

**UNIVERSIDAD SAN PABLO - CEU**  
**ESCUELA POLITÉCNICA SUPERIOR**  
**INGENIERÍA INFORMÁTICA**



**PROYECTO FINAL DE CARRERA**

**ONTOLOGYTEST: SOFTWARE PARA LA  
EVALUACIÓN DE ONTOLOGÍAS**

**Autor: Sara García Ramos**

**Directores: Abraham Otero Quintana, Mariano Fernández**

Febrero 2009



UNIVERSIDAD SAN PABLO-CEU

ESCUELA POLITÉCNICA SUPERIOR

División de Ingeniería Informática y de Telecomunicación

## Calificación del Proyecto Fin de Carrera

<b>Datos personales del alumno</b>	
D.N.I.	
APELLIDOS	NOMBRE
<b>Directores</b>	
<b>Director 1</b> (tantos como sean los directores)	
D./D <sup>a</sup>	
<b>Tribunal calificador</b>	
<b>Presidente</b>	
D./D <sup>a</sup>	FIRMA
<b>Secretario</b>	
D./D <sup>a</sup>	FIRMA
<b>Vocal</b>	
D./D <sup>a</sup>	FIRMA
<b>Fecha de calificación</b>	
<b>Calificación</b>	

## **RESUMEN**

Este proyecto surge con el objetivo de realizar una herramienta que proporcione soporte para la evaluación de ontologías implementadas en OWL. Para ello se ha desarrollado *OntologyTest*, una herramienta que permite a los usuarios definir un conjunto de requerimientos funcionales para una ontología, comprobar si se verifican de modo automático, e inspeccionar los resultados de dicha verificación. Los requerimientos se persisten en un archivo XML, de tal modo que es posible volver a ejecutarlos (o editarlos) en cualquier momento del ciclo de vida de la ontología.

Cada requerimiento funcional está formado por un escenario y un conjunto de tests a ejecutar sobre ese escenario. El escenario son las condiciones iniciales requeridas para la ejecución de los tests, y cada test verifica uno o varios requerimientos funcionales a evaluar. Los requerimientos pueden definirse empleando una serie de asistentes desarrollados con este fin, o empleando el lenguaje de consultas SPARQL.

La ejecución de los tests se lleva a cabo mediante el razonador Pellet, aunque la arquitectura de la herramienta es extensible y soporta la creación de drivers para otros razonadores. El resultado se presenta como un informe en el que se indican qué tests han pasado y cuáles no, y por qué han fallado estos últimos.

La herramienta ha sido validada con las siguientes ontologías:

- Umls.owl (<http://www.phinformatics.org/Assets/Ontology/umls.owl>).
- Family.owl (<http://www.owl-ontologies.com/family.owl>).
- Ontosem.owl (<http://morpheus.cs.umbc.edu/aks1/ontosem.owl>).

## **ABSTRACT**

This Project aims to develop a tool that provides support for ontologies evaluation (ontologies implemented in OWL). OntologyTest has been developed to achieve this objective. It is a tool that allows users to define a set of functional requirements for the ontology, automatically check if they are fulfilled, and inspect the results of the check. The requirements definition is persisted in an xml file, so it is possible to execute (or edit) them at any time of the ontology's life cycle.

Each functional requirement involves a scenario and a set of tests to execute over the scenario. The scenarios are the initial required conditions to execute the tests, and each test verifies one or more functional requirements to evaluate. The requirements can be defined using a set of tools developed with this purpose, or using the Sparql query language.

Executions of tests are carried out through a Pellet reasoner, though the architecture tool is extensible and supports the creation of new drivers for other reasoners. The result appears like a report for both the tests that have been passed and the tests that have failed, as well as explanations about why the failures have occurred.

The tool has been validated by the following ontologies:

- Umls.owl (<http://www.phinformatics.org/Assets/Ontology/umls.owl>).
- Family.owl (<http://www.owl-ontologies.com/family.owl>).
- Ontosem.owl (<http://morpheus.cs.umbc.edu/aks1/ontosem.owl>).

# ÍNDICE DE CONTENIDOS

<b>1 INTRODUCCIÓN.....</b>	<b>1</b>
1.1 EL PROBLEMA .....	1
1.2 OBJETIVOS .....	2
1.3 DEFINICIONES .....	2
1.3.1 Ontología .....	2
1.3.2 Lenguaje de representación de ontologías: OWL DL.....	4
1.3.3 Lógicas de primer orden .....	5
1.3.4 Lógicas Descriptivas (DLs).....	5
<b>2 ESTADO DE LA CUESTIÓN .....</b>	<b>7</b>
<b>3 ESPECIFICACIÓN DE REQUISITOS .....</b>	<b>11</b>
3.1 REQUISITOS FUNCIONALES .....	11
3.1.1 Añadir ontología .....	11
3.1.2 Crear Proyecto.....	11
3.1.3 Crear test .....	12
3.1.4 Editar test.....	13
3.1.5 Visualizar test.....	13
3.1.6 Importar test.....	13
3.1.7 Crear instancias.....	13
3.1.8 Editar instancias .....	14
3.1.9 Importar instancias.....	14
3.1.10 Visualizar instancias.....	14
3.1.11 Asociar instancias a test.....	14
3.1.12 Ejecutar test.....	14
3.1.13 Mostrar resultados de la ejecución del test .....	14
3.1.14 Guardar Proyecto.....	15
3.1.15 Abrir Proyecto Existente .....	15
3.1.16 Seleccionar Idioma .....	15
3.1.17 Ver Ontología .....	15
3.1.18 Cerrar Proyecto.....	15
3.1.19 Eliminar Proyecto.....	16
3.2 REQUISITOS DE INTERFACES EXTERNAS .....	16
3.2.1 Interfaces de usuario.....	16
3.2.2 Atributos de software .....	28
3.3 REQUISITOS DE RENDIMIENTO .....	29

3.3.1 Capacidad de respuesta de la interfaz de usuario al ejecutar tests .....	29
3.4 REQUISITOS TECNOLÓGICOS .....	29
<b>4 METODOLOGÍA.....</b>	<b>31</b>
4.1 CARACTERÍSTICAS DEL PROYECTO.....	31
4.2 METODOLOGÍA UTILIZADA.....	32
<b>5 ANÁLISIS DE LA TECNOLOGÍA DISPONIBLE .....</b>	<b>34</b>
5.1 PLATAFORMA DE DESARROLLO .....	34
5.2 TECNOLOGÍA PARA EL DESARROLLO DE LA INTERFAZ GRÁFICA .....	36
5.3 SELECCIÓN DEL RAZONADOR .....	36
5.3.1 KAON2 [kaon2] .....	38
5.3.2 PELLET [pellet].....	38
<b>6 DISEÑO.....</b>	<b>42</b>
6.1 MODELO DE DATOS .....	44
6.1.1 Representación de los Tests y de la Ontología.....	45
6.1.2 Ejecución de los Tests.....	52
6.1.3 Gestión del Razonador .....	59
6.2 PERSISTENCIA DE LA INFORMACIÓN.....	68
6.3 INTERFAZ GRÁFICA DE USUARIO .....	71
6.3.1 Estructura de los menús .....	78
<b>7 IMPLEMENTACIÓN.....</b>	<b>81</b>
7.1 ESPECIFICACIÓN DEL FRAMEWORK Y DEL RAZONADOR .....	81
7.1.1 Creación del modelo.....	82
7.1.2 Descripción de los Tests .....	84
7.1.3 Ejecución de los Tests.....	90
7.2 PERSISTENCIA DE LA INFORMACIÓN.....	95
7.3 GESTIÓN DEL RAZONADOR .....	95
7.4 GESTIÓN DE LOS MENÚS EN LA INTERFAZ GRÁFICA .....	99
7.5 CREACIÓN DE UN .JAR CON LA APLICACIÓN EMPAQUETADA .....	100
7.6 INTERNACIONALIZACIÓN .....	102
<b>8 VERIFICACIÓN, VALIDACIÓN Y PRUEBAS.....</b>	<b>108</b>
8.1 CODE REVIEW .....	108
8.2 ANÁLISIS ESTÁTICO DE CÓDIGO .....	109
8.3 VALIDACIÓN .....	110
<b>9 EVOLUCIÓN DEL PROYECTO Y CONCLUSIONES .....</b>	<b>113</b>

<b>10</b>	<b>TUTORIALES .....</b>	<b>115</b>
10.1	CREAR Y TRABAJAR CON UN PROYECTO.....	115
10.1.1	Crear un proyecto nuevo .....	116
10.1.2	Abrir un proyecto existente.....	117
10.1.3	Crear un Test Simple .....	118
10.1.4	Crear un Test Sparql .....	122
10.1.5	Importar un Test .....	124
10.1.6	Editar un test .....	126
10.1.7	Ver un test.....	128
10.1.8	Crear instancias .....	129
10.1.9	Importar Instancias .....	132
10.1.10	Editar instancias.....	133
10.1.11	Ver Instancias.....	133
10.1.12	Ejecutar un test / Conjunto de tests .....	136
10.1.13	Ontología.....	141
10.1.14	Idiomas .....	141
10.2	GUARDAR EL PROYECTO.....	141
10.3	ELIMINAR PROYECTO.....	143
10.4	CERRAR PROYECTO .....	143
<b>11</b>	<b>BIBLIOGRAFÍA .....</b>	<b>144</b>

