Desarrollo de una aplicación Web 2.0 para la captura y análisis de la valoración del usuario

Proyecto Final De Carrera

Aurelio Segura Maroto

13/09/2010

Directores:

Pedro José Valderas Aranda

Estefanía Serral Asensio

A Baldomero, Concha y Marina

Índice

1. Introducción **4**

1.1. Objetivos 4

1.2. Motivación 4

1.3. ¿Qué es una ontología? 5

1.4. Estructura de la memoria 8

2. Herramientas y Tecnologías **9**

2.1. Herramientas 10

2.1.1. Eclipse 10

2.1.2. Protégé 11

2.1.3. Tomcat 13

2.1.4. Jena 13

2.1.5. Jastor 14

2.2. Tecnologías 14

2.2.1. HTML 15

2.2.2. CSS 15

2.2.3. JEE 15

2.2.4. JSF 16

2.2.5. JFreeChart 17

2.2.6. SPARQL 18

3. Implementación 19

3.1. Modelo 19

3.2. API OWL 20

3.3. Mapeo de OWL a Java 21

3.4. Clases DAO 24

3.5. Login 28

3.6. Estado Actual 28

3.7. Histórico 34

3.8. Estadísticas 41

3.9. Otros aspectos considerados 46

4. Interfaz de Usuario 48

5. Instalación 51

6. Conclusiones 52

Índice de figuras

1. Patrón DAO Diagrama de clases 25
2. Patrón DAO Diagrama de secuencia 27
3. Gráfico de Eventos 41
4. Gráfico de eventos durante la última semana 42
5. Patterns Execution 44
6. Environment Properties 45
7. Devices 46
8. System Information 49
9. Statistics 50

1. Introducción

1.1. Objetivos

El presente documento describe el trabajo realizado en el proyecto: "Desarrollo de una aplicación Web 2.0 para la captura y análisis de la valoración del usuario".

El objetivo principal del proyecto es mostrar la información de un entorno inteligente. Para ello se ha pensado que la forma más cómoda para el usuario era mostrarlo mediante una aplicación web.

En este Proyecto Fin de Carrera se quiere un portal web dinámico para mostrar información sobre un entorno inteligente intentando que cubra todas las necesidades del usuario. El portal que se desea desarrollar servirá para mostrar toda la información relevante para el usuario. Al ser una aplicación web esta información se podrá consultar desde cualquier parte, esto permitirá al usuario saber en cada momento, aunque no se encuentre físicamente en el entorno inteligente, cual es el estado del mismo.

También se quiere crear un completo histórico de los eventos que han ocurrido. De esta manera el usuario podrá ver, de forma cómoda los eventos ocurridos en los últimos días aunque también tendrá la oportunidad de ver los eventos que se ejecutaron en cualquier día en concreto.

Hay que destacar que este Proyecto Final de Carrera es un subproyecto de uno más grande sobre sistemas pervasivos que se está llevando a cabo en el Centro de Investigación de Métodos de Producción de Software (ProS) de la Universidad Politécnica de Valencia.

1.2. Motivación

La principal motivación de este Proyecto Final de Carrera ha sido el interés de la tecnología requerida para realizar adecuadamente el proyecto. Las

tecnologías web han ido creciendo exponencialmente a medida que se ha ido incrementando el uso de internet en la población. Además, con el aumento de la tecnología en los hogares, la domótica es una rama que ha ido aumentando en los últimos años. Este proyecto me ha dado la posibilidad de ampliar mis conocimientos sobre este tema.

Es importante destacar que en este Proyecto Final de Carrera se han utilizado conceptos de alto nivel de abstracción en una ontología. Una ontología busca catalogar la información de los recursos. Con las ontologías, los usuarios organizarán la información de manera que los agentes de software podrán interpretar el significado y, por tanto, podrán buscar e integrar datos mucho mejor que ahora. Gracias al conocimiento almacenado en las ontologías, las aplicaciones podrán extraer automáticamente datos, procesarlos y sacar conclusiones de ellos, así como tomar decisiones y negociar con otros agentes o personas.

Se ha generado un API para usar la ontología en un nivel alto de abstracción, de esta manera el código es menos sensible a cambios en la ontología, mejora su reutilización y mantenimiento.

Por último, quiero destacar que este proyecto ha representado una gran oportunidad para aprender nuevas tecnologías.

1.3. ¿Qué es una ontología?

Las ontologías proceden del campo de la Inteligencia Artificial; son vocabularios comunes para las personas y aplicaciones que trabajan en un dominio. Según el Grupo de Trabajo en Ontologías del consorcio W3C, una ontología define los términos que se usan para describir y representar un cierto dominio. Uso la palabra "dominio" para denotar un área específica de interés (el río Duero, por ejemplo) o un área de conocimiento (física, aeronáutica, medicina, contabilidad, fabricación de productos, etc.). Toda ontología representa cierta visión del mundo con respecto a un dominio. Por ejemplo, una

ontología que defina “ser humano” como “espécimen vivo o muerto correspondiente a la especie Homo sapiens; primate bípedo que pertenece a la familia de los homínidos, como los chimpancés, gorilas y orangutanes” expresa una visión del mundo totalmente distinta a la de una ontología que lo defina como “sujeto consciente y libre, centro y vértice de todo lo que existe; todos tienen la misma dignidad, pues han sido creados a imagen y semejanza de Dios”.

Cualquier persona tiene en su cabeza ontologías mediante las que representa y entiende el mundo que lo rodea. Estas ontologías no son explícitas, en el sentido de que no se detallan en un documento ni se organizan de forma jerárquica o matemática. Todos usamos ontologías en las que Automóvil representa un medio de transporte y tiene cuatro ruedas. ¿Formalizamos este tipo de ontologías? No, sería innecesario: los automóviles son tan habituales que todos compartimos la información de lo que son. Lo mismo sucede cuando pensamos en el dominio familiar: sabemos que una familia se compone de varios miembros, que un hijo no puede tener más de un padre y una madre biológicos, que los padres tienen o han tenido padres... No necesitamos explicitar este conocimiento, pues forma parte de lo que todo el mundo sabe. Sin embargo, cuando se tratan términos poco comunes o cuando se quiere que estos términos sean procesados por máquinas, se precisa explicitar las ontologías; esto es, desarrollarlas en un documento o darles una forma que sea inteligible para las máquinas.

Las máquinas carecen de las ontologías con las que nosotros contamos para entender el mundo y comunicarse entre ellas; por eso necesitan ontologías explícitas. En cuanto dos sistemas de información (sistemas ERP, bases de datos, bases de conocimiento) intentan comunicarse, aparecen problemas semánticos que dificultan o imposibilitan la comunicación entre ellos . Los problemas semánticos son de dos tipos: de dominio y de nombre. Los conflictos de dominio aparecen cuando conceptos similares en cuanto a significado, pero no idénticos, se representan en distintos dominios. Por ejemplo, el concepto representado por Trabajador en una base de datos (BD) puede

corresponder a un trabajador cualificado, mientras que otra BD puede usar Trabajador para cualquier trabajador, sea o no cualificado. Ambos conceptos están muy vinculados, pero no son equivalentes ni deberían mezclarse. Usando ontologías, podría especificarse que el primer concepto corresponde a una especialización del segundo; y un sistema de razonamiento automático basado en ontologías impediría, por ejemplo, que se contratara para tareas cualificadas a trabajadores no cualificados.

En la ingeniería del software, las ontologías ayudan a la especificación de los sistemas de software. Como la falta de un entendimiento común conduce a dificultades en identificar los requisitos y especificaciones del sistema que se busca desarrollar, las ontologías facilitan el acuerdo entre desarrolladores y usuarios.

Exiten diferentes lenguajes para implementar una ontología, en este Proyecto Final de Carrera se ha usado OWL (Ontology Web Language). OWL se compone de las siguientes características:

- Clase: Una clase define un grupo de individuos que se agrupan por que comparten una serie de propiedades. Por ejemplo, Peter y David son miembros de la clase Persona. Las clases pueden ser organizadas en una jerarquia de especialización mediante el uso de subClassOf. Hay una clase, llamada Thing, que es superclase de todas las clases de OWL. Una clase puede contener propiedades, estas propiedades (o atributos) contienen la información de las instacias (en OWL al concepto de instancia se le denomina Individual).

- Propiedad: Las propiedades contienen la información de las instancias de una clase. El dominio de propiedad indica qué clases tendrán esa propiedad. El Rango de una propiedad indica de qué tipo van a ser los datos que va a contener esa propiedad. Hay dos tipos de rangos, el de objects en el cual el rango será una clase de la ontología o el

de datatypes en el cual el rango será un tipo básico (String, int, date, datetime...).

- Individual: Los individuals son instancias de las clases, las propiedades son las que deben diferenciar un individual de otro.

1.4. Estructura de la memoria

El resto de la memoria de este Proyecto Final de Carrera está organizada de la siguiente manera:

- En la **sección 2** se describen las herramientas y tecnologías que han sido utilizadas para la realización de este Proyecto Final de Carrera.
- En la **sección 3** se comenta como se ha realizado la implementación de la aplicación. Este apartado se ha dividido en los aspectos más relevantes.
- En la **sección 4** se muestra, mediante imágenes, el estado final de la aplicación Web desarrollada.
- La **sección 5** es una guía de instalación o puesta en marcha de la aplicación. En la guía se usa el servidor web Tomcat, aunque la aplicación desarrollada en este Proyecto Final de Carrera puede funcionar en cualquier otro servidor de aplicaciones que soporte Java.
- La **sección 6** presenta las conclusiones de este Proyecto Final de Carrera

2. Herramientas y Tecnologías

Durante la realización de este proyecto se han utilizado diversas tecnologías y herramientas. Algunas han sido propuestas por los directores del proyecto y otras han sido incorporados durante la realización del proyecto para solucionar los problemas que surgían en la realización del mismo.

El IDE de desarrollo utilizado para este proyecto ha sido el entorno **Eclipse**, **Eclipse** tiene una fácil integración (mediante plugins) con las tecnologías y herramientas que a priori se iban a utilizar en la realización de este Proyecto Final de Carrera.

Se ha utilizado la herramienta **Protégé** como entorno para manipular y modificar la ontología. Gracias a esta herramienta se han podido modificar algunas clases de la ontología para adecuarlas al proyecto además de que se pueden añadir de forma muy cómoda "Individuals" para poder probar la aplicación.

Al tratarse de una aplicación web, se ha utilizado un **Apache Tomcat** servidor de aplicaciones durante la fase de desarrollo y testeo. Tomcat es un servidor que se puede integrar dentro de Eclipse permitiendo al desarrollador probar los módulos de la aplicación de forma rápida y cómoda.

Dado que para almacenar la información del proyecto se ha elegido utilizar una ontología y la tecnología de desarrollo del proyecto es Java se ha utilizado la librería **Jena** que permite acceder desde Java a la ontología. Además **Jena** incorpora un motor de consultas SPARQL (lenguaje de consultas para ontologías, parecido a SQL) que permite consultar la ontología de manera eficaz. También se ha utilizado una especie de extensión de **Jena** llamada **Jastor** que genera las Clases Java a partir del modelo OWL, de esta manera utilizando tanto Jastor como Jena se pueden recuperar los "Individuals" pero en instancias Java (Es como las herramientas de mapeo objeto-relacional, hibernate, pero para ontologías).

Como se ha comentado, se utiliza tecnología Java para desarrollar el proyecto, en concreto se ha utilizado **JSF** utilizando la implementación MyFaces.

Por último hay que destacar que la aplicación web tiene una parte de estadísticas. Se han utilizado gráficos para ilustrar estas estadísticas dado que para el usuario es más amigable. Para realizar estos gráficos se ha utilizado la librería **JFreeChart** además para poder integrar esta librería con **JSF** de una forma fácil y cómoda se ha utilizado **chartcreator** que añade elementos necesarios para integrar los gráficos en **JSF.**

2.1. Herramientas

2.1.1. Eclipse

Eclipse es una comunidad de código abierto cuyos proyectos se enfocan hacia la creación de una plataforma de desarrollo extensible, tiempos de ejecución y los entornos de aplicación para crear, desplegar y gestionar software durante todo su ciclo de vida. Eclipse es normalmente conocido como un Java IDE, pero es mucho más que un IDE para Java.

La comunidad de código abierto Eclipse tiene más de 60 proyectos de código abierto. Estos proyectos se pueden organizar en 7 categorías diferentes:

1. Enterprise Development
2. Embedded and Device Development
3. Rich Client Platform
4. Rich Internet Applications
5. Application Frameworks
6. Application Lifecycle Management (ALM)

7. Service Oriented Architecture (SOA)

La mayoría de proveedores de soluciones TI, empresas innovadoras, universidades e instituciones de investigación y particulares son los que extienden, apoyan y complementan la plataforma Eclipse.

La Fundación Eclipse es una organización sin ánimo de lucro, una corporación de miembros que alberga los proyectos de Eclipse. La Fundación proporciona servicios para ejecutar la infraestructura de TI, ayuda en los proyectos de código abierto durante el proceso de desarrollo Eclipse y proporciona marketing y negocio para mantener la comunidad Eclipse.

La Fundación Eclipse no es la encargada de desarrollar los proyectos de código abierto. Todo el software de código abierto en Eclipse es desarrollado por los desarrolladores de código abierto que son voluntarios o aportados por las organizaciones o individuos.

Eclipse comenzó como un proyecto de IBM. Fue desarrollado por Object Technology Internacional (OTI) como un remplazo en Java para el VisualAge. En noviembre de 2001, se formó un consorcio para promover el desarrollo de Eclipse como código abierto. La Fundación Eclipse se creó en enero de 2004
.

Para desarrollar el código del proyecto se ha usado el IDE Eclipse 3.4.2 con los plugins de JEE. También se ha instalado un plugin para poder tener subversion integrado en el IDE.

La elección de esta versión de eclipse se debió a que si bien había versiones más nuevas del IDE éstas no permitían de una manera satisfactoria integrarse con las tecnologías que se iban a usar para el desarrollo del proyecto.

2.1.2. Protégé

Protégé es un editor de código abierto y un sistema de adquisición de conocimiento. Proporciona un conjunto de herramientas para construir modelos de do-

minio y aplicaciones basadas en el conocimiento con ontologías. Protégé implementa un amplio conjunto de estructuras de modelado-conocimiento y acciones de apoyo para la creación, visualización y manipulación de ontologías en varios formatos de representación. Se puede personalizar para proporcionar una forma amigable para la creación de modelos de conocimiento y entrada de datos. Además, Protégé se puede ampliar mediante plugins y mediante su API para la construcción de herramientas basadas en el conocimiento y aplicaciones.

En una ontología se describen los conceptos y relaciones que son importantes en un dominio determinado, proporcionando un vocabulario para ese dominio, así como una especificación informatizada del significado de los términos utilizados en el vocabulario. Las ontologías van desde las taxonomías y clasificaciones a esquemas de base de datos. En los últimos años, las ontologías se han adoptado en muchos negocios y en la comunidad científica como una forma de compartir, reusar y procesar el conocimiento del dominio. Las ontologías son fundamentales para muchas aplicaciones como los portales de conocimiento científico, la gestión de la información y la integración de sistemas, comercio electrónico y servicios web semánticos.

La plataforma Protégé admite dos formas principales de ontologías de modelos:

- El editor Protégé-Frames permite a los usuarios construir y poblar las ontologías basado en marcos, de acuerdo con "Open Knowledge Base Connectivity protocol (OKBC)". En este modelo, una ontología consta de un conjunto de clases organizadas en una jerarquía para representar los conceptos del dominio, un conjunto de ranuras asociadas a las clases para describir sus propiedades y relaciones, y un conjunto de dichas clases.

- El editor Protégé-OWL permite a los usuarios construir ontologías para la Web Semántica, en particular para la W3C, "Web Ontology Language

(OWL)"

Protégé fue desarrollado por Stanford Center for Biomedical Informatics Research en Stanford University School of Medicine.

Para la realización de este proyecto se ha utilizado la versión 3.4.4

2.1.3. Tomcat

Apache Tomcat (o Jakarta Tomcat o simplemente Tomcat) es un contenedor de servlets de código abierto desarrollado por Apache Software Foundation (ASF).

Apache Tomcat es una implementación de código abierto de la especificación de las tecnologías Java Servlets y JavaServerPages. Las especificaciones de Java Servlets y Java Server Pages se desarrollan bajo Java Community Process.

Apache Tomcat incluye herramientas de configuración y administración, pero también se puede configurar mediante la edición de archivos XML.

Durante la realización de este proyecto se ha usado Tomcat para realizar las pruebas sobre la aplicación web.

2.1.4. Jena

Jena es un framework Java para la creación de aplicaciones para la Web Semántica. Proporciona un entorno de programación para RDF, RDFS, OWL y SPARQL e incluye un motor de inferencia basado en reglas.

Jena es de código abierto y fue desarrollado a partir del trabajo con HP Labs Semantic Web Programme.

El framework de Jena incluye:

- Un API RDF
- Leer y escribir RDF en RDF/XML, N3 y N-Triples
- Un API OWL
- Almacenamiento en memoria y persistente.
- Motor de consultas SPARQL

2.1.5. Jastor

Jastor es un generador de código abierto en Java que emite JavaBeans desde una ontología Web (OWL) permitiendo acceso seguro y cómodo a todo el RDF almacenado en un modelo Jena. Jastor genera las interfaces Java, las implementaciones, las fábricas y los oyentes según las propiedades y las jerarquías de clase en la Ontología Web.

2.2. Tecnologías

Para la implementación de la aplicación web se han utilizado diferentes tecnologías y lenguajes de programación para implementar la parte gráfica y la lógica del portal web. A continuación se van a comentar las diferentes tecnologías usadas en la implementación de este Proyecto Final de Carrera.

Como servidor de aplicaciones se ha utilizado para hacer las pruebas, como se ha comentado antes, Apache Tomcat 6.0 aunque no debería haber ningún problema en usar otro servidor de aplicaciones (como podría ser jboss por ejemplo).

Para el diseño de la interface de nivel de presentación se ha utilizado HTML combinándolo con hojas de estilo CSS. La tecnología web usada en la aplicación ha sido JEE y JSF.

2.2.1. HTML

HTML, siglas de HyperText Markuo Language (Lenguaje de Marcas de Hipertexto), es el lenguaje de marcado predominante para la construcción de páginas web. Se utiliza para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes. HTML se forma de "etiquetas", rodeadas por corchetes angulares(<,>). HTML también puede describir, hasta un cierto punto, la apariencia de un documento, y puede incluir un script (por ejemplo Javascript), el cual puede afectar el comportamiento de navegadores web y otros procesadores de HTML.

2.2.2. CSS

Las hojas de estilo en cascada (Cascading Style Sheets, CSS) son un lenguaje formal usado para definir la presentación de un documento estructurado escrito en HTML o XML. El W3C (World Wide Web Consortium) es el encargado de formular la especificación de las hojas de estilo que servirán de estándar para los agentes de usuario o navegadores.

La idea que se encuentra detrás del desarrollo de CSS es separar la estructura de un documento de su presentación.

2.2.3. JEE

Java EE (Java Enterprise Edition) es una plataforma de programación para desarrollar software en lenguaje Java que pueda ser ejecutado en un servidor de aplicaciones.

JEE difiere de Java Standard Edition (Java SE) en que añade las librerías que proporcionan funcionalidad para desplegar tolerancia a fallos, distribuida, multi nivel, basado en modular los componentes que se ejecutan en un servidor de aplicaciones.

La plataforma era conocida como Java 2 Enterprise Edition (J2EE) hasta que el nombre fue cambiado a Java EE en la versión 5.

Java EE está definida por sus especificaciones. Al igual que con otras especificaciones de Java Community Process, los proveedores deben cumplir con ciertos requisitos de conformidad para declarar sus productos conformes a Java EE.

Java EE incluye varias API, como JDBC, RMI, email, JMS, servicios web, XML ... y define cómo coordinarlas. Java EE también configura algunas especificaciones únicas para componentes. Éstas incluyen Enterprise JavaBeans, conectores, servlets, portlets, JavaServerPages y varias tecnologías de servicios web. Esto permite a los desarrolladores crear aplicaciones empresariales que son portables y escalables, y que se integran con las tecnologías anteriores. Un servidor de aplicación Java EE puede manejar transacciones, seguridad, escalabilidad, concurrencia y gestión de los componentes desplegados, con el fin de permitir a los desarrolladores concentrarse más en la lógica de negocio en lugar de la infraestructura y las tareas de integración.

2.2.4. JSF

Java Server Faces (JSF) es un framework de aplicaciones web basado en Java destinado a simplificar el desarrollo y la integración de las interfaces de usuario.

JSF está basado en un framework MVC de desarrollo web basado en el componente impulsando el diseño del modelo de la interfaz de usuario. Las solicitudes son procesadas por el FacesServlet, que carga la vista apropiada de la plantilla, construye el árbol de componentes, los procesos de eventos y hace una respuesta (HTML) al cliente. El estado de los componentes de la IU (Interfaz de usuario) se guarda al final de cada solicitud y es restaurado en la siguiente creación de la vista. JSF 2 utiliza facelets para su tecnología de pantalla por defecto.

Existen diversas implementaciones de JSF, para este Proyecto Final de Carrera se ha utilizado Apache MyFaces 2. Además se ha utilizado también Apache TomaHawk que es un conjunto de componentes JSF creado por el equipo de desarrollo de MyFaces.

2.2.5. JFreeChart

JFreeChart es una librería Java de código abierto que facilita a los desarrolladores mostrar gráficos de calidad profesional en sus aplicaciones. JFreeChart incluye:

- Una consistente y bien documentada API, que soporta una amplia gama de tipos de gráficas.
- Diseño flexible que es fácil de ampliar, y que se dirige tanto del lado del servidor como de las aplicaciones del lado del cliente.
- Apoyo a muchos tipos de salida, incluyendo componentes Swing, archivos de imagen y gráficos vectoriales.

Se soportan los siguientes tipos de gráficas:

- Los gráficos XY (línea,dispersión...)
- Gráficos circulares
- Gantt
- Barras (horizontales y verticales, apiladas o independientes)
- Un solo valor (termómetro, brújula, indicador de velocidad)

2.2.6. SPARQL

SPARQL es un lenguaje de consulta RDF, su nombre es un acrónimo recursivo que significa **S**PARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage. Fue estandarizado por RDF Data Access Working Group (DAWG) del World Wide Web Consortium y se considera clave de la tecnología web semántica.

SPARQL permite consultas de patrones triples, conjunciones, disyunciones y patrones opcionales.

3. Implementación

Para este Proyecto Final de Carrera se ha utilizado el dominio de una casa domótica.

3.1. Modelo

Se ha proporcionado por parte de los directores del proyecto la ontología de contexto y el metamodelo de tareas.

El modelo de contexto contiene toda la información sobre el estado de la vivienda domótica además también posee la información sobre los eventos que se han ido ejecutando en el sistema.

Las principales clases de este modelo son:

- ContextModel: Contiene toda la información del sistema, desde esta clase es desde donde se puede navegar a cualquier clase del modelo.
- User: Representa información sobre un usuario, su información personal, la información para poder acceder a la aplicación, la localización dentro de la vivienda, preferencias que tiene (por ejemplo a qué hora se levanta), las políticas que tiene ...
- Location: Representa un sitio de la casa (cocina, comedor, baño...) contiene información de las propiedades del entorno que tiene y de los servicios además de tener información de a qué otras habitaciones se pueden acceder desde ella.
- Service: Representa un servicio de la vivienda, contiene la información del estado del servicio, de los métodos que tiene dicho servicio además de dónde se encuentra ese servicio y a la categoría que representa dicho servicio.

- Event: Representa un Evento que ha ocurrido en el sistema, el modelo tiene 4 tipos de eventos: BehaviourPatternExecution, ContextChange, DeviceEvent, UserAction. Para todos ellos se conoce la fecha y la hora en la que ocurrió el evento. Después para cada evento en particular se conocen otros detalles como por ejemplo para el BehaviourPatternExecution el patrón que se ejecutó, para el UserAction el usuario y el método que se ejecutó ...

- PatternExecution: Representa un patrón de ejecución, es la clase que une a los dos meta modelos dado que en el modelo de características lo que se encuentra es el detalle del Patrón. Un patrón de comportamiento es un conjunto de tareas que se suelen llevar a cabo en contextos similares. Por ejemplo, algunos patrones de comportamiento pueden ser determinados por nuestro ciclo de vida, como leer el correo electrónico y abrir ciertas páginas web tan pronto como tengamos conexión a Internet, mientras que otras son reacciones a cosas que suceden a nuestro alrededor. Para más información sobre un PatternExecution se puede consultar [2]

El modelo de tareas como se ha dicho contiene la información sobre cada patrón de comportamiento definido en el sistema.

3.2. API OWL

API es una clase Java que tiene todo lo necesario para leer un fichero OWL. Para ello usa el framework Jena para poder leer un fichero OWL. El API OWL contiene una serie de métodos que permiten hacer consultas SPARQL directamente sobre el fichero. Dado que las consultas que se tienen que hacer en este proyecto son siempre las mismas se crearon métodos auxiliares que solamente tuvieran estas consultas.

Con el propósito de hacer lo más independiente posible la aplicación web del método de almacenamiento de los datos se ha utilizado el patrón de diseño DAO. La aplicación web lee los datos de las clases DAO creados y éstos cuando se crean son los que llaman al API para recibir los datos. En el caso de que en un futuro los datos se pasen a guardar en otro formato diferente solamente habría que cambiar las clases DAO creadas para que leyeran los datos del nuevo método.

Dado que el fichero donde se encuentra la ontología podía cambiar de localización se decidió cargar la información de éste de forma dinámica. Para ello se ha utilizado un fichero properties para almacenar la información de la URI donde se encuentra la ontología. La librería JENA trabaja preferiblemente con uris y no con rutas físicas para cargar las ontologías (aunque tiene métodos que permiten leer de rutas físicas).

3.3. Mapeo de OWL a Java

Aunque al principio del proyecto no se consideró necesario, conforme avanzaba el mismo se vio que sería interesante poder almacenar la información en objetos Java equivalentes a los de la ontología. De esta manera se podía aumentar el nivel de abstacción usado beneficiando su mantenimiento y reutilización. Un dato importante en este “mapeo” era poder hacerlo de manera automática por si se realizaba algún cambio en la ontología o para poder reutilizarlo en otras ontologías.

La primera intención era, utilizando el API OWL y el código generado del ecore del modelo de contexto hacer a mano el mapeo. La idea fue descartada dado que hacerlo de forma “artesanal” haría que cualquier pequeño cambio en la ontología habría que volver ha realizar a mano el mapeo y aunque para un pequeño cambio a lo mejor no fuera mucho esfuerzo en un cambio con clases y relaciones la dificultad y el tiempo podrían ser un problema.

Finalmente se decidió intentar encontrar algún proyecto que fuera similar a hibernate pero para OWL. Se probaron diferentes proyectos y al final se ha optado por usar una especie de plugin de **Jena** llamado **Jastor** que genera automáticamente las interfaces, las implementaciones, las factorías.... Aparte de éste se probaron otros proyectos:

- Protégé: La versión de Protégé usada en la realización del proyecto tiene un plugin que genera directamente las interfaces y las implementaciones además de una clase Factoría para poder acceder a las instancias del modelo. El problema fue a la hora de usarlo daba errores en algunas librerías del API y no se podía usar. Las nuevas versiones de Protégé no tienen este plugin y la documentación de la página no ayudó a saber qué tipo de incompatibilidad había. La conclusión a la que llegamos que las versiones Java utilizadas por las librerías y las usadas para este proyecto no serían las mismas 1.5 a 1.6 y algún método podría haber cambiado.

- JAOB: JAOB otra librería usada, igual que Protégé también generaba automáticamente las instancias. A diferencia de Protégé esta librería no daba error ni incompatibilidades y parecía recuperar bien las instancias. Al final nos dimos cuenta de que algunos atributos no eran recuperados correctamente y aparecían atributos a null, por eso se decidió descartar esta librería. De todas maneras la librería estaba muy bien documentada, además en su página web había un estudio hecho por parte del autor de diversas librerías que hacían lo mismo que la suya. Este estudio ayudó dado que fue más fácil localizar la herramienta que al final terminamos utilizando.

- Kazuki: En el mismo Protégé también había la opción de generar las clases para Kazuki, aunque este plugin daba error. La idea de probar esta librería fue por la documentación que tenía en la página web. El problema fue similar al ocurrido con el de Protégé, en el código fuente

tenía un error en una llamada a una función de Java que había cambiado y no se podía usar.

- Jastor: Esta librería hace uso del API de Jena. Con ésta se pueden generar automáticamente las interfaces, las implementaciones, las factorías y los oyentes, con lo cual si hay algún cambio en el modelo OWL se podría volver a generar todo de forma rápida. El principal problema de Jastor es la poca documentación que había sobre la librería, la mayoría de la documentación hace referencia a cómo crear objetos en Java con las clases generadas y cómo almacenarlos en un modelo OWL, justamente lo contrario de lo que nosotros queríamos. Al final logramos utilizar la librería para los usos que le queríamos dar (mapear los Individuals en instancias Java). Jastor genera una clase Factory desde donde se pueden acceder a todos los Individuals del modelo, cada clase tiene dos métodos (uno para recuperar una instancia de la clase y otra que recupera la lista de instancias). Con las pruebas realizadas comprobamos que recuperamos satisfactoriamente las instancias sin el problema que teníamos usando JAOB.

Al final el proyecto que utilizamos para hacer el mapeo fue el que generaba correctamente toda la información de la ontología, el principal problema de la mayoría de proyectos vistos (aparte de los citados arriba se vieron otros pero no se probaron ni testearon) es la poca documentación diponible y que la mayoría llevaban varios años abandonados.

Una de las cosas que hay que tener en cuenta a la hora de trabajar con las clases generadas por la herramienta es que en OWL los atributos pueden ser todos multievaluados. Por ejemplo, la clase Event tiene un atributo fecha, que por definición del modelo solamente tendrá 1 solo valor, en un modelo OWL se puede poner cardinalidad para especificar que ese atributo solamente podrá tener 1 valor pero la herramientas de mapeo suele tener limitaciones y algunas de estas limitaciones es la comentada anteriormente, no “saben”

interpretar las cardinalidades con lo cual para la clase todos sus atributos serán una lista y habrá que tener eso en cuenta cuando se quiera leer o modificar un atributo. También debe correr a cargo del desarrollador controlar las cardinalidades de los atributos para que el modelo sea coherente. Con el ejemplo citado anteriormente desde Java se podría crear un Evento en el que tuviera dos fechas, pero al guardar esa instancia en el modelo OWL haría que éste estuviera mal, dado que no cumpliría con especificación del modelo. Un caso similar es cuando se quiere leer un atributo, no es lo mismo esperar 1 valor que una lista de valores y hay que tenerlo en cuenta. Jastor es bastante cómodo en este sentido dado que nunca te vas a encontrar atributos con valores a null, un atributo sin valor se corresponde con una lista vacía. El desarrollador tendrá que controlar manualmente los atributos dependiendo de si es un atributo multievaluado o de si es un atributo con un único valor.

3.4. Clases DAO

Se ha utilizado una versión del patrón DAO (Data Access Object) para recuperar la información del archivo OWL.

El problema que viene a resolver este patrón es el de contar con diversas fuentes de datos (base de datos, archivos, servicios externos, etc). De tal forma que se encapsula la forma de acceder a la fuente de datos. Este patrón surge históricamente de la necesidad de gestionar una diversidad de fuentes de datos, aunque su uso se extiende al problema de encapsular no sólo la fuente de datos, sino además ocultar la forma de acceder a los datos. Se trata de que el software cliente se centre en los datos que necesita y se olvide de cómo se realiza el acceso a los datos o de cual es la fuente de almacenamiento.

Un DAO define la relación entre la lógica de presentación y empresa por una parte y por otra los datos. El DAO tiene un interfaz común, sea cual sea el modo y fuente de acceso a datos.

Algunas características:

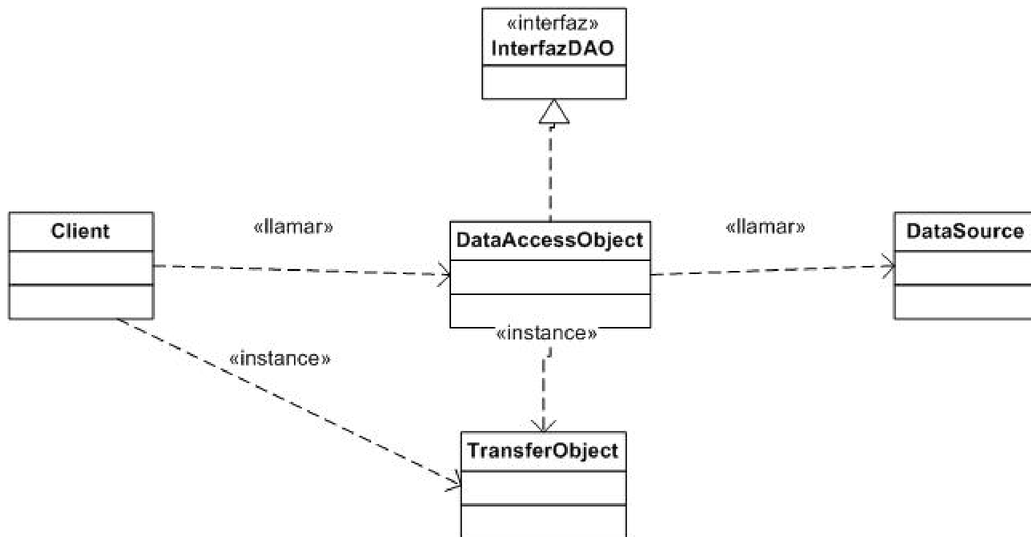

Figura 1: Patrón DAO Diagrama de clases

- No es imprescindible, pero en proyectos de cierta complejidad resulta útil que el DAO implemente un interfaz. De esta forma los objetos cliente tienen una forma unificada de acceder a los DAO.

- El DAO accede a la fuente de datos y la encapsula para los objetos clientes. Entendiendo que oculta tanto la fuente como el modo de acceder a ella.

- El TransferObject encapsula una unidad de información de la fuente de datos.

En el la figura 2 se muestran las interacciones entre los elementos del patrón. En este gráfico el TransferObject se denomina ValueObject. Puede observarse las llamadas que recibe y genera el DAO para una consulta y actualización de datos:

1. El DAO es creado por el cliente (BusinessObject) (llamada 1 en la figura 2).

2. A continuación el cliente solicita los datos al DAO (getData) (2).
3. El DAO responde a la llamada pidiendo los datos a la fuente de datos (2.1).
4. Para cada fila recibida, el DAO crea un TransferObjet (ValueObject de la figura 2) (2.2).
5. El DAO devuelve al cliente el(los) TransferObject (2.3).
6. A continuación el cliente define un TransferObject mediante llamadas a setProperty (3,4, y 5 de la figura 2).
7. En DAO.setData() se solicita (5.1 y 5.2) al TransferObjet o ValueObject (nuestra persona del ejemplo) los datos (edad, sexo, etc.) para realizar el acceso a datos (dataSource.setData()), (5.3).

Para este Proyecto Final de Carrera no se ha usado estrictamente el patrón DAO sino una versión un poco más pequeña. Las clases DAO se encargan de llamar a los métodos del API OWL creado en el proyecto, estos métodos son consultas SPARQL que devuelven la lista con los resultados, la clase DAO trata esa lista y la encapsula de una forma cómoda para del desarrollador. Normalmente los datos son devueltos mediante ArrayList ya sean de String o para datos un poco más complejos lo que contiene el ArrayList un Map con la clave y el valor.

Las clases DAO en principio iban a ser usadas para recuperar los datos de toda la aplicación. Al final, gracias al uso de la librería Jastor sólamente se usan para la parte de histórico y para recuperar información en algunas estadísticas.

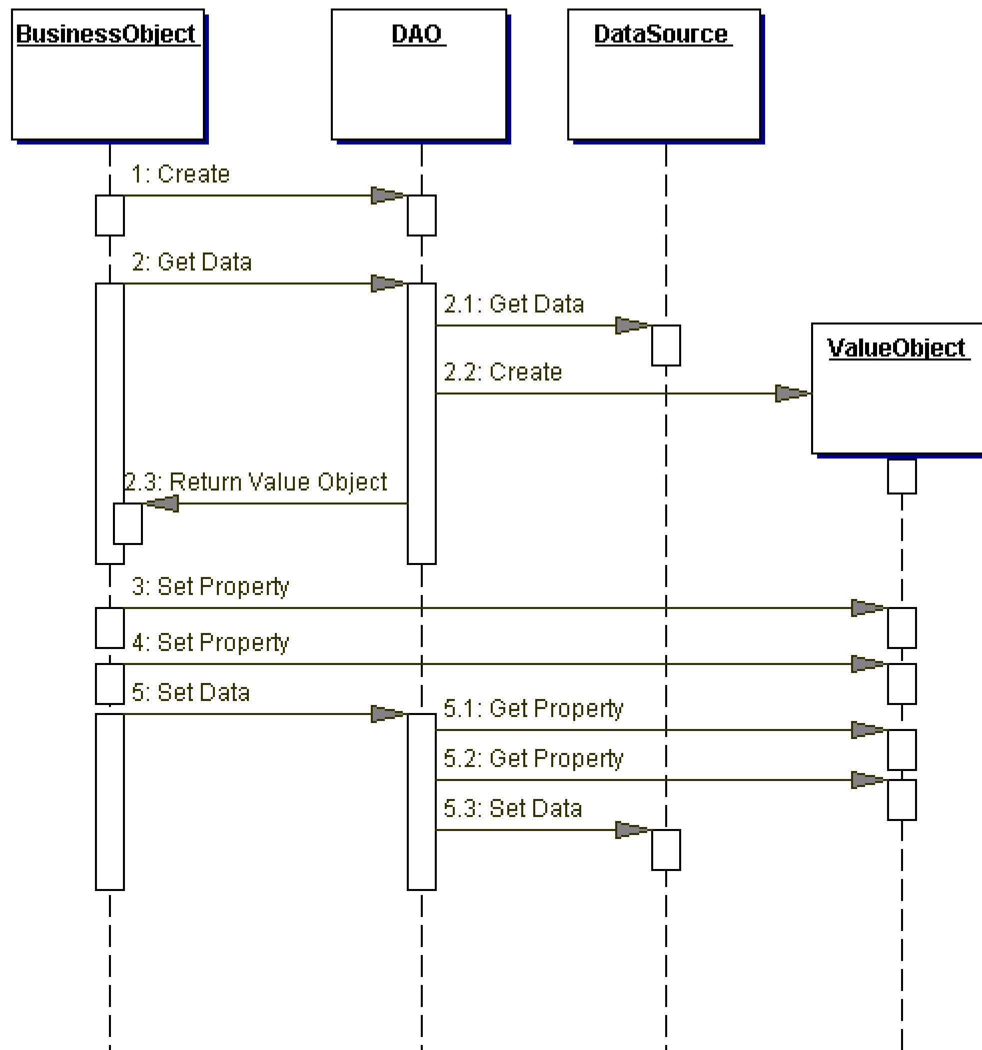

Figura 2: Patrón DAO Diagrama de secuencia

3.5. Login

Todas las partes de la aplicación en las que se muestra algo de información de la vivienda domótica están restringidas su acceso a los usuarios de dicha vivienda. Como se ha comentado en el apartado 3.1 la clase User contiene la información de login y password para poder acceder a la aplicación. Si se intenta acceder a un área restringida de la aplicación saldrá un mensaje de error en la pantalla y no dejara ver la información. Loguearse en la aplicación puede hacerse desde cualquier parte de la misma en la opción que se encuentra en la esquina superior derecha de la pantalla. Una vez logueado en la aplicación (hay que tener en cuenta que el login es case sensitive, no es lo mismo "Peter" que "peter") ya se podrán acceder a todas las partes de la aplicación web.

3.6. Estado Actual

El estado actual de la aplicación muestra el estado actual de la vivienda domótica. Para representar la información se ha utilizado una representación en forma de árbol. En ella se pueden comprobar los servicios que tiene la vivienda (pueden estar agrupados por localización o por categoría, con el botón que se encuentra arriba del árbol se puede cambiar el modo), las propiedades del entorno agrupadas por localización, las propiedades temporales y los datos del usuario que está logueado en la aplicación, estos datos son:

- Con quien está relacionado
- Datos de contacto
- Datos personales
- Políticas que tiene

- Preferencias

Como se ha comentado anteriormente se han utilizado las clases DAO para generar el árbol. Para generar el árbol se ha usado el componente tree2 de la librería TomaHawk, este componente lee un objeto del tipo TreeNode y lo representa.

Hay que tener en cuenta varias cuestiones:

- Habrá que tener un BEAN que genere el TreeNode, el componente JSF se encarga de representarlo en la web, no de crearlo.
- Cada elemento del árbol será un objeto del tipo TreeNodeBase, el constructor de este objeto tiene 3 parámetros (tipo, descripción y si es o no es hoja). El tipo se usa para representar la información y la descripción es lo que se quiere mostrar.
- Para cada tipo de TreeNodeBase dentro habrá que tener un componente facet que es el encargado de representar esa parte del árbol.

```
<t:tree2 id="Tree" showRootNode="false" value="#{
    treejastorCurrentBEAN.treeData}" var="node"
    varNodeToggler="t">
        <f:facet name="ServiceLocation">
                <h:panelGroup>
                        <f:facet name="expand"></f:
                            facet>
                        <f:facet name="collapse"></f:
                            facet>
                        <h:outputText value="#{node.
                            description}" styleClass="
                            nodeFolder"/>
```

```
                    <h:outputText value="(#{node.
                        childCount})" styleClass="
                        childCount" rendered="#{!
                        empty node.children}" />
            </h:panelGroup>
    </f:facet>
    <f:facet name="Service">
            <h:panelGroup>
                    <h:outputText value="#{node.
                        description}" />
            </h:panelGroup>
    </f:facet>
</t:tree2>
```

El código anterior muestra un ejemplo del árbol, se pueden comprobar dos tipos claros de facet, el ServiceLocation y el Service. El primero será un TreeNodeBase con el atributo de hoja a false y el segundo con el atributo de hoja a true.

Dentro del facet se pueden usar otros elementos JSF (en este caso se han usado outputText pero se podrían haber usado imágenes por ejemplo) de esta manera las posibilidades de personalizar el árbol son enormes.

Se han implementado dos BEANs para generar el árbol:

El primer BEAN que se creó se usó leyendo los datos de las clases DAO, una vez recuperados los datos de las clases se procedía a ir construyendo el árbol de arriba a abajo, por ejemplo para construir la parte del árbol de servicios organizados por localización se hacía de la siguiente manera. Primero se recuperaban las localizaciones del sistema utilizando LocationDAO, una vez recuperadas para cada localización se recuperaba la lista de servicios que tenía esa localización. El siguiente fragmento de código muestra como se ha hecho.

```java
//treeData es representa al árbol general
ArrayList<String> locations = new LocationDAO(lb.getApi
    ()).getLocations();
TreeNodeBase ServicesLocations = new TreeNodeBase("
    ServiceLocation", "Services group by Location",
    false);
for (int i = 0; i < locations.size(); i++) {
        ArrayList<Map<String, String>> services= new
            ServiceLocationDAO(lb.getApi(),locations.get
            (i)).getServices();
        TreeNodeBase location = new TreeNodeBase("
            Location", locations.get(i), false);
        if(services.size()>0)
        {
                for (int j = 0; j < services.size(); j
                    ++)
                        location.getChildren().add(new
                            TreeNodeBase("Service",
                            services.get(j).get("name")+
                            " - "+services.get(j).get("
                            state"), true));
        }
        ServicesLocations.getChildren().add(location);
}
treeData.getChildren().add(ServicesLocations);
```

Como se ha comentado anteriormente, mientras el proyecto avanzaba nos dimos cuenta de que igual era cómodo tener los individuals de OWL en instancias de Java. Una vez que se logró hacer funcionar la librería Jastor decidimos cambiar el BEAN creado para el Estado Actual y en vez de usar las clases DAO utilizar la librería Jastor para recuperar las instancias.

De esta manera se creó un nuevo BEAN exactamente igual que el anterior pero utilizando Jastor en vez de las clases DAO.

Ahora en vez de crear la clase DAO y recuperar los datos lo que se hace es llamar al Factory generado por Jastor pasándole el modelo Jena y pidiéndole que devuelva todas las instancias de la clase que necesitamos.

```
List loc =Factory.getAllLocation(model);
TreeNodeBase ServicesLocations = new TreeNodeBase("
   ServiceLocation", "Services group by Location",
   false);
for (int i = 0; i < loc.size(); i++) {
        LocationImpl locimp = (LocationImpl) loc.get(i)
           ;
        Iterator locname=locimp.getName();
        if(locname.hasNext())
        {
                TreeNodeBase location=location=new
                   TreeNodeBase("Location", locname.
                   next().toString(), false);
                Iterator ser=locimp.getServices();
                while(ser.hasNext())
                {
                        ServiceImpl serimpl= (
                           ServiceImpl) ser.next();
                        Iterator sername=serimpl.
                           getName();
                        Iterator serstate=serimpl.
                           getState();
                        String serstring="";
                        if(sername.hasNext())
```

```
                                        serstring+=sername.next
                                           ();
                                if(serstate.hasNext())
                                        serstring+=" - "+
                                           serstate.next();
                                if(serstring.length()>0)
                                        location.getChildren().
                                           add(new TreeNodeBase
                                           ("Service",
                                           serstring, true));
                        }
                        ServicesLocations.getChildren().add(
                           location);
                }
}
treeData.getChildren().add(ServicesLocations);
```

Como se puede comprobar en el código anterior, todos los atributos son listas (Iterators), aunque se sepa que el atributo va a tener un único valor (el name y el state es un ejemplo). El desarrollador tiene que tener esto en cuenta a la hora de acceder a dichos atributos. Si comparamos el código necesario para hacer este BEAN y para hacer el anterior BEAN parece que a primera vista hacerlo usando Jastor es mucho más laborioso que haciéndolo usando las clases DAO. La realidad es que después de generar con Jastor el único código que hay que usar es el puesto arriba, si no usamos Jastor lo primero que tendríamos que hacer es realizar las consultas necesarias en SPARQL, segundo crear las clases DAO y tratar los datos de la consulta y devolverlos en un formato cómodo, por último tendríamos que acceder a esos datos como se ha puesto anteriormente. Al final, dependiendo de lo que queramos mostrar y de la situación se podrá usar un método u otro para recuperar la información.

3.7. Histórico

Una parte de la información más interesante que tiene el modelo es la relativa a los eventos que se han ido ejecutando a lo largo del tiempo en la vivienda. Por esta razón se ha intentado crear una forma cómoda para consultar dicha información.

Por un lado queríamos que la información fuera fácil de acceder, al final optamos por una representación en forma de árbol donde los eventos estuvieran organizados entre los ejecutados hoy, ejecutados ayer, ejecutados durante la última semana y los ejecutados durante el último mes. Además de esa información también se quería que se pudiera consultar cualquier día para ver qué eventos se habían ejecutado ese día en concreto.

En el momento de hacer esta sección nos paramos a pensar con qué método de los dos que teníamos se haría más fácil y cómodo (de las clases DAO o el de Jastor). Como se ha comentado anteriormente se disponía de Jena que permite consultas SPARQL con lo cual es muy fácil y rápido pedirle al API que te devuelva la información de los eventos que han ocurrido en un día en concreto (o en una franja de días), mediante Jastor únicamente tenemos el método para recuperar todas las instancias de una clase, después estas instancias deberán ser tratadas y ordenadas para mostrarse. Dado que el hecho de ir instancia a instancia comprobando que era muy costoso se decidió que para el histórico lo mejor era utilizar el motor SPARQL que al ser un lenguaje de consultas que tenía implementado el order by devolvía la información en el orden exacto en el que se quería mostrar.

Se ha comentado también que el modelo de contexto tiene cuatro tipos diferentes de eventos. En el modelo de contexto estas cuatro clases heredan de Event pero en SPARQL no se podían recuperar los eventos dado que no había ningún individual de la clase Event (eran o BehaviourPatternExecution, o ContextChange, o DeviceEvent, o UserAction) por esta razón se pensó que lo mejor era hacer el histórico para cada tipo de evento. De esta manera

no solamente podíamos decir qué fecha y a qué hora se había ejecutado el evento sino que además podríamos decir qué patrón se había ejecutado, si el evento era un BehaviourPatternExecution o qué usuario había ejecutado un UserAction.

Las consultas SPARQL que permiten recuperar la información son:

```
SELECT ?date ?time ?expaname
WHERE {
        ?event rdf:type pros:BehaviourPatternExecution.
        ?event pros:date ?date. ?event pros:time ?time.
        ?event pros:executedPattern ?expa.
        ?expa pros:name ?expaname.
        FILTER (?date = xsd:date("2010-08-23") ).
}
ORDER BY DESC(?date) DESC(?time)
```

Cuando queríamos recuperar los eventos de un día en concreto (fechas en formato ingles yyyy-mm-dd)

```
SELECT ?date ?time ?expaname
WHERE {
        ?event rdf:type pros:BehaviourPatternExecution.
        ?event pros:date ?date. ?event pros:time ?time.
        ?event pros:executedPattern ?expa.
        ?expa pros:name ?expaname.
        FILTER ((?date < xsd:date("2010-08-24") ) && (?
            date > xsd:date("2010-08-18") ) ).
}
ORDER BY DESC(?date) DESC(?time)
```

Cuando queríamos recuperar de un rango de días.

Como se puede observar, el SPARQL es un lenguaje parecido al SQL. En el SELECT indicamos los datos que queremos recuperar. En la cláusula

WHERE indicamos cómo se accede a esos datos. Por ejemplo en la consulta anterior, en la primera condición del WHERE estamos indicando que queremos recuperar una Instancia cuyo tipo sea un BehaviourPatternExecution. En la segunda, tercera y cuarta condición estamos recuperando propiedades de ese individual. Las dos primeras propiedades son de tipo básico (date y time) la tercera hace referencia a una instancia de otra clase (ExecuttedPattern) en la última condición del WHERE accedemos a esa instancia y recuperamos su nombre. En el WHERE se pueden indicar filtros, en nuestro caso hemos puesto un filtro para que los eventos recuperados estén entre dos fechas. Por último con la cláusula ORDER BY se pueden ordenar los resultados. Hay otros tipos de cláusulas aunque son menos corrientes, por ejemplo en la consulta anterior si un evento no tuviera un ExecuttedPattern no saldría en el resultado de la consulta (o si no tuviera la fecha o la hora a la que se ha ejecutado), si un dato fuera opcional que estuviese o no estuviese se podría indicar mediante la cláusula OPTIONAL.

Una vez añadidos en al API las funciones necesarias para poder recuperar los cuatro tipos de eventos se paso a generar las clases DAO correspondientes.

Por último se pasó ha crear un BEAN parecido al de Estado Actual pero que ahora tuviese la información histórica. Teniendo en cuenta:

- Para las fechas se ha usado una instancia de la clase Calendar de Java
- Para convertir la fecha a formato inglés se ha usado la clase java.SQL.Date cuyo método toString devuelve la fecha en formato inglés.
- Con el método Calendar.add podíamos añadir o restar días al Calendar, de esta manera no hay que preocuparse por el cambio de mes o de año cuando buscas en un rango de días.

El siguiente fragmento de código muestra una parte del BEAN

```
Calendar cal= Calendar.getInstance();
```

```
ArrayList<Map<String, String>> behaviourpatterns;
TreeNodeBase treebehaviourpatterns = new TreeNodeBase("
   BehaviourPatterns", "Behaviour Patterns Execution",
   false);

/*Today*/
TreeNodeBase treebehaviourpatternstoday = new
   TreeNodeBase("Today", "Today", false);
behaviourpatterns=new BehaviourPatternDAO(lb.getApi(),
   new java.sql.Date(cal.getTime().getTime()).toString
   ()).getBehaviourpatterns();
for (int i = 0; i<behaviourpatterns.size(); i++) {
        Map<String, String> dato = behaviourpatterns.
           get(i);
        treebehaviourpatternstoday.getChildren().add(
           new TreeNodeBase("Event", " Date: "+dato.get
           ("date")+" "+dato.get("time").substring(2)+"
            Executed Pattern: "+dato.get("expaname"),
           false));
}
treebehaviourpatterns.getChildren().add(
   treebehaviourpatternstoday);

/*yesterday*/

cal.add(Calendar.DAY_OF_YEAR, -1);
TreeNodeBase treebehaviourpatternsyesterday = new
   TreeNodeBase("Yesterday", "Yesterday", false);
behaviourpatterns=new BehaviourPatternDAO(lb.getApi(),
   new java.sql.Date(cal.getTime().getTime()).toString
```

```
    ()).getBehaviourpatterns();
for (int i = 0; i<behaviourpatterns.size(); i++) {
        Map<String, String> dato = behaviourpatterns.
            get(i);
        treebehaviourpatternsyesterday.getChildren().
            add(new TreeNodeBase("Event", " Date: "+dato
            .get("date")+" "+dato.get("time").substring
            (2)+" Executed Pattern: "+dato.get("expaname
            "), false));
}
treebehaviourpatterns.getChildren().add(
    treebehaviourpatternsyesterday);

cal.add(Calendar.DAY_OF_YEAR, 1);

/*Last Week*/
String dateup=new java.sql.Date(cal.getTime().getTime()
    ).toString();
cal.add(Calendar.DAY_OF_YEAR, -7);
String datedown=new java.sql.Date(cal.getTime().getTime
    ()).toString();;
behaviourpatterns=new BehaviourPatternDAO(lb.getApi(),
    dateup, datedown).getBehaviourpatterns();
TreeNodeBase treebehaviourpatternslastweek = new
    TreeNodeBase("LastWeek", "Last Week", false);
for (int i = 0; i<behaviourpatterns.size(); i++) {
        Map<String, String> dato = behaviourpatterns.
            get(i);
        treebehaviourpatternslastweek.getChildren().add
            (new TreeNodeBase("Event", " Date: "+dato.
```

```
            get("date")+" "+dato.get("time").substring
            (2)+" Executed Pattern: "+dato.get("expaname
            "), false));
}
treebehaviourpatterns.getChildren().add(
   treebehaviourpatternslastweek);
cal.add(Calendar.DAY_OF_YEAR, 7);

/*Last Month*/
cal.add(Calendar.MONTH, -1);
datedown=new java.sql.Date(cal.getTime().getTime()).
   toString();

behaviourpatterns=new BehaviourPatternDAO(lb.getApi(),
   dateup, datedown).getBehaviourpatterns();
TreeNodeBase treebehaviourpatternslastmonth = new
   TreeNodeBase("LastMonth", "Last Month", false);
for (int i = 0; i<behaviourpatterns.size(); i++) {
        Map<String, String> dato = behaviourpatterns.
            get(i);
        treebehaviourpatternslastmonth.getChildren().
            add(new TreeNodeBase("Event", " Date: "+dato
            .get("date")+" "+dato.get("time").substring
            (2)+" Executed Pattern: "+dato.get("expaname
            "), false));
}
treebehaviourpatterns.getChildren().add(
   treebehaviourpatternslastmonth);
cal.add(Calendar.MONTH, 1);
```

```
treeData.getChildren().add(treebehaviourpatterns);
```

En el fragmento anterior se puede observar como lo primero que se hace es recuperar la instancia de calendar, esta instancia se usará para obtener los días de las consultas. La variable treebehaviourpatterns representa al nodo que contendrá el resto de nodos (today, yersterday, last week, last moth). Para recuperar los datos se usa la Clase DAO, las fecha siguen el formato inglés y por eso se ha usado la clase java.sql.Date para generar el String con la fecha. Con el método Calendar.add podemos añadir o restar días, con este método no hay que preocuparse si cambias de mes o de año dado que lo hace automáticamente. Una vez recuperados los datos se pasan a añadirlos al árbol de la misma manera que se ha explicado anteriormente.

La otra parte de la sección es consultar los eventos de cualquier día en concreto. Para ello se ha utilizado un elemento del tipo inputCalendar que está siempre visible para que el usuario de la aplicación pueda pinchar en un día exacto y se muestre la información. El elemento calendario está asociado a un BEAN donde cada vez que se cambia la fecha se genera un nuevo árbol de una forma similar a lo explicado anteriormente y esta información se muestra por pantalla.

El siguiente fragmento de código es sobre el elemento inputcalendar.

```
<h:form id="calendarForm">
        <t:inputCalendar monthYearRowClass="
           yearMonthHeader" weekRowClass="weekHeader"
           currentDayCellClass="currentDayCell" value
           ="#{calendarBEAN.firstDate}" />
        <t:tree2 id="Tree2" showRootNode="false" value
           ="#{calendarBEAN.treeData}" var="node"
           varNodeToggler="t">
                .....
                ....
```

```
                ....
        </t:tree2>
</h:form>
```

3.8. Estadísticas

El último apartado de la aplicación es el apartado de estadísticas, la idea era hacer una sección con algunas estadísticas que pudieran ser interesantes destacar de la aplicación web. Al final lo que tiene más importancia en el modelo son los eventos del sistema con lo cual se ha intentado hacer estadísticas de los mismos. Se han creado los siguientes gráficos:

La figura3 es un gráfico circular que representa los eventos del sistema. En dicho gráfico se puede ver por colores el porcentaje de eventos de cada tipo que hay en la aplicación.

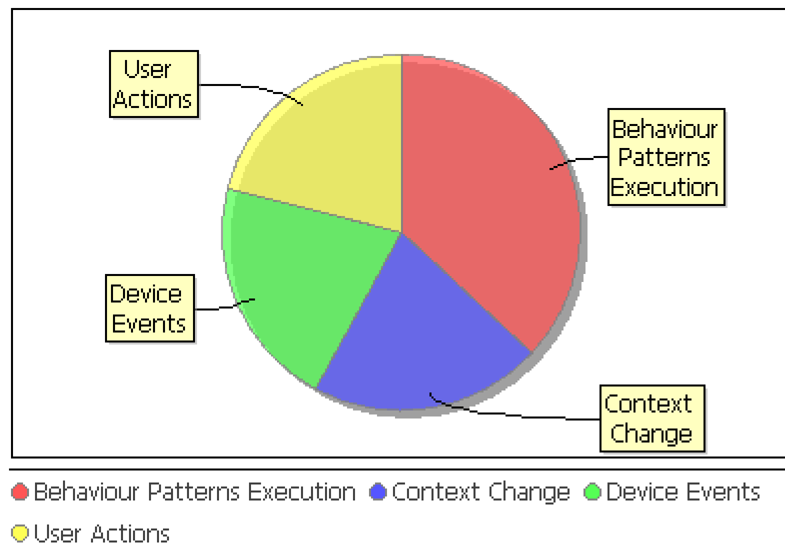

Figura 3: Gráfico de Eventos

La figura 4 hace referencia a los eventos que se han ejecutado en los últimos

7 días mostrando para cada día el número de eventos de cada tipo que se han ejecutado. Se ha hecho de una forma parecida a las estadísticas de visitas de las páginas web (por ejemplo la información de google analytics)

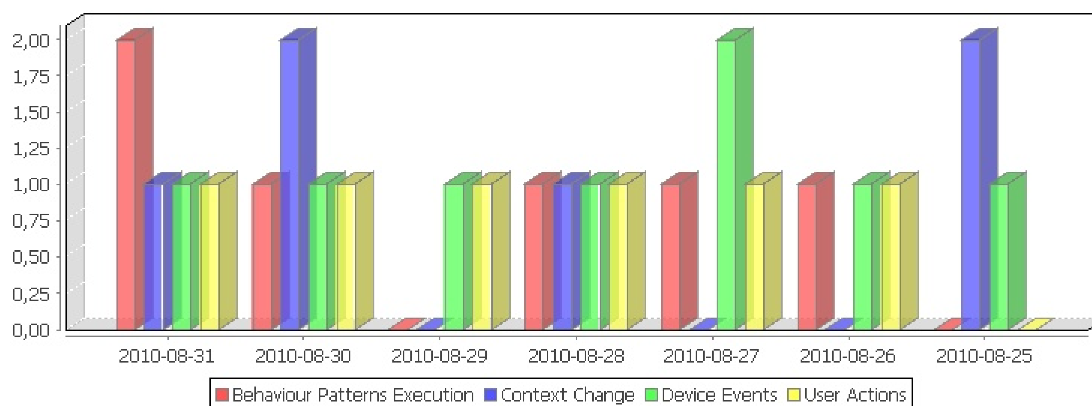

Figura 4: Gráfico de eventos durante la última semana

Para poder generar estos gráficos se ha utilizado la librería JFreeChart y la librería ChartCreator.

La librería ChartCreator añade un nuevo elemento JSF llamado chart, este elemento lee de un BEAN un DataSet con la información del gráfico y a partir de los parámetros que se le especifiquen y usando la librería JFreeChart construye el gráfico.

Esta librería es muy apropiada dado que el usuario puede integrar fácilmente gráficos en una aplicación JSF. Normalmente si no se usara el charCreator habría además de crear el dataset con la información crear el chart correspondiente, este chart tendría que ser convertido a una imagen y que la página llamara automáticamente a la imagen generada. Usando ChartCreator te ahorras la última parte y el desarrollador solamente se tiene que preocupar de rellenar los datos y configurar el chart.

El siguiente código muestra cómo se ha creado el chart de Eventos.

```
DefaultPieDataset pieDataset = new DefaultPieDataset();

List behaviour=Factory.getAllBehaviourPatternExecution(
    model);
List contextc=Factory.getAllContextChange(model);
List device=Factory.getAllDeviceEvent(model);
List useractions=Factory.getAllUserAction(model);

pieDataset.setValue("Behaviour Patterns Execution",
    behaviour.size());
pieDataset.setValue("Context Change", contextc.size());
pieDataset.setValue("Device Events", device.size());
pieDataset.setValue("User Actions", useractions.size())
    ;
```

```
<ch:chart id="chartevents" datasource="#{chartBEAN.
    events}"type="pie" title="Eventos" is3d="false"
    alpha="75" startAngle="90" legend="true"
    legendBorder="true" ></ch:chart>
```

Los siguientes gráficos son del mismo tipo que el la figura 3 pero representa la información de cada tipo de Evento.

La figura 5 representa la proporción de patrones que se han ejecutado en los eventos del tipo Behaviour Pattern Execution.

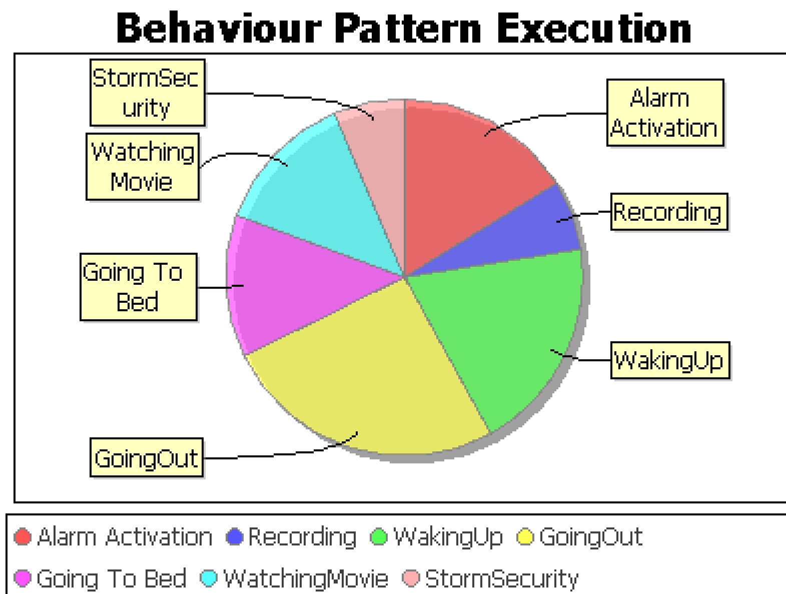

Figura 5: Patterns Execution

La figura 6 muestra la proporción de las propiedades del entorno usadas en los eventos del tipo Context Change.

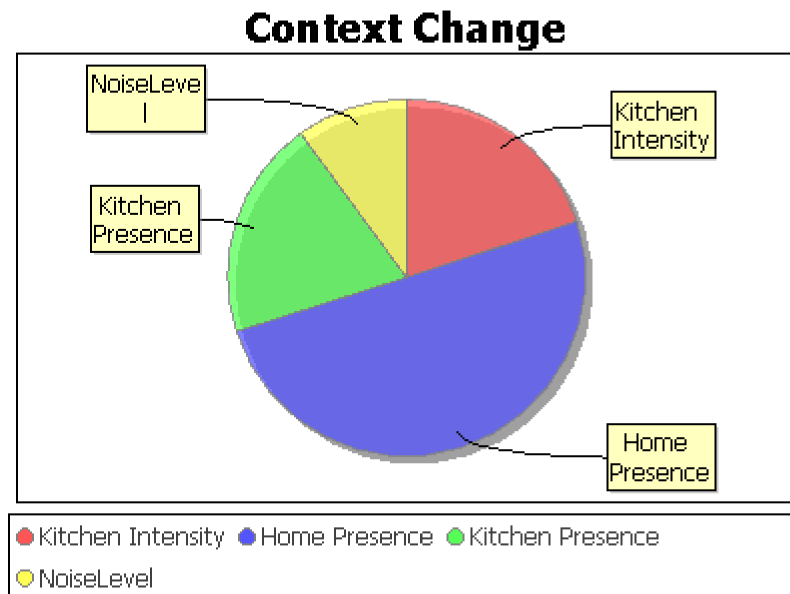

Figura 6: Environment Properties

La figura 7 representa la proporción de los dispositivos utilizados durante la ejecución de los eventos Device Events.

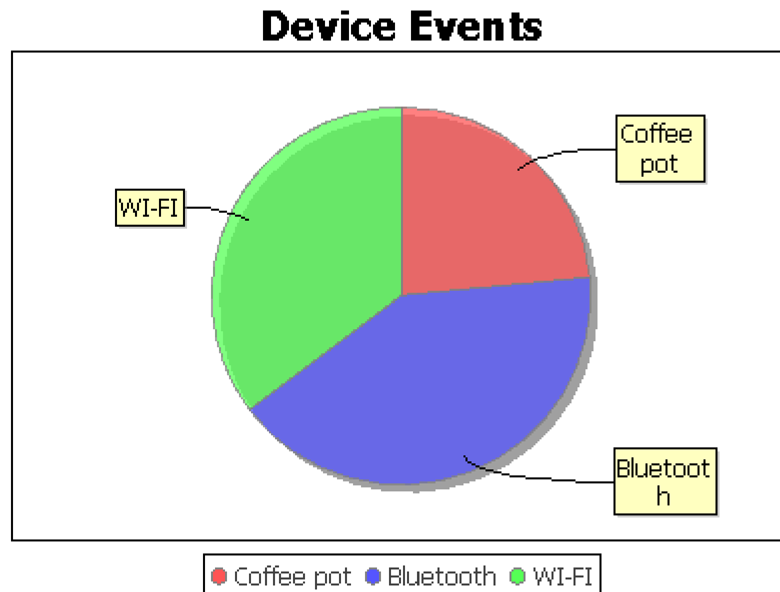

Figura 7: Devices

3.9. Otros aspectos considerados

Aparte de lo comentado anteriormente durante el desarrollo de este Proyecto Final de Carrera se han tenido en cuenta otras cuestiones durante la implementación.

Como se ha dicho en la sección 3.1 la clase User contiene el login y el password de los usuarios que tienen que acceder a la aplicación web (Hay partes de la aplicación que están restringidas a los usuarios).

Para el login se ha creado un BEAN (LoginBEAN.java) que es el que se encarga de decir a la aplicación si hay un usuario conectado y quién es ese usuario.

Utilizando el BEAN y la librería JSTL podemos saber si hay un usuario conectado y si no lo hay avisarle y evitar que entre en las zonas restringidas de la aplicación. En el LoginBean se encuentra la información del usuario,

ésta es necesaria dado que en el Estado Actual se usa esta información para mostrar solamente los datos del usuario que está conectado y no los datos de otro usuario de la aplicación.

Uno de los problemas ha sido poder acceder a los BEANs desde JSTL dado que JSTL y JSF son dos tecnologías diferentes. Para acceder a una variable en JSTL se usa la siguiente sintaxis ${variable} mientras que para acceder a JSF se usa #{variable}. Algunos de los BEANs de la aplicación se van en la variable session (scopesession) de esta manera dado que desde JSTL se puede acceder al scopesession siguiendo navegando a través de la session logramos acceder al BEAN.

```
${sessionScope.loginBean.connected}
```

Durante las primeras pruebas nos dimos cuenta que crear una nueva instancia del API_OWL cada vez que se quería hacer una consulta hacía que la aplicación fuera excesivamente lenta (El tiempo de crear una instancia es menor de medio segundo, pero si durante la carga de una página se creaban 10 o incluso 20 instancias cada vez que se quería leer información la página tardaba bastante en cargar). La solución fue añadir la instancia del API_OWL a un BEAN del tipo session para reutilizarlo y no tener que crearlo siempre. Finalmente se decidió añadirlo al LoginBEAN, de esta manera cuando un usuario se loguea a la aplicación crea su instancia del API_OWL y la reutiliza para todas las consultas.

Para solucionar el problema en el Estado Actual de qué secciones se tienen que mostrar (si hay que mostrar los servicios ordenados por localización o ordenados por categoría) se creó un BEAN de tipo session en el que tuviera las preferencias del usuario que está conectado. La razón de hacer un bueno BEAN y no reutilizar por ejemplo el BEAN de login es simplificar y dividir para que la aplicación sea más fácil de ampliar en el futuro.

4. Interfaz de Usuario

El Proyecto Final de Carrera, tal y como se ha comentado anteriormente, consiste en mostrar información sobre una sistema inteligente. Se ha elegido una tecnología web dado que de esta manera el usuario puede acceder a la aplicación desde cualquier lugar.

El la figura 8 se puede observar como queda la página de la información del sistema. En la parte izquierda de la página, como se ha explicado en la sección 3.6, se muestra la información actual del sistema. El usuario puede ver los servicios de la página ordenados por localización (por defecto) o por categoría. También puede compobar las propiedades del entorno y las temporales. Por último el usuario puede comprobar sus datos, como los datos personales o las preferencias que tiene. En la parte derecha de la misma se encuentra la parte de hístorico, como se ha comentado en la sección 3.7 consta de dos secciones. El árbol superior muestra el histórico de eventos del sistema, como se puede comprobar en la imagen los eventos están agrupados por tipo. A continuación se pueden seleccionar, utilizando el calendario, los eventos de cualquier día en concreto, aunque no se puede poner fechas futuras.

Figura 8: System Information

En la figura 9 se muestra el estado final de la página de estadísticas. En esta sección se puede observar todas las gráficas comentadas que se han realizado en el Proyecto Final Carrera, En la figura se ven las gráficas de Eventos y la de última semana, para más información sobre estas gráficas y otras adicionales se puede consultar la sección3.8.

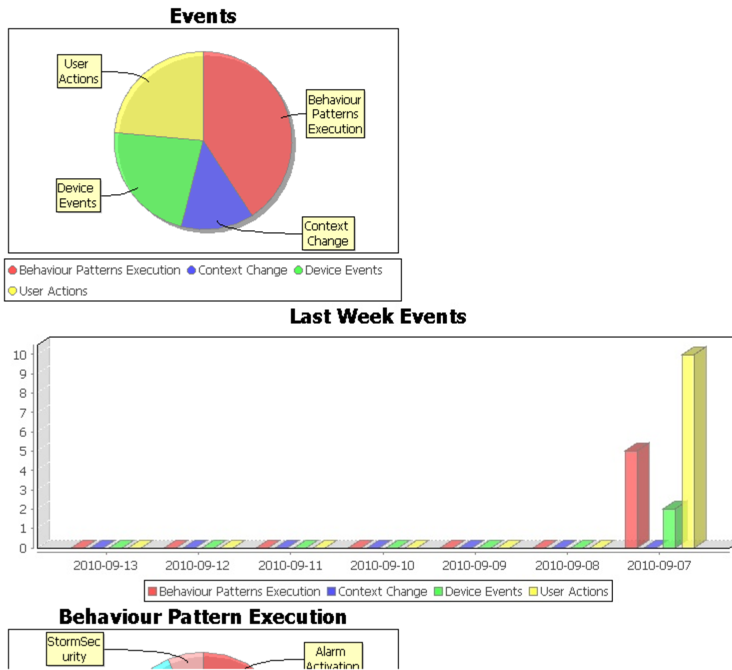

Figura 9: Statistics

5. Instalación

Para poner en marcha la aplicación se necesita un servidor web que soporte Java. La aplicación puede funcionar en distintos servidores de aplicaciones (Tomcat, jBoss ...). A continuación se va a describir los pasos necesarios para utilizar para usar la aplicación junto a Tomcat.

1. Tomcat necesita tener la máquina virtual instalada en el sistema.
2. Una vez instalada la máquina virtual procedemos a crear la variable de entorno JAVA_HOME poniendo como valor la ruta de instalación de Java.
3. Hay que añadir al PATH la ruta JAVA_HOME/bin
4. Copimos el archivo war a la carpeta “TOMCAT/webapps”
5. Ejecutamos en la carpeta “TOMCAT/bin” el archivo **startup.bat** si estamos en un sistema operativo Windows o **startup.sh** si estamos en Linux. Cuando sale un mensaje del estilo “Server startup in 2500ms” indica que el servidor esta ya funcionando. Se puede acceder a la aplicación mediante la dirección http://localhost:8080/PFC

6. Conclusiones

En este Proyecto Fin de Carrera se ha desarrollado un portal web dinámico para mostrar información sobre un entorno inteligente intentando que cubra todas las necesidades del usuario. El portal que se ha desarrollado sirve para mostrar toda la información relevante para el usuario. Al ser una aplicación web esta información se podría consultar desde cualquier parte, esto permite al usuario saber en cada momento cual es el estado del entorno inteligente.

También se ha creado un completo histórico de los eventos que han ocurrido en el entorno inteligente, de esta manera el usuario puede ver, de forma cómoda los eventos ocurridos en los últimos días aunque también tiene la oportunidad de ver los eventos que se ejecutaron en cualquier día en concreto. De esta manera el usuario puede consultar los cambios que ha habido en el entorno inteligente, por ejemplo, si estuviésemos en un dominio de casas domóticas prodríamos comprobar en vacaciones si se ha regado el césped.

Por último, se ha creado una sección el que el usuario puede ver de manera gráfica el porcentaje de eventos que hay de cada tipo en el sistema y la cantidad de eventos de cada tipo que se han ejecutado en el sistema.

El Proyecto Final de Carrera me ha permitido aprender sobre las necesidades que puede tener los sistemas pervasivos además que ha servido para aprender nuevas tecnologías como JSF, OWL, SPARQL que me pueden servir en el futuro profesional.

Hay que destacar que este Proyecto Final de Carrera es un subproyecto de uno más grande sobre sistemas pervasivos que se está llevando a cabo en el Centro de Investigación de Métodos de Producción de Software (ProS) de la Universidad Politécnica de Valencia.

Referencias

[1] SERRAL, E., PÉREZ, F., VALDERAS, P. y PELECHANO, V. Adaptation to User Behaviour by Evolving Systems through Models at Runtime. Submitted to Information and Software Technology, Marzo 2010.

[2] SERRAL, E., VALDERAS, P. y PELECHANO, V. Supporting Runtime System Evolution to Adapt to User Behaviour, in the 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10). 2010. Volume 6051 of Lecture Notes in Computer Science. pp. 378-392. ISBN: 978-3-642-13093-9

[3] THESERVERSIDE. Your Enterprise Java Community [en línea] [fecha consulta 15 07 2010] Disponible en <http://www.theserverside.com/>.

[4] YOSHTEC KB. A small knowledge base [en línea] [fecha consulta 17 07 2010] Disponible en <http://wiki.yoshtec.com/java-owl-api>.

[5] JENA. A Semantic Web Framework for Java [en línea] [fecha consulta 05 07 2010] Disponible en <http://jena.sourceforge.net/>.

[6] JASTOR. Typesafe, Ontology Driven RDF Access from Java [en línea] [fecha conshttp://myfaces.apache.org/ulta 05 08 2010] Disponible en <http://jastor.sourceforge.net/>.

[7] ECLIPSE. Web de la empresa [en línea] [fecha consulta 05 08 2010] Disponible en <http://www.eclipse.org/>.

[8] APACHE MYFACES. JavaServers [en línea] [fecha consulta 07 08 2010] Disponible en <http://myfaces.apache.org/>.

[9] JFREECHART. Java [en línea] [fecha consulta 08 08 2010] Disponible en <http://www.jfree.org/jfreechart/>.

[10] ORACLE. Recurso de la empresa [en línea] [fecha consulta 10 08 2010] Disponible en <http://www.oracle.com/technetwork/java/javaee/javaserverfaces139869.html> y en <http://java.sun.com/blueprints/enterprise/index.html>.