es transparente en Jena). Se crea el individuo, se obtienen el tipo de las propiedades que se van a utilizar (paso análogo en Jena), se crean los axiomas de cambio para cada una de las propiedades y se aplican dichos cambios. De nuevo al final se vuelve a llamar al método de la clase padre para que actualice el número de pacientes y te devuelva el objeto de la clase paciente.

D.3.3 printEnfermos()

```
public Object[] printEnfermos(){
      OWLClass enfermo=df.getOWLClass(IRI.create(NS+"#Enfermo"));
      Vector v=new Vector();
      long t0=System.currentTimeMillis();
      String s;
      String enfermos;
 for(Node<OWLNamedIndividual> i : hermit.getInstances(enfermo, false)){
          s=(i.getEntities()+"");
          enfermos=s.substring(42, s.length()-2);
          v.add(enfermos);
       }
       Object []enferms=  v.toArray();
       long t=System.currentTimeMillis();
       hermit.flush();
       return enferms;
}
```

El método printEnfermos() en este caso es mucho más sencillo ya que es el razonador hermit el que hace todo el trabajo aprovechándose de las restricciones definidas en la ontología. Para ello se obtiene la clase que queremos que el razonador nos muestre, en este caso la clase Enfermo. Una vez tenemos la clase, se

le dice al razonador que con un bucle for recorra todos los individuos de la ontología y nos devuelva los que estén enfermos a través del método getInstances(), cada uno de esos individuos es añadido a un array que es el que devolverá el método printEnfermos() para que la interfaz gráfica sea capaz de localizar cada uno de ellos en el hospital y representarlos adecuadamente. En este caso también se ha medido el tiempo de clasificación utilizando el mismo método descrito en el apartado D.2.3.

D.3.4 curarPacientes()

```
public void curarPacientes(float prob){
      Object []enfermos=printEnfermos();
      Paciente p;
      float pr;
      for(int i=0;i<enfermos.length;i++){
            pr=(float)Math.random();
            if(pr<=prob){
                  p=getPaciente(enfermos[i]+"");
if(p.tensionDias>=90)p.tensionDias=p.tensionDias-(int)(20*Math.random());
if(p.tensionSis>=140)p.tensionSis=p.tensionSis-(int)(20*Math.random());
if(p.frecResp>=20)p.frecResp=p.frecResp-(int)(5*Math.random());
if(p.frecCard>=100)p.frecCard=p.frecCard-(int)(10*Math.random());
if(p.temp>=35)p.temp=p.temp-(int)(2*Math.random());
if(p.oxigeno<=95)p.oxigeno=p.oxigeno+(int)(5*Math.random());
                  setPaciente(p);
                  }

            }
}
```

El método curarPacientes es un método que utilizará el hilo implementado en la interfaz gráfica que sirve principalmente para recorrer el hospital y curar a un tanto por ciento de los enfermos. Para ello lo primero que hace es conseguir el array de enfermos mediante el método printEnfermos(), recorre dicho array y para cada uno de los enfermos genera un número aleatorio entre 0 y 1. Si dicho número es menor que la probabilidad pasada por parámetro, mejorarán sus constantes vitales de forma significativa. Se actualizarán sus valores y mediante el método setPaciente() se guardarán en la ontología.

D.3.5 enfermarPacientes()

```
public void enfermarPacientes(float prob){
      Paciente p;
      float pr;
      for(int i=0;i<numPacientes;i++){
            pr=(float)Math.random();
            if(pr<=prob){
                  p=getPaciente("Paciente"+i);
if(p.tensionDias<90)p.tensionDias=p.tensionDias+(int)(20*Math.random());
if(p.tensionSis<140)p.tensionSis=p.tensionSis+(int)(20*Math.random());
if(p.frecResp<20)p.frecResp=p.frecResp+(int)(5*Math.random());
if(p.frecCard<100)p.frecCard=p.frecCard+(int)(10*Math.random());
if(p.temp<35)p.temp=p.temp+(int)(2*Math.random());
if(p.oxigeno>95)p.oxigeno=p.oxigeno-(int)(5*Math.random());
                  setPaciente(p);
                  }

}
```

Este método es el inverso al método curarPacientes() en vez de curar a un porcentaje de los enfermos lo que hace es empeorar las constantes vitales del porcentaje indicado de los pacientes. Para ello recorre toda la planta del hospital, genera de nuevo un número aleatorio entre 0 y 1, y si dicho número es menor que la probabilidad pasada por parámetro empeorará significativamente las constantes vitales de dicho paciente usando el método setPaciente().

Glosario

1. *World Wide Web Consortium* (W3C) es un consorcio que se dedica a producir estándares para la Web semántica.

2. *XML* es un lenguaje de marcas desarrollado por el W3C y permite definir la gramática de lenguajes específicos.

3. *Framework* es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, un framework puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto. Para entenderlo fácilmente se puede decir que es el esqueleto sobre el cual varios objetos son integrados para una solución dada.

4. *HTML* lenguaje de Marcas de Hipertexto, es el lenguaje de marcado predominante para la construcción de páginas web.

5. *URIs* son cadenas que identifican algún recurso -como imágenes, documentos, archivos,…, para hacerlo disponible bajo una gran cantidad de esquemas como HTTP o FTP.

6. *URL* significa Uniform Resource Locator, es decir, localizador uniforme de recurso.

7. *SQL* Structured Query Language, es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones sobre las mismas.

8. *API* es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

9. *RDQL* lenguaje de consulta RDF.

10. *SPARQL* es un acrónimo recursivo del inglés SPARQL Protocol and RDF Query Language. Se trata de una recomendación para crear un lenguaje de consulta dentro de la Web semántica.

11. *Ontología*: es el término ontología en informática hace referencia a la formulación de un exhaustivo y riguroso esquema conceptual dentro de un dominio dado, con la finalidad de facilitar la comunicación y la compartición de la información entre diferentes sistema

Bibliografía

[1] J.Lupón. “Evaluación de la efectividad en Cataluña para la gestión remota de pacientes con insuficiencia cardiaca”. Disponible en octubre de 2013 a través de la URL: http://www.academia.cat/files/425-933-DOCUMENT/Lupon-41010-7.pdf

[2] Google Glass. Disponible en octubre de 2013 a través de la URL: http://www.google.com/glass/

[3] Rafael García. “El Dr. Guillén restransmite una operación vía Google Glass” 2013. Disponible en octubre de 2013 a través de la URL: http://computerhoy.com/noticias/apps/dr-guillen-restransmite-operacion-google-glass-6411

[4] A.Fernández. “mobiCeliac”. 2013. Disponible en octubre de 2013 a través de la URL: http://www.mobiceliac.com/

[5] Higehos Home. Disponile en octubre de 2013 a través de la URL: http://www.clinicadelaasuncion.com/tecnologia/

[6] Martínez Romero. “The iOSC3 System”. 2013. Disponible en octubre de 2013 a través de la URL: http://www.marcosmartinezromero.com/researchworks/2013/2013-CMMM.pdf

[7] P. Castells. “La web semántica”. 2003. Disponible en octubre de 2013 a través de la URL: http://blogs.enap.unam.mx/asignatura/francisco_alarcon/wp-content/uploads/2012/01/web_semantica.pdf

[8] A.Aranda. “Razonadores en la web semántica”. 2007. Disponible en octubre de 2013 a través de la URL: www.tdg-seville.info/Download.ashx?id=66

[9] B.McBride. “An introduction to RDF and the Jena RDF API”. 2010. Disponible en octubre de 2013 a través de la URL: http://jena.apache.org/tutorials/rdf_api.html

[10] B.McBride. “RDF Vocabulary Description Language 1.0: RDF Schema”. 2004 Disponible en octubre de 2013 a través de la URL: http://www.w3.org/TR/rdf-schema/

[11] S.Bechhofer. “OWL Web Ontology Language Reference”. 2004. Disponible en octubre de 2013 a través de la URL: http://www.w3.org/TR/owl-ref/

[12] The Apache Software Foundation. ”SPARQL Tutorial”. 2011. Disponible en octubre de 2013 a través de la URL: Disponible en http://jena.apache.org/tutorials/sparql_data.html

[13] The Knowledge Representation and Reasoning Group. “Hermit OWL Reasoner”. 2008. Disponible en octubre de 2013 a través de la URL: http://hermit-reasoner.com/

[14] Protegé. Disponible en octubre de 2013 a través de la URL: http://protege.stanford.edu/

[15] The Apache Software Foundation. "Jena Ontology API". 2011. Disponible en octubre de 2013 a través de la URL: http://jena.apache.org/documentation/ontology/index.html

[16] The OWL API. Disponible en octubre de 2013 a través de la URL: owlapi.sourceforge.net

[17] Java. Disponible en octubre de 2013 a través de la URL: http://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)

[18] Natividad Martínez. Interfacez gráficas de usuario en Java. 2004. Disponible en octubre de 2013 a través de la URL: http://www.it.uc3m.es/tlp/

[19] Horridge. "The OWL API: A Java API for Working with OWL 2 Ontologies". 2010. Disponible en octubre de 2013 a través de la URL: http://webont.org/owled/2009/papers/owled2009_submission_29.pdf

[20] Horridge. "A Practical Guide To Building OWL Ontologies Using Protegé 4 and CO-Ode Tools". 2010. Disponible en http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial-p4.0.pdf

[21] J.Henss."A database backend for OWL". 2009. Disponible en octubre de 2013 a través de la URL: http://webont.org/owled/2009/papers/owled2009_submission_3.pdf

[22] The Knowledge Representation and Reasoning Group. "OWL API Reasoners". 2013. Disponible en octubre de 2013 a través de la URL: http://owlapi.sourceforge.net/reasoners.html

[23] The Apache Software Foundation. "Jena Ontology API". 2011. Disponible en octubre de 2013 a través de la URL: http://jena.apache.org/documentation/ontology/index.html

[24] Connolly." DAML+OIL". 2001. Disponible en octubre de 2013 a través de la URL: http://www.w3.org/TR/daml+oil-reference

[25] The Apache Software Foundation. "Getting Started with Apache Jena". 2013.Disponible en octubre de 2013 a través de la URL: http://jena.apache.org/getting_started/index.html

[26] J. González Ortega. "Lenguajes de recuperación". 2010. Disponible en octubre de 2013 a través de la URL: http://serqlsparql.50webs.com/SeRQL_SPARQL.pdf

[27] E. Prud'hommeaux. "SPARQL Query Language for RDF". 2008. Disponible en octubre de 2013 a través de la URL: http://www.w3.org/TR/rdf-sparql-query/

[28] VisualVM. Disponible en octubre de 2013 a través de la URL: http://visualvm.java.net/

[29] F.Noy. “Guía para crear tu primera ontología”. 2005. Disponible en octubre de 2013 a través de la URL: http://protege.stanford.edu/publications/ontology_development/ontology101-es.pdf

[30] Gruninguer y Fox. “Methodology for the Design and Evaluation of Ontologies”. 1995.

[31] Ontolingua. “Ontolingua System Reference Manual”. 1997. Disponible en octubre de 2013 a través de la URL: http://www-ksl.stanford.edu/htw/dme/thermal-kb-tour/ONTOLINGUA.html

[32] DAML. “DAML Ontology Library”. 2004. Disponible en octubre de 2013 a través de la URL: http://www.daml.org/ontologies

[33] Rosch. “Principles of Categorization”. 1978

[34] Dickinson. “Using Jena with Eclipse”. 2009. Disponible en octubre de 2013 a través de la URL: http://www.iandickinson.me.uk/articles/jena-eclipse-helloworld/