# ÍNDICE DE FIGURAS

FIGURA 1 ARQUITECTURA DL [RUCKHAUS05] .....	6
FIGURA 2 DETALLE DE LA CABECERA DE LA CREACIÓN DE TESTS .....	17
FIGURA 3 DETALLE DE LA INTERFAZ PARA LA CREACIÓN DE TESTS CON FORMATO AYUDA .....	17
FIGURA 4 DETALLE DE LA CREACIÓN DE TESTS MEDIANTE EL FORMATO TEXTO .....	18
FIGURA 5 DETALLE DE UN TEST SPARQL .....	19
FIGURA 6 DETALLE DE LA CABECERA DE LAS INSTANCIAS.....	19
FIGURA 7 DETALLE DE LA CREACIÓN DE INSTANCIAS MEDIANTE LA OPCIÓN AYUDA.....	20
FIGURA 8 DETALLE DE LA CREACIÓN DE INSTANCIAS MEDIANTE EL FORMATO AYUDA.....	21
FIGURA 9 DETALLE DE LA OPCIÓN IMPORTAR TEST S E IMPORTAR INSTANCIAS .....	22
FIGURA 10 DETALLE DEL PASO INTERMEDIO PARA IMPORTAR TESTS .....	22
FIGURA 11 DETALLE DE LA LISTA DE TESTS SIMPLES .....	23
FIGURA 12 DETALLE DEL SUBMENÚ PARA LOS TESTS Y LAS INSTANCIAS .....	23
FIGURA 13 DETALLE DEL MENÚ DE TESTS CON LA OPCIÓN DE VER .....	23
FIGURA 14 DETALLE DEL PASO INTERMEDIO PARA VER UN TEST.....	24
FIGURA 15 DETALLE DE LA VISTA DE UN TEST .....	25
FIGURA 16 DETALLE DEL MENÚ CON LA OPCIÓN DE EJECUTAR TESTS.....	26
FIGURA 17 DETALLE DE LA SELECCIÓN DE TEST PARA SU EJECUCIÓN .....	26
FIGURA 18 DETALLE DE LA VISTA DE LA EJECUCIÓN DE LOS TESTS .....	27
FIGURA 19 ARQUITECTURA DEL RAZONADOR PELLET [PARSIA,SIRIN] .....	39
FIGURA 20 DIAGRAMA DE PAQUETES DE LA APLICACIÓN.....	44
FIGURA 21 DIAGRAMA DE CLASES DEL PAQUETE DEL MODELO.....	45
FIGURA 22 DIAGRAMA CON CLASES QUE REPRESENTAN LOS TESTS Y LA ONTOLOGÍA.....	46
FIGURA 23 DETALLE DE LA CLASE SCENARIOTEST .....	48
FIGURA 24 DETALLE DE LAS CLASES SPARQLQUERYONTOLOGY Y QUERYONTOLOGY .....	49
FIGURA 25 DETALLE DE LA CLASE INSTANCIAS .....	50
FIGURA 26 DETALLE DE LAS CLASES CLASSINSTANCES Y PROPERTYINSTANCES .....	50
FIGURA 27 DETALLE DE LA CLASE VALIDARTESTS.....	51
FIGURA 28 DETALLE DE LA CLASE COLLECTIONTEST.....	52
FIGURA 29 DIAGRAMA DE CLASES DEL MÓDULO DE EJECUCIÓN DE LOS TESTS.....	53
FIGURA 30 DETALLE DE LA INTERVENCIÓN DE LOS PATRONES COMMAND Y TEMPLATE METHOD .....	55
FIGURA 31 DETALLE DEL PATRÓN COLLECTING PARAMETER .....	57
FIGURA 32 DETALLE DE LA CLASE ONTOLOGYTESTRESUTL.....	57
FIGURA 33 DETALLE DE LA CLASE ONTOLOGYTESTFAILURE .....	58
FIGURA 34 DETALLE DE LA CLASE EXECQUERYSPARQL.....	59
FIGURA 35 DIAGRAMA DE CLASES DEL RAZONADOR.....	60
FIGURA 36 DETALLE DE LAS CLASES QUE IMPLEMENTAN EL RAZONADOR.....	61



FIGURA 37 DETALLE DE LA CLASE REASONER .....	62
FIGURA 38 DETALLE DE LA INDEPENDENCIA DEL RAZONADOR .....	68
FIGURA 39 DETALLE DE LA INTERFAZ IOMANAGERIMPLEMENTATION.....	70
FIGURA 40 DIAGRAMA DE CLASES PARA LA PERSISTENCIA.....	71
FIGURA 41 DETALLE DE LA INTERFAZ PRINCIPAL.....	72
FIGURA 42 DETALLE DE LA LISTA DE TESTS SIMPLES .....	73
FIGURA 43 DETALLE DEL RESULTADO DE LA EJECUCIÓN DE LOS TESTS .....	73
FIGURA 44 DETALLE DEL SUBMENÚ PARA LOS TESTS.....	74
FIGURA 45 DETALLE DEL SUBMENÚ PARA LAS INSTANCIAS .....	74
FIGURA 46 DETALLE DEL PROCESO DE CREACIÓN DE UN TEST PARA UN USUARIO NOVEL .....	75
FIGURA 47 DETALLE DEL PROCESO DE CREACIÓN DE UN TEST PARA UN USUARIO EXPERTO.....	76
FIGURA 48 DETALLE DE LA REPRESENTACIÓN GRÁFICA DEL RESULTADO DE LA EJECUCIÓN DE LOS TESTS ..	77
FIGURA 49 DETALLE DE LA OPCIÓN NUEVO PROYECTO.....	78
FIGURA 50 DETALLE DE LA OPCIÓN CREAR TEST.....	79
FIGURA 51 DETALLE DE LA OPCIONES PARA LAS INSTANCIAS .....	79
FIGURA 52 DETALLE DE LAS OPCIONES PARA LA EJECUCIÓN .....	80
FIGURA 53 DIAGRAMA DE PAQUETES .....	82
FIGURA 54 DETALLE DE LAS CLASES ENCARGADAS DEL RAZONADOR .....	96
FIGURA 55 DETALLE DE LA CLASE REASONER .....	97
FIGURA 56 DETALLE DEL ARCHIVO MANIFEST.MF.....	101
FIGURA 57 ESTRUCTURA DE DIRECTORIOS A CREAR .....	101
FIGURA 58 CREACIÓN DEL ARCHIVO APPLICATION.JAR.....	101
FIGURA 59 DETALLE DEL CONTENIDO DEL ARCHIVO ONE-JAR-BOOT-0.96.JAR .....	102
FIGURA 60 PRIMER PASO DEL ASISTENTE DE NETBEANS PARA LA INTERNACIONALIZACIÓN .....	103
FIGURA 61 SELECCIÓN DE LAS CLASES A INTERNACIONALIZAR .....	103
FIGURA 62 DETALLE DEL PROCESO DE INTERNACIONALIZACIÓN .....	104
FIGURA 63 ÚLTIMO PASO DEL PROCESO DE INTERNACIONALIZACIÓN .....	105
FIGURA 64 DETALLE DEL ARCHIVO SPANISH.PROPERTIES .....	105
FIGURA 65 DETALLE DEL ARCHIVO ENGLISH.PROPERTIES .....	106
FIGURA 66 DETALLE DE LA INTERFAZ GRÁFICA EN INGLÉS .....	107
FIGURA 67 DETALLE DE LA INTERFAZ GRÁFICA EN ESPAÑOL .....	107
FIGURA 68 MENÚ PRINCIPAL DE LA APLICACIÓN.....	115
FIGURA 69 OPCIONES DEL MENÚ ARCHIVO.....	115
FIGURA 70 PASO 1 DE LA CREACIÓN DE UN PROYECTO.....	116
FIGURA 71 PASO 2 DE LA CREACIÓN DE UN PROYECTO.....	117
FIGURA 72 MENÚ PRINCIPAL UNA VEZ CARGADO EL PROYECTO .....	117
FIGURA 73 PROCESO DE CARGA DE UN PROYECTO .....	118
FIGURA 74 CREACIÓN DE UN TEST SIMPLE .....	119
FIGURA 75 DETALLE DE LA CREACIÓN DE UN TEST DE INSTANCIACIÓN.....	119

FIGURA 76 FORMATO TEXTO PARA LA CREACIÓN DE UN TEST DE INSTANCIACIÓN.....	120
FIGURA 77 DETALLE DEL ALMACENAMIENTO DE UN TEST SIMPLE .....	121
FIGURA 78 PANEL PARA AÑADIR UN COMENTARIO A UNA CONSULTA .....	121
FIGURA 79 CREACIÓN DE UN TEST SPARQL .....	122
FIGURA 80 NUEVO TEST SPARQL.....	123
FIGURA 81 IMPORTAR UN TEST.....	124
FIGURA 82 PROCESO DE IMPORTAR UN TEST .....	125
FIGURA 83 VISTA COMPLETA DE UN TEST .....	126
FIGURA 84 EDITAR UN TEST .....	127
FIGURA 85 SELECCIÓN DEL TEST A EDITAR .....	128
FIGURA 86 VER TEST .....	128
FIGURA 87 CREAR NUEVO CONJUNTO DE INSTANCIAS.....	129
FIGURA 88 CREACIÓN DE UN CONJUNTO DE INSTANCIAS.....	130
FIGURA 89 DETALLE DEL PROCESO DE CREACIÓN DE INSTANCIAS .....	131
FIGURA 90 ASOCIAR INSTANCIAS A UN TEST.....	132
FIGURA 91 IMPORTAR INSTANCIAS.....	133
FIGURA 92 EDITAR INSTANCIAS .....	133
FIGURA 93 VER INSTANCIAS .....	134
FIGURA 94 SELECCIÓN DEL CONJUNTO DE INSTANCIAS A VER .....	135
FIGURA 95 VISTA DE UN CONJUNTO DE INSTANCIAS .....	135
FIGURA 96 EJECUTAR TESTS .....	136
FIGURA 97 SELECCIÓN DE LOS TESTS A EJECUTAR .....	137
FIGURA 98 VISTA DE LA APLICACIÓN TRAS LA EJECUCIÓN DE LOS TESTS .....	138
FIGURA 99 ÁRBOL RESULTADO DE LA EJECUCIÓN DE LOS TESTS .....	139
FIGURA 100 RESULTADO DE LA EJECUCIÓN PARA UN TEST .....	140
FIGURA 101 VER ONTOLOGÍA .....	141
FIGURA 102 SELECCIONAR IDIOMA.....	141
FIGURA 103 GUARDAR PROYECTO.....	142
FIGURA 104 GUARDAR UN PROYECTO EN UN DIRECTORIO ESPECÍFICO.....	143

## ÍNDICE DE TABLAS

TABLA 1 DESCRIPCIÓN DE HERRAMIENTAS PARA LA EVALUACIÓN DE ONTOLOGÍAS .....	9
TABLA 2 TIPOS DE TESTS .....	13
TABLA 3 RAZONADORES VERSUS CRITERIOS DE SELECCIÓN.....	37
TABLA 4 ONTOLOGÍAS PARA EL DESARROLLO DE META-PRUEBAS .....	110
TABLA 5 SÍNTESIS DE LAS META-PRUEBAS PARA EL PROYECTO Y LAS INSTANCIAS .....	111
TABLA 6 SÍNTESIS DE LAS PRUEBAS PARA LOS TESTS.....	112

## **ÍNDICE DE DIAGRAMAS**

DIAGRAMA 1 DIAGRAMA DE SECUENCIA PARA LA EJECUCIÓN DE LOS TESTS.....	56
DIAGRAMA 2 DIAGRAMA DE SECUENCIA PARA LA CARGA DEL RAZONADOR.....	62

## 1 Introducción

El proyecto se enmarca dentro de los sistemas de evaluación de ontologías. Su propósito es proporcionar una herramienta que permita que un usuario pruebe y evalúe una ontología desarrollada en lenguaje OWL mediante la generación de tests de pruebas. Estos tests de pruebas consisten en un conjunto de consultas realizadas por el usuario y en un conjunto de resultados que el usuario espera obtener. La herramienta evalúa dichas consultas y produce un mensaje de error en caso de que el resultado esperado no coincida con el obtenido tras la evaluación, o uno de aceptación en caso contrario.

Los tests de pruebas engloban tanto tests simples como tests más complejos. Los primeros son aquellos que no utilizan el lenguaje SPARQL para realizar las consultas, ya que requieren del usuario un conocimiento mínimo sobre las ontologías y sobre la forma de evaluarlas y extraer información. Los segundos son aquellos que únicamente utilizan el lenguaje SPARQL para realizar las consultas y que, por lo tanto, necesitan del usuario un conocimiento más profundo sobre éste lenguaje.

Los tests que se van a realizar sobre la ontología van a ser aquellos que prueben los distintos mecanismos de inferencia sobre la *ABox* (el conjunto de afirmaciones), a partir de una base de hechos introducido por el usuario en forma de instancias, bien desde la herramienta desarrollada, que permite crear y asociar conjuntos de instancias a los tests y a la ontología, o bien desde otros programas de desarrollo de ontologías como por ejemplo, Protegé.

### 1.1 El problema

Actualmente existen una serie de herramientas que permiten que un usuario evalúe su ontología comprobando si la sintaxis es correcta y detectando inconsistencias.

Los sistemas que permiten realizar consultas SPARQL sobre las ontologías son muy limitados, ya que las consultas se realizan de una en una y por cada cambio que se hace en la ontología es necesario volver a realizar la consulta para comprobar éstos cambios.

Hoy en día, este tipo de evaluación se queda corto para el desarrollo industrial de ontologías. Es necesario un sistema que no se quede sólo en la sintaxis, sino que permita que el usuario cree conjuntos de pruebas con los que trabajar, almacenando todas las consultas SPARQL que desee sin necesidad de escribirlas de nuevo a cada cambio que se haga en la ontología o en su conjunto de instancias.

## **1.2 Objetivos**

1. Desarrollar una herramienta que permita que un usuario final pueda realizar, almacenar y modificar conjuntos de test sobre una ontología.
2. Probar una ontología de la misma forma que un programador puede evaluar el software que realiza, mediante pruebas que escribe a priori del desarrollo del software y que, una vez desarrollado éste, ejecuta para comprobar si existen errores.
3. Crear conjuntos de pruebas con esta herramienta que pasará sobre su ontología para obtener un informe de errores. La ejecución de los tests utilizará la *ABox*.

## **1.3 Definiciones**

### **1.3.1 Ontología**

Las ontologías son componentes básicos en la reutilización y compartición de conocimiento. A menudo son utilizadas para permitir la interacción entre distintos sistemas. Se asume la definición de ontología que utilizaron Studer y colaboradores (1998) y que se basa en las definiciones de Gruber (1993) y Borst (1997).

*“Una ontología es una especificación formal y explícita de una conceptualización compartida.” (...)* “Conceptualización se refiere a un modelo abstracto sobre algún fenómeno en el mundo habiendo identificado los conceptos relevantes de dicho fenómeno. Explícita significa que el tipo de conceptos usados, así como las restricciones en su utilización, están explícitamente definidos. Formal se refiere al hecho de que las ontologías deben poder ser leídas por máquinas.

*Compartida refleja el hecho de que una ontología recoge el conocimiento consensuado”.*

Esta definición se refiere a ontologías formales (aquellas que pueden ser usadas por las máquinas). No todas las ontologías deben ser formales: hay ontologías que se expresan con una forma restringida y estructurada del lenguaje natural, pero éste proyecto sólo considerará las ontologías formales, entendiendo éstas como aquellas que tienen semánticas formales, es decir, semánticas que describen el significado del conocimiento de una forma precisa. La palabra precisa significa aquí, que la semántica no se refiere a opiniones ni intuiciones, y que las máquinas y las personas deben interpretar los términos de la misma forma. Disponer de una semántica formal resulta indispensable para implementar sistemas de inferencia o de razonamiento automático.

Este proyecto evalúa el segundo punto indicado arriba, es decir, le permitirá al usuario realizar pruebas para determinar, mediante consultas, si el significado que asume para los términos incluidos en su ontología se corresponde con la definición de ésta.

Toda ontología modela, mediante conceptos y relaciones, un dominio o un área de conocimiento. Las ontologías se suelen representar mediante sus componentes básicos, que se describen a continuación (clasificación propuesta inicialmente por Gruber (1993) y refinada por Gómez Pérez y colegas (2003)):

Clases: suelen representar conceptos, tanto abstractos como concretos. Se suelen organizar en taxonomías a través de las cuales se pueden aplicar los mecanismos de herencia.

Relaciones: las relaciones indican el tipo de asociación o interacción entre los conceptos del dominio.

Atributos: los atributos son las propiedades de los conceptos y, por tanto, matizan su definición. Los atributos pueden ser heredados de un concepto padre o propios del concepto.

Funciones: son un tipo especial de relación en donde el enésimo elemento es único para los  $n-1$  elementos precedentes.

Instancias: una instancia representa la ejemplificación en el mundo real de un concepto dado.

Axiomas: son expresiones que son ciertas siempre. Normalmente se utilizan para representar conocimiento que no puede ser definido formalmente de otro modo. Los axiomas, como en lógica, se utilizan para verificar la consistencia de la ontología o del conocimiento que almacena pero también se pueden utilizar para completar la definición del significado de los componentes estableciendo restricciones, deduciendo nueva información, etc.

### *1.3.2 Lenguaje de representación de ontologías: OWL DL*

El Lenguaje de Ontologías Web (OWL) está diseñado para ser usado en aplicaciones que necesitan procesar el contenido de la información en lugar de únicamente representar información para los humanos. OWL facilita un mejor mecanismo de interpretabilidad de contenido Web que los mecanismos admitidos por XML, RDF, y esquema RDF (RDF-S) tal cual, proporcionando vocabulario adicional junto con una semántica formal. OWL tiene tres sublenguajes, con un nivel de expresividad creciente: OWL Lite, OWL DL y OWL Full, aunque OWL DL será en el que se centra el presente trabajo, ya que las ontologías que pueden evaluarse con la herramienta desarrollada tendrán que estar escritas en este lenguaje.

OWL DL permite obtener la máxima expresividad conservando completitud computacional (el procedimiento de deducción es computable para todas las consultas posibles) y resolubilidad (el procedimiento de deducción requiere un tiempo finito para todas las consultas posibles). OWL DL incluye todas las construcciones del lenguaje de



OWL, pero sólo pueden ser usadas usadas 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 se denomina de esta forma debido a su correspondencia con la lógica de descripción [w3.org]

### 1.3.3 Lógicas de primer orden

De acuerdo con este formalismo, una ontología corresponde a un conjunto de fórmulas de la Lógica de Primer Orden. Ese conjunto de fórmulas se denominan teoría.

Las interpretaciones legales de una ontología son todos los modelos de la teoría de la Lógica de Primer Orden. Por lo tanto, una ontología o teoría se compondrá de un conjunto de axiomas y reglas de inferencia que caracterizan a un conjunto de teoremas. Un ejemplo de lenguaje de ontologías de lógica de predicados de primer orden es KIF [Genesereth y Fikes,92].

### 1.3.4 Lógicas Descriptivas (DLs)

Son una familia de formalismos basados en lógica para la representación de conocimiento. De hecho, las DLs son subconjuntos de la LPPO.

- Describen el dominio en función de conceptos (clases), roles (relaciones) e individuos (instancias).

Un lenguaje de DL está formado por:

- Un conjunto de constructores para definir conceptos y roles complejos a partir de otros más simples.
- Conjuntos de axiomas para proporcionar aserciones acerca de conceptos, roles e individuos.
- Constructores que incluyen booleanos (and, or, not).

Tal y como muestra la Figura 1 una base de conocimientos en DL está formada por un esquema (*TBox*) y un conjunto de afirmaciones que respetan el esquema (*ABox*).

La *TBox* es un conjunto de axiomas en los que se especifican inclusiones y equivalencias de clases y propiedades. Por ejemplo:

- $C \sqsubseteq D$  (inclusión de clase)
- $C \equiv D$  (equivalencia de clase)
- $R \sqsubseteq S$  (inclusión de propiedad)
- $R \equiv S$  (equivalencia de propiedad)

La *ABox* es un conjunto de axiomas en los que se especifican instanciaciones sobre clases e individuos.

- $x \in D$  (instanciación de clase)
- $\langle x, y \rangle \in R$  (instanciación de propiedad)

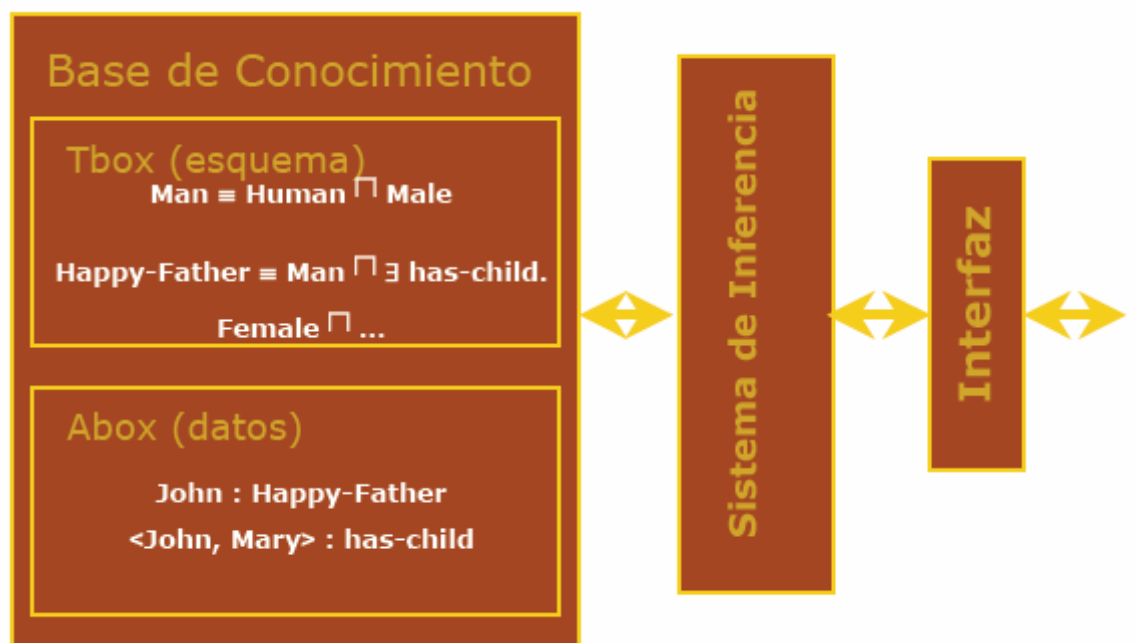


Figura 1 Arquitectura DL [Ruckhaus05]

## 2 Estado de la cuestión

Actualmente son dos los enfoques (complementarios) más extendidos que hay sobre lo que es la evaluación de ontologías.

El primero, propone evaluar una ontología para saber si ésta va a ser o no apropiada para aplicarse a un propósito particular. Este punto de vista está motivado por el hecho de que una de las características principales y más importantes de las ontologías es su reutilización. En consecuencia, se establece la necesidad de sistemas que ayuden a los usuarios a elegir las ontologías más apropiadas para nuevos proyectos. Entre los sistemas de evaluación que siguen esta aproximación están, por ejemplo, **OntoMetric**, que determina si una ontología es o no apropiada para aplicarse a un determinado tema; y **OntoManager**, que determina la veracidad y la aplicabilidad de una ontología para un determinado dominio. De esta forma los usuarios tienen sistemas de ayuda para determinar cómo de apropiada es una ontología para un nuevo sistema.

El segundo enfoque propone evaluar la ontología durante todo su ciclo de vida, desde el punto de vista de la representación del conocimiento, para evitar los distintos tipos de problemas que pueden surgir y hacer que la ontología no sea correcta. Estos problemas son [Corcho et al.]:

### 1. Inconsistencia

- Circularidad
- Errores de particiones
  - Clases comunes y descomposiciones disjuntas y particiones
  - Clases externas descomposiciones no exhaustivas y particiones
  - Instancias comunes y descomposiciones disjuntas y particiones

- Instancias externas descomposiciones no exhaustivas y particiones
  - Errores semánticos
2. Estado incompleto
- Clasificación de conceptos incompleta
  - Errores de particiones
    - Conocimiento disjunto omitido
    - Conocimiento exhaustivo omitido
3. Redundancia
- Gramática
    - Redundancia en relaciones de subclases
    - Redundancia en relaciones de instancias
  - Misma definición de clases
  - Misma definición de instancias

Conociendo y evitando estos problemas, los usuarios pueden diseñar y mantener ontologías de calidad, es decir, aquellas que sean:

Significativas — Todas las clases deben poder ser instanciadas.

Válidas — Capturan la intuición de los expertos en el dominio.

Minimamente redundantes — Sin sinónimos no intencionales.

Axiomatización detallada — Descripciones detalladas.

Actualmente se dispone de una serie de herramientas para ayudar a los usuarios a lograr este objetivo. Una de ellas es **ODEval**, que evalúa la ontología desde el punto de vista de la representación del conocimiento, es decir, desde el punto de vista de las

relaciones que especifican la taxonomía de la ontología (las relaciones entre sus clases y atributos) a través de los conceptos taxonómicos de RDF(s). De esta forma, ayuda a los desarrolladores de ontologías a diseñarlas sin anomalías. Existen otras dos herramientas que permiten evaluar las ontologías: **OntoEdit** y **ODEClean** (basada en la metodología OntoClean). En la Tabla 1 se muestra una tabla descriptiva de las herramientas existentes y anteriormente comentadas. [Hart et al., 05]

Herramienta	Descripción
<b>OntoClean</b>	Se aplica a lo largo de todo el ciclo de vida de la ontología. Detecta inconsistencias formales y semánticas en las propiedades de la ontología. Utiliza las nociones de esencia y rigidez.
<b>OntoMetric</b>	Determina la idoneidad de la ontología para un tema específico. Evalúa la ontología estudiando varias de sus características y sus objetivos, para ver si encaja en un determinado proyecto.
<b>OntoEdit</b>	Evalúa las relaciones taxonómicas. Utiliza las nociones de esencia y rigidez.
<b>ODEval</b>	Evalúa la taxonomía de los conceptos, detecta inconsistencias, redundancias y ciclos. Soporta ontologías RDF(S), DAML+OIL y OWL.
<b>OntoManager</b>	Ayuda a determinar la veracidad y la aplicabilidad de una ontología en un dominio específico. Modifica la ontología de acuerdo a los criterios del usuario.

**Tabla 1 Descripción de herramientas para la evaluación de ontologías**

Sin embargo, hasta ahora no se había desarrollado una utilidad software que permitiera la evaluación de la ontología con respecto a sus requisitos particulares, concretamente con respecto a una serie de pruebas que fueran diseñadas estableciendo, para cada una de ellas, de qué afirmaciones se parte, qué consultas se realizan y qué respuestas se esperan.

Precisamente ésta es la laguna que cubre el proyecto final de carrera descrito en la presente memoria.

Hoy en día no existe ninguna herramienta que permita esto tal y como se ha visto. OntologyTest le permite al usuario crear conjuntos de tests de forma sencilla (a

través de una interfaz gráfica muy intuitiva) sobre las *ABox* e inferir conocimiento para así poder probar si la ontología que se ha realizado cumple con sus especificaciones.

Para ello, OntologyTest evalúa la ontología con la ayuda de un razonador que soporta OWL-DL. Para lograr éste objetivo, ha sido necesario hacer un estudio sobre los razonadores existentes para determinar cuál sería el más apropiado par llevar a cabo esta tarea. Dicho estudio aparece en el punto Selección del razonador

## 3 ESPECIFICACIÓN DE REQUISITOS

En este apartado se presentan los requisitos funcionales que deberán ser satisfechos por el sistema y que han sido el resultado final de todo el proceso incremental de desarrollo del sistema, como resultado de la interacción con el experto en ontologías. Todos los requisitos aquí expuestos son esenciales, es decir, no sería aceptable un sistema que no satisficiera alguno de los requisitos aquí presentados.

### 3.1 *Requisitos funcionales*

Los requisitos funcionales para el sistema que se va a desarrollar son los siguientes:

#### 3.1.1 *Añadir ontología*

El sistema será capaz de cargar cualquier ontología indicada por el usuario siempre que esta:

1. Esté escrita en lenguaje OWL;
2. Sea consistente;
3. Esté almacenada de forma local en el ordenador del usuario.

Durante el proceso de carga el sistema verificará si se cumplen estas condiciones. En caso de que se cumplan el usuario podrá proceder a utilizar la herramienta; en caso contrario se le indicará que ocurre un error.

#### 3.1.2 *Crear Proyecto*

El sistema permitirá crear proyectos. Un proyecto es un archivo XML que contiene la sesión de trabajo realizada con la herramienta y guardada por el usuario. El proyecto estará formado por los tests y las instancias que haya realizado el usuario, así como por una referencia a la ontología sobre la que se ha trabajado y su namespace.

### 3.1.3 Crear test

Permite comprobar si, mediante la ontología, se responde correctamente a una consulta.

El usuario proporcionará: el nombre del test, su descripción en lenguaje natural (opcional), una consulta y el resultado esperado. También puede proporcionar en el momento de crear el test el conjunto de instancias que asociará al mismo (véase requisito 3.1.11 Asociar instancias a test).

La Tabla 2 muestra los tipos de test con sus tipos de consultas y sus tipos de resultados esperados correspondientes.

Tipo de Test	Tipo de resultado	Tipo de Consulta	Descripción
Test de instanciación	True – si se espera que el individuo pertenezca a la clase  False – en caso contrario	Clase(individuo)	Se pasa el test si y sólo si el individuo pertenece a la clase.
Test de recuperación	Lista esperada de individuos que pertenecen a la clase	Clase	Se pasa el test si y sólo si todo individuo de la lista pertenece a la clase.
Test de realización	Clase esperada más específica a la que pertenece el individuo.	Individuo	Se pasa el test si y sólo si la clase esperada es la más específica a la que pertenece el individuo.
Test de satisfactibilidad	True – si se espera poder añadir la instancia a la clase sin perder la consistencia.  False – en caso contrario	Clase(individuo)	Se pasa el test si y sólo bien si se espera que se puedan añadir instancias manteniendo la consistencia y realmente se puede, bien no se espera, y realmente no se puede.
Test de clasificación	Lista de clases esperadas a las que pertenece el individuo.	Individuo	Se pasa el test si y sólo si toda clase de la lista es instanciada por el individuo.
Test SPARQL	Resultado esperado de la consulta.	Consulta SPARQL	Se pasa el test si y sólo si el resultado de la ejecución de la consulta Sparql coincide con el



			introducido en resultado esperado.
--	--	--	------------------------------------

**Tabla 2 Tipos de Tests**

#### **3.1.4 Editar test**

El sistema permitirá que el usuario pueda, en cualquier momento, editar los test que haya realizado.

#### **3.1.5 Visualizar test**

El sistema le permitirá al usuario ver un informe completo del test que seleccione, indicándole en éste:

- Tipo de test
- Nombre del test
- Descripción
- Instancias asociadas al mismo
- Consultas realizadas

#### **3.1.6 Importar test**

El sistema permitirá que el usuario importe tests desde otros proyectos (realizados previamente con la misma herramienta) a su proyecto de trabajo actual.

#### **3.1.7 Crear instancias**

El sistema permitirá al usuario crear conjuntos de instancias (de clase y de propiedad). Con estos conjuntos podrá realizar las siguientes operaciones:

- Añadirlos a la ontología
- Asociarlos a un test
- Guardarlos

Las instancias introducidas deberán ser conformes a la ontología a la que se quieran asociar.

### *3.1.8 Editar instancias*

El sistema permitirá que el usuario pueda, en cualquier momento, editar las instancias que haya realizado.

### *3.1.9 Importar instancias*

El sistema permitirá que el usuario importe instancias desde otros proyectos (realizados previamente con la misma herramienta) a su proyecto de trabajo actual.

### *3.1.10 Visualizar instancias*

El sistema le permitirá al usuario ver un informe completo de las instancias que seleccione, indicándole en éste:

- Nombre del conjunto de instancias
- Descripción
- Instancias de clase
- Instancias de propiedad

### *3.1.11 Asociar instancias a test*

El sistema desarrollado le permitirá al usuario asociar conjuntos de instancias a sus tests de distintas formas:

- Desde la propia creación de los tests.
- Desde un archivo de instancias indicándole a que test las quiere asociar.

### *3.1.12 Ejecutar test*

El usuario podrá en cualquier momento elegir entre ejecutar un test, una selección de tests o todos los test que tiene creados hasta el momento.

### *3.1.13 Mostrar resultados de la ejecución del test*

El sistema, tras cada ejecución de los tests, le mostrará al usuario su resultado, indicando para cada test:

- Si ha fallado o ha pasado
- Las consultas que han sido correctas (indicando para ello el resultado esperado y el obtenido)
- Las consultas que han fallado (indicando para ello el resultado esperado y el obtenido)

#### *3.1.14 Guardar Proyecto*

El sistema le permitirá al usuario guardar el proyecto que ha realizado en cualquier momento, bien en el directorio de trabajo, o en el directorio que el usuario seleccione. El proyecto será el conjunto de tests y de instancias que el usuario haya creado así como una referencia a la ontología sobre al que ha trabajado. El proyecto será guardado en un documento XML.

#### *3.1.15 Abrir Proyecto Existente*

El usuario podrá abrir un proyecto existente para trabajar con él. Éste proyecto deberá haber sido creado con esta herramienta y tener extensión .XML.

#### *3.1.16 Seleccionar Idioma*

El usuario podrá seleccionar el idioma con el que desea trabajar con la aplicación, pudiendo elegir entre español, inglés (uk) o inglés (us).

#### *3.1.17 Ver Ontología*

El usuario podrá, desde la aplicación, ver la ontología con la que está trabajando.

#### *3.1.18 Cerrar Proyecto*

El usuario podrá cerrar el proyecto con el que está trabajando.

### 3.1.19 Eliminar Proyecto

El usuario podrá eliminar el proyecto con el que está trabajando. Esto implicará eliminar tanto el archivo .xml que especifica el proyecto como la carpeta que lo contiene.

## 3.2 Requisitos de interfaces externas

### 3.2.1 Interfaces de usuario


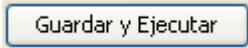


Al tener el software de OntologyTest que ofrecer interfaces tanto al usuario final como al usuario que desee modificar el razonador o el *framework* utilizado para trabajar con la API de OWL, se distinguirá a quién va dirigida cada interfaz. Si el lector del documento está interesado en la herramienta desde el punto de vista del usuario final deberá limitarse sólo a las secciones que definen las interfaces con este tipo de usuario.

#### *Interfaces de usuario final*

##### *Generación de Tests Simples*




Mediante el menú de generación de tests simples será posible crear cinco tipos de tests, cada uno de los cuales ofrecerá información sobre consultas concretas a la ontología.

Todos los tests tendrán como campos comunes, tal y como se ve en la Figura 2, el nombre y la descripción para el test, así como las distintas operaciones que se pueden realizar sobre ellos, que son:

- Guardar (identificado mediante el botón )
- Guardar y ejecutar (identificado mediante el botón )
- Ejecutar (identificado mediante el botón )
- Asociar instancias (identificado mediante el botón )

**Figura 2 Detalle de la cabecera de la creación de tests**

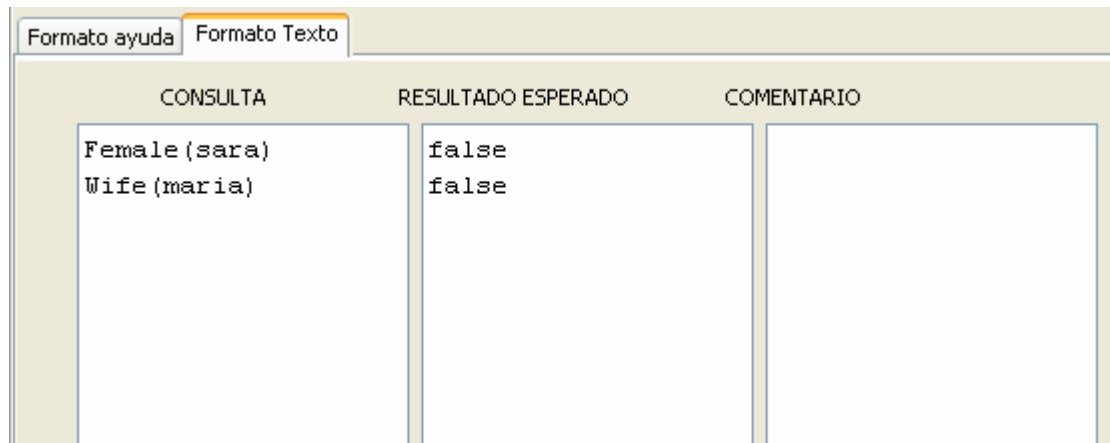
El usuario tendrá dos formas posibles para realizar sus consultas. Ambas formas aparecen identificadas por pestañas independientes dentro de la pantalla de creación del test, identificadas como: “Formato Ayuda” y “Formato Texto”.

El primero muestra una interfaz más guiada para la creación de consultas de forma individual, pudiéndose borrar cada consulta mediante el botón identificado con el símbolo  o duplicarla mediante el botón , además de agregar cualquier comentario que se desee a esa consulta pinchando en el botón .

En la Figura 3 se muestra la interfaz gráfica para la creación de tests mediante el modo ayuda:

**Figura 3 Detalle de la interfaz para la creación de tests con Formato Ayuda**

El segundo muestra una interfaz de texto planto para las consultas, resultados esperados y comentarios, facilitando así el hecho de poder copiar y pegar fácilmente los datos entre los tests, tal y como se ve en la Figura 4







CONSULTA	RESULTADO ESPERADO	COMENTARIO
Female(sara)	false	
Wife(maria)	false	

**Figura 4 Detalle de la creación de tests mediante el Formato Texto**

El usuario final podrá especificar para cada test un nombre identificativo (distinto para cada test), una breve descripción, las consultas/respuestas esperadas que desee asociar al test así como los conjuntos de instancias (Ver sección “Carga de Instancias”).

#### *Generación de Tests SPARQL*

Mediante el menú de generación de tests SPARQL será posible crear un test con tantas consultas SPARQL como se desee, junto con sus correspondientes resultados esperados. Se podrá navegar mediante los botones   sobre las consultas ya creadas así como eliminarlas con el botón  o limpiar los campos mediante el botón .

El usuario final podrá especificar para cada test un nombre identificativo (distinto para cada test), una breve descripción, las consultas SPARQL/respuestas esperadas que desee asociar al test así como los conjuntos de instancias, tal y como ocurría para los tests simples.

En la Figura 5 se puede ver un ejemplo de un tests Sparql.

Introduzca el nombre del test:

Test SPARQL

Añada una descripción para el test:

Probando los tests de consultas sparql

Introduzca la consulta en SPARQL:

```

PREFIX
rdfs:<http://www.owl-ontologies.com/famil
y.owl>
SELECT ?subject ?object
WHERE {?subject rdfs:subClassOf ?object }
    
```

Resultado esperado:

```

subject (female,male)
object (wife)
    
```

Guardar y Ejecutar

+ Instancias

**Figura 5 Detalle de un Test Sparql**

### *Generación de Instancias*

Mediante el menú de Instancias será posible crear tanto instancias de clase como de propiedad. Para cada conjunto de instancias se especificará un nombre identificativo y una descripción, tal y como se ve en la Figura 6

Nombre para el conjunto de instancias:

Descripción:

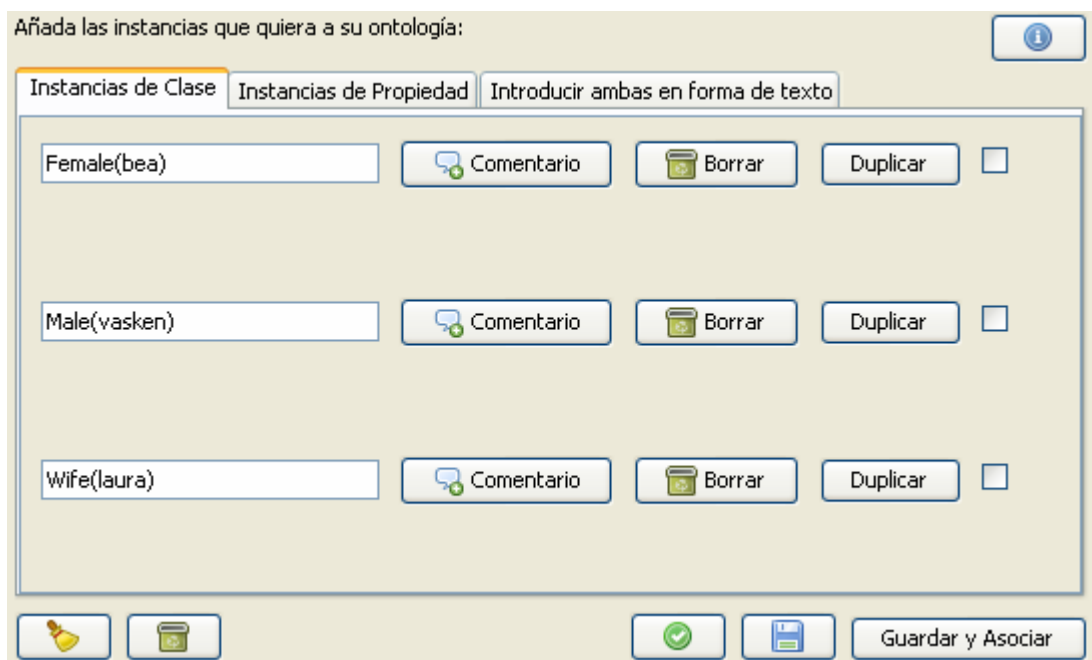
**Figura 6 Detalle de la cabecera de las Instancias**

Para diferenciar las instancias de clase y propiedad estas aparecen especificadas en pestañas de navegación con el nombre de “Instancias de Clase” e




“Instancias de Propiedad” sobre las que el usuario podrá moverse para crear en cada momento las que necesite.

El usuario tendrá dos formas posibles para crear las instancias. Ambas formas aparecen en un submenú dentro del menú de principal de generación de instancias indicadas con los nombres: “Formato Ayuda” y “Formato Texto”.

En la Figura 7 se ve un ejemplo del proceso de creación de instancias de clase.

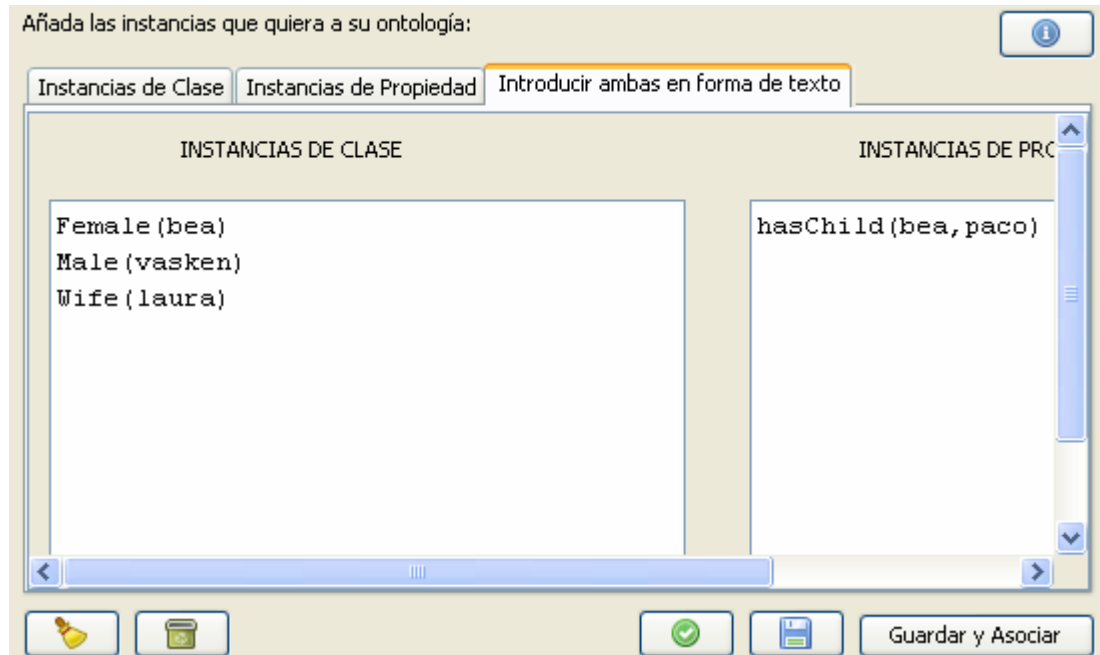


**Figura 7 Detalle de la creación de instancias mediante la Opción Ayuda**

Las pestañas “Instancias de Clase” e “Instancias de Propiedad” (ver Figura 8) muestran una interfaz sencilla para la creación de instancias de forma individual, pudiéndose borrar cada instancia mediante el botón identificado con el símbolo  o duplicarla mediante el botón , además de agregar cualquier comentario que se desee a esa instancia pinchando en el botón .


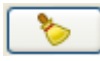



La pestaña “Introducir ambas en forma de texto” muestra una interfaz de texto plano para las instancias de clase y propiedad, facilitando así el hecho de poder copiar y pegar fácilmente diferentes grupos de instancias entre los tests. Un ejemplo para este tipo se ve en la Figura 8.





**Figura 8** Detalle de la creación de instancias mediante el Formato Ayuda

Sobre cada conjunto de instancias será posible realizar distintas acciones:

- Eliminar las instancias seleccionadas (mediante el botón )
- Borrar las instancias seleccionadas (mediante el botón )
- Guardar el conjunto de instancias (mediante el botón )
- Asociarlo a un test (mediante el botón )
- Guardar y asociarlo (mediante el botón )

### *Carga de Tests Simples, SPARQL e Instancias*

Desde el menú principal “Tests” o “Instancias” se podrá acceder a la opción de importar. Una representación gráfica de esta acción se ve en la Figura 9.

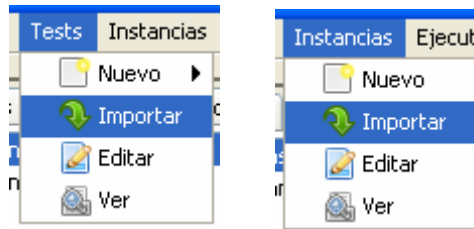


Figura 9 Detalle de la opción Importar Tests e Importar Instancias

Con esta opción se accede a la pantalla representada en la Figura 10 que pedirá que se seleccione el proyecto desde el cual se quieren importar los tests o las instancias. Dicho proyecto ha tenido que ser creado y guardado anteriormente con esta herramienta. Una vez seleccionado un proyecto válido, aparecerán los tests o las instancias que contiene, con un breve resumen a la derecha. Se podrá entonces ver el test o la instancia completa o importarlo.

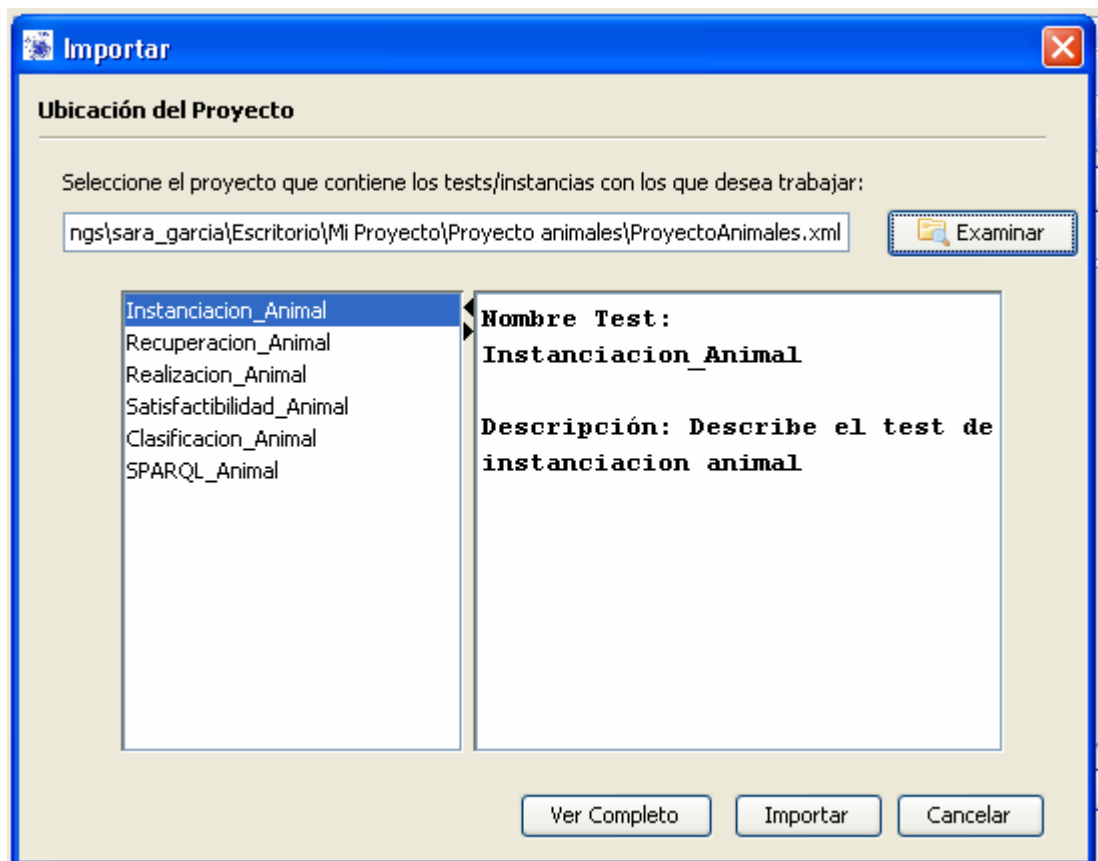

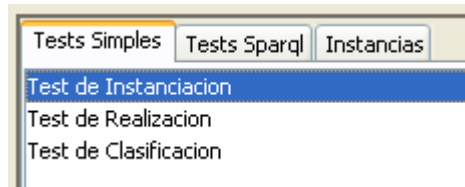


Figura 10 Detalle del paso intermedio para importar tests

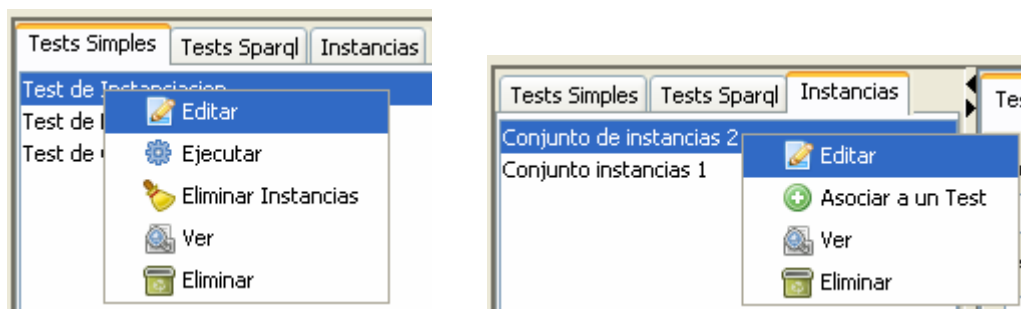
### Almacenamiento de Tests Simples, Sparql e Instancias

Tanto los tests como las instancias podrán almacenarse durante la sesión de trabajo mediante el botón . Se irán añadiendo en forma de lista a la parte derecha y contenedora de los mismos de la aplicación, tal y como muestra la Figura 11.



**Figura 11** Detalle de la lista de Tests Simples

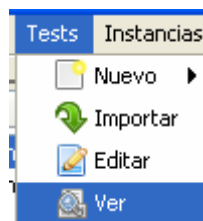
Una vez guardados, si se hace clic derecho sobre alguno de los elementos de la lista, se accederá a un menú rápido con varias opciones y distintas en el caso de tests e instancias, que pueden verse en la Figura 12.



**Figura 12** Detalle del submenú para los tests y las instancias

### Visualización de Tests Simples, SPARQL e Instancias

Mediante la opción del menú principal de Tests e Instancias se puede acceder a la opción “Ver Tests” o “Ver instancias” como muestra la Figura 13.



**Figura 13** Detalle del menú de Tests con la opción de Ver

Al pinchar sobre esta opción, aparecerá una pantalla con todos los tests o instancias (dependiendo de si hemos accedido desde el menú Tests o desde el menú Instancias) guardados en la sesión de trabajo para que se seleccione el test/instancia que se desea ver, tal y como muestra la Figura 14.

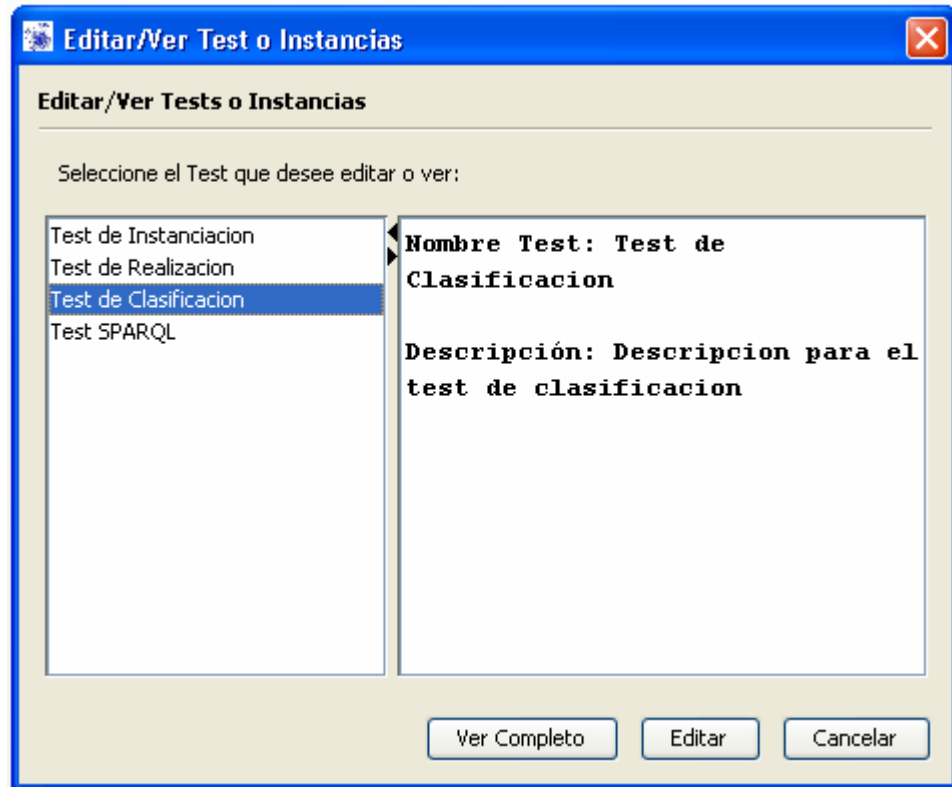
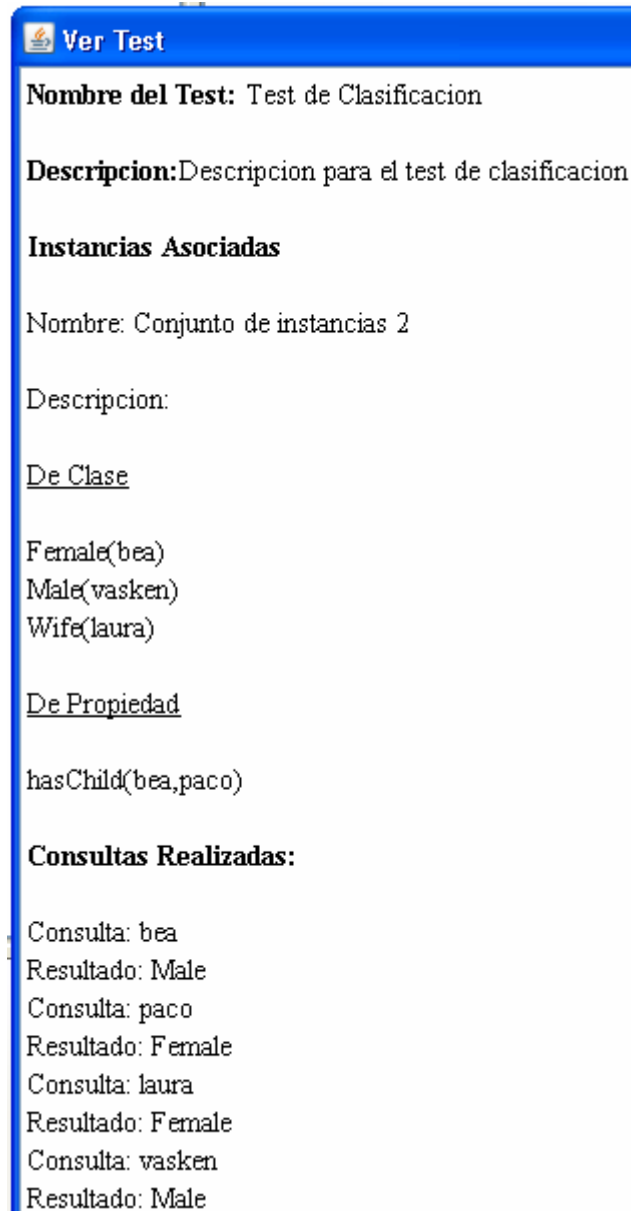


Figura 14 Detalle del paso intermedio para ver un test

La vista será un texto describiendo el test o las instancias, tal y como se ve en la Figura 15.



**Figura 15** Detalle de la vista de un test

### *Ejecución de Tests Simples y Sparql*

Desde el menú principal existe la opción de ejecutar todos los tests guardados en la sesión de trabajo (Figura 16), o de seleccionar cuáles ejecutar (Figura 17). También existe la posibilidad de ejecutar los tests desde su proceso de creación.



Figura 16 Detalle del menú con la opción de Ejecutar tests

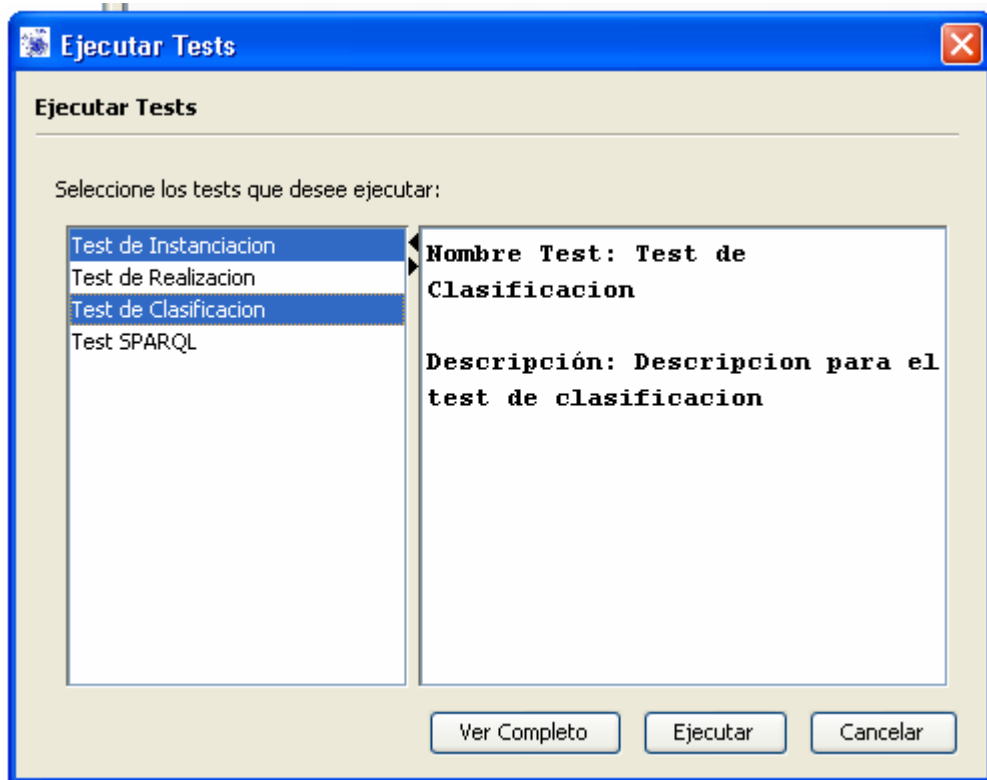


Figura 17 Detalle de la selección de test para su ejecución

Como se ve en la Figura 17, el usuario será el encargado de seleccionar los tests que desee ejecutar. Para facilitar la elección del test se le proporciona una breve descripción. Existe la opción de ver el test completo.

El resultado de la ejecución de los tests se mostrará indicando en color rojo los test que han fallado, y en color verde los que han sido correctos. Para cada test, se especificará cuáles han sido las consultas que han fallado (indicando qué resultado se esperaba obtener y cuál se ha obtenido) y cuáles las que han sido correctas, tal y como puede verse en la Figura 18.

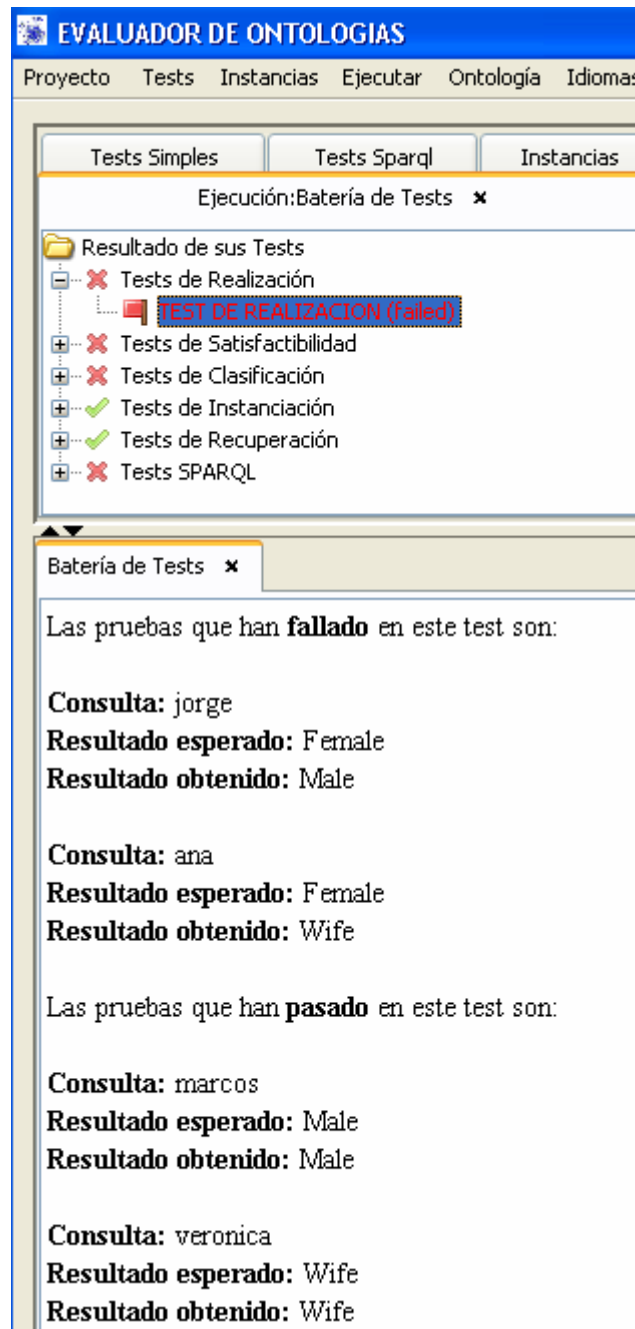


Figura 18 Detalle de la vista de la ejecución de los tests

### *Interfaces de Usuario*

#### *Añadir framework de desarrollo sobre Java para OWL (usuario)*

Este apartado está indicado para un usuario experto que desee modificar parte de la implementación del sistema para adaptarlo a sus necesidades, pudiendo seleccionar,

por ejemplo, el razonador que desea utilizar para la evaluación de la ontología (por defecto el razonador seleccionado es Pellet).

Para ello, el usuario deberá implementar una interfaz que defina las siguientes operaciones:

- Añadir instancias de clase a la ontología.
- Añadir instancias de propiedad a la ontología.
- Añadir un razonador a la ontología.
- Borrar todas las instancias asociadas a la ontología.
- Añadir tantos tests como se deseen a la ontología.

### 3.2.2 Atributos de software

#### ***Extensión***

Se pretende que esta herramienta sea integrada en el futuro con otras en el proceso de evaluación de ontologías. Los requerimientos actuales de OntologyTest se han desarrollado en base a algunas de las necesidades que hoy en día se plantean y aún están sin resolver sobre la evaluación de ontologías, tales como el hecho de tener una herramienta para probarlas de igual forma que hoy en día se puede probar el software a través de una herramienta como JUnit. Por lo tanto, es posible que en el futuro aparezcan nuevas necesidades o que se quieran resolver las aquí no contempladas, por lo que existirá la necesidad de extender el código y/o utilizar otros *frameworks*. Llegado ese punto, la facilidad de extensión del desarrollo de OntologyTest será clave.

Como consecuencia, el desarrollo debe ser modular. Existirá una gran cohesión dentro de cada módulo, minimizándose el acoplamiento entre ellos. Deberá documentarse la funcionalidad de cada módulo para facilitar su sustitución o modificación.



### ***Legibilidad del código***

Como cualquier desarrollo Java, empleará los convenios de notación típicos del lenguaje y el código fuente estará implementado convenientemente para facilitar su legibilidad.

## **3.3 Requisitos de rendimiento**

### ***3.3.1 Capacidad de respuesta de la interfaz de usuario al ejecutar tests***

Al lanzar la ejecución de los tests es posible que su tiempo de ejecución sea considerable en función del tamaño de la ontología a evaluar. Esto es debido a que la herramienta valida la ontología introducida a través de un razonador, por lo que si ésta tiene un tamaño elevado (a partir de 1Mb) el proceso se puede tardar entre uno o cinco minutos. Para evitar que la interfaz de usuario deje de responder a las acciones del usuario, las funciones que ejecutan dichos tests deberán lanzarse en un hilo de ejecución independiente del de la interfaz de usuario. En todo momento se permitirá que el usuario detenga la ejecución de los tests mediante un botón “Cancelar”; esto es, termine el hilo de ejecución de la función que lo realiza y cualquier hilo de ejecución que se haya lanzado desde éste.

También se ejecutarán en hilos independientes la carga de la ontología a la hora de abrir un proyecto existente o de crear uno nuevo, ya que si la ontología es extensa el proceso puede alargarse.

## **3.4 Requisitos tecnológicos**

El sistema deberá satisfacer los siguientes requisitos tecnológicos:

- 1) La ontología a evaluar debe estar almacenada en local por el usuario.
- 2) OntologyTest deberá funcionar, al menos, en:

- Un procesador Intel Pentium de 1,73 GHz, a 267 MHz y 104 Gb de memoria RAM.

- El sistema operativo Windows XP.

3) El sistema estará implementado en Java. Se podrán utilizar otros lenguajes en las interfaces externas.

## **4 Metodología**

La oferta de metodologías en el campo de la ingeniería del software (ya de por sí bastante amplia) se ha visto incrementada de forma notable en estos últimos años. Algunas de estas metodologías son: extreme Programming (XP) [Beck04], Scrum [Schwaber04], Crystal [Cockburn04], Feature Driven Development (FDD) [Palmer02], Rational Unified Process (RUP) [Kroll03], Dynamic Systems Development Method (DSDM) [DSDM03]. Cada una de ellas puede optimizar el proceso de desarrollo si se utiliza para resolver el problema para el cual fue diseñada. Sin embargo, si se aplican fuera de un entorno adecuado pueden arruinar el proyecto.

Entre los diferentes factores que influyen en las características de la metodología destacamos: la limitación temporal, los niveles de calidad tolerables, la formación y experiencia de los integrantes (tanto en tecnología como en metodologías), las dimensiones del equipo de trabajo, el impacto de un posible retraso en la fecha de entrega, la disponibilidad del experto, la cantidad de documentación que el equipo puede generar y manipular adecuadamente, etc. En la práctica, no debe emplearse una metodología de desarrollo software ya existente sin más, ya que las particularidades de cada proyecto lo hacen diferente de todos los demás. Lo oportuno es tener en cuenta todos los factores influyentes y, tomando ideas de la extensa bibliografía sobre metodologías existente, construir una propia que se adapte de una forma adecuada al proyecto.

### **4.1 Características del proyecto**

Se deben resaltar dos aspectos a la hora de hablar de las características del proyecto: la morfología del equipo de desarrollo y las propiedades de la solución que se desea implementar.

En el equipo de desarrollo se cuenta con dos expertos con una amplia experiencia: uno en el campo de las ontologías y otro en el de la plataforma Java y las tecnologías de representación gráficas que se emplean (*Swing*) [Elliott02]. La elección de *Swing* como herramienta para realizar la interfaz gráfica, se ha debido principalmente a que *Swing* es la librería gráfica estándar de desarrollo dentro de Java,

porque es completamente multiplataforma, y porque tiene un soporte muy bueno en los entornos del desarrollo, especialmente en *Netbeans*.

Gracias al conocimiento de los problemas que debe solucionar este software y de las posibilidades y limitaciones que presenta la plataforma con la que se desarrolla, es posible acotar en cierta medida el alcance del proyecto, si bien, a lo largo de su desarrollo, han ido surgiendo nuevas ideas en cuanto a funcionalidad se refiere. Basándose en su experiencia en el desarrollo y evaluación de ontologías, uno de nuestros expertos está en disposición de realizar una especificación de requisitos previos que no resulte muy distante de la especificación final que se deberá implementar. La característica fundamental de la solución que debemos desarrollar es la de ser independiente de un *framework* y escalable. Por tanto es necesaria la apertura a cambios en cualquier punto del proceso de desarrollo. Esto está cercano a una metodología ágil, es decir, integración paulatina de nuevas funcionalidades a la herramienta. Esto permite detectar problemas rápidamente, reduciendo el coste de modificación de los requisitos y del modelo implementado.

## **4.2 Metodología utilizada**

Una pieza clave en las metodologías ágiles es la de revisar el desarrollo de la aplicación con un periodo de tiempo corto por parte de los expertos. Al formar éstos parte del equipo de desarrollo esto no será un problema. Para facilitar el acceso al código fuente hemos utilizado un sistema de control de versiones [code.google.com]. Esto, unido a una amplia disponibilidad a la hora de realizar reuniones, ha permitido ir puliendo el software definido según ha ido avanzando en su desarrollo.

Por *code review* [Cohen06] se entiende el proceso por el cual un desarrollador muestra el código creado a otros desarrolladores de igual nivel o superior para que lo revisen. Permite a los programadores compartir su experiencia, mejorando la habilidad global de todos ellos. Existen diversas técnicas para llevar a cabo este *code review*:

- Revisión periódica en la cual un supervisor analiza el código desarrollado por su supervisado. Es un miembro externo al equipo de desarrollo el que analiza el código. Éste ha sido el caso de éste proyecto.

- Revisión del código por parte de otro miembro del equipo con al menos la misma experiencia que el desarrollador, preferiblemente esta experiencia será mayor. A diferencia del punto anterior, es un miembro del equipo de desarrollo quién analiza el código.
- Revisión del código por parte del equipo de desarrollo al completo, analizando y corrigiéndolo en público. Varios miembros del equipo revisan el código en este caso.

Gracias al avanzado conocimiento en ingeniería del software de nuestros expertos y en concreto de uno de ellos en la plataforma Java, nos ha sido posible utilizar un *code review* continuo utilizando la técnica de revisiones efectuadas por un desarrollador de mayor experiencia que el equipo de desarrollo.

Por último, nada de lo anterior sería realmente útil si no se prueba el código. Paralelamente a la construcción del software y tras cada *code review* se ha ido probando el software ya funcionando, pudiendo así detectar errores.

Uniando las técnicas de *code review* y desarrollo de soluciones reales desde fases tempranas, conseguiremos aumentar la calidad del código.

## 5 Análisis de la tecnología disponible

Al contar en este proyecto, como miembro del equipo de desarrollo, con un experto tanto en el problema que deseamos solucionar, como en las plataformas y tecnologías que hemos empleado, ha originado que la fase análisis se haya minimizado. Desde un principio, este experto proporcionó de una manera concreta tanto el alcance del proyecto como las plataformas que se debían utilizar en su desarrollo.

Se ha considerado como buena documentación el desarrollo de diagramas de clase, tanto para la comunicación interna entre experto y desarrolladores del software, como para los futuros usuarios a la hora de crear sus extensiones al producto, por lo que han sido desarrollados e incluidos en la documentación.

A continuación se justifican las decisiones más relevantes relativas a las tecnologías que se han usado.

### 5.1 Plataforma de desarrollo

La plataforma de desarrollo utilizada ha sido *Netbeans* 6.1 [Netbeans.org] y el lenguaje de desarrollo Java. Las ventajas de utilizar este lenguaje frente a otros, son las siguientes:

- **Multiplataforma.** El desarrollo del *software* a desarrollar está orientado desde un principio a su posterior uso en el campo de las ontologías. En este entorno, el *hardware* utilizado puede ser heterogéneo. Java proporciona la capacidad de que el *software* sea funcional en cualquier sistema que disponga de una máquina virtual de Java de manera directa.
- **Librerías.** Java cuenta con una gran cantidad de librerías existentes, muchas de ellas *open source*, desarrolladas por terceras partes. Se han utilizado las librerías de *Jena* y *Pellet* necesarias para el desarrollo de la aplicación.
- **Documentación.** Un grupo importante de usuarios a los cuales está destinado el *software* es aquel en el que sus miembros no tienen grandes conocimientos de programación. Es importante que estos usuarios tengan una base de datos de

documentación amplia que resuelva sus dudas. El uso de Java proporciona esta documentación de manera gratuita desde la propia desarrolladora de la plataforma (*Sun Microsystems*) [sun.com c]. Proporciona tanto una explicación detallada de las *APIs* de Java (*Javadoc*) [sun.com d] como explicaciones y ejemplos de cómo utilizarlas correctamente [sun.com e]. Adicionalmente, la mayoría de entornos de desarrollo de Java, permiten generar de manera automática la documentación del código que se desarrolle (*Javadoc*). Esto facilitará la tarea de documentar los tests desarrollados para nuestra herramienta.

Una decisión que se tomó fue la de establecer qué versión de Java debería instalar el usuario. J2SE 1.5.0 proporcionaba en un principio toda la funcionalidad necesaria. Durante el desarrollo del proyecto se lanzó la versión Java SE 6. Esta nueva versión proporciona dos herramientas que facilitan ciertas tareas del desarrollo:

- ***Layouts***. Se incorporaron nuevos *layouts* como clases nativas de Java. Estos *layouts* son utilizados por el editor gráfico de la plataforma de desarrollo *Netbeans* (*Matisse*) [Netbeans.org]. El uso de este editor facilitará, principalmente, el diseño de pantallas de configuración de propiedades. En J2SE 1.5.0 estos *layouts* se incorporaban como una librería externa.
- ***Swingworker***. Al desarrollar aplicaciones con interfaz gráfica un problema común es el de ejecutar en el mismo hilo de ejecución de la interfaz (*EDT* en el caso de *Swing*) tareas pesadas; esto implica que la interfaz gráfica no responda correctamente durante ese periodo [O’Cooner07]. En el caso de tareas que pueden consumir una gran cantidad de tiempo (carga de archivos, ejecución de tests, etc.) es necesario crearlas en hilos diferentes. *Swingworker* es una utilidad integrada en Java SE 6 que facilita enormemente la gestión de estos hilos de ejecución.

Por estos dos motivos, principalmente por el uso de *Swingworker*, se considerará razonable el hecho de forzar al usuario a disponer de la versión Java SE 6 o superiores para poder ejecutar la herramienta.

## **5.2 Tecnología para el desarrollo de la interfaz gráfica**

La tecnología utilizada para el desarrollo de la interfaz gráfica de usuario en Java es *Swing*. Esta elección se debe a que *Swing* es la librería gráfica estándar de desarrollo dentro de Java, ya que es completamente multiplataforma y tiene un soporte muy bueno en los entornos de desarrollo, especialmente en *Netbeans*.

## **5.3 Selección del razonador**

Un razonador es un componente software que permite operar lógicamente sobre una base de conocimiento, por ejemplo para verificar su consistencia o inferir nueva información.

Para poder realizar un software que permita evaluar ontologías y ejecutar los tests, es necesario que éste se apoye en un razonador que le permita verificar si la ontología a evaluar es válida (no es contradictoria) y está verificada (no tiene inconsistencias internas). Según lo expuesto, antes de evaluar una ontología, comprobaremos con un razonador si ésta es válida y está verificada. De no ser así, la validación no será posible.

Existen varios razonadores OWL. Algunos son: CEL (<http://lat.inf.tu-dresden.de/systems/cel>), CEREBRA ENGINE (<http://www.webmethods.com>), FACT++ (<http://owl.man.ac.uk/factplusplus>), fuzzyDL (<http://gaia.isti.cnr.it/~straccia/software/fuzzyDL/fuzzyDL.html>), el razonador de KAON2 (<http://kaon2.semanticweb.org>), MSPASS (<http://www.cs.man.ac.uk/~schmidt/mspass>), PELLET (<http://pellet.owldl.com>), QUONTO (<http://www.dis.uniroma1.it/~quonto>), RACER PRO (<http://www.racer-systems.com/>).

Sin embargo, para realizar la elección sobre cuál de ellos se desarrollará el proyecto, en primer lugar se especifican una serie de condiciones que el razonador debe de cumplir:

- 1.- Debe ser un razonador no comercial, es decir, con licencia libre para su uso.
- 2.- Debe poder ser utilizado en un sistema operativo Windows.



3.- Tiene que poder ser accedido desde Java con facilidad.

4.- Debe ser un razonador para OWL DL como mínimo.

A continuación y para cada razonador, se indicará qué condiciones de las anteriormente expuestas satisface cada uno, como se ve en la Tabla 3.

**CEL** es un razonador desarrollado exclusivamente para Linux, por lo que queda descartado. **CEREBRA ENGINE** no satisface las condiciones preestablecidas porque es un razonador que requiere licencia de pago. **FACT++** también queda descartado, pues es un razonador implementado en C++ y no es fácilmente accesible desde Java. **FuzzyDL** es un razonador que no soporta OWL DL. **MSPASS** está implementado en C, por lo que tampoco satisface los criterios. **QUONTO** no soporta OWL DL por lo que queda descartado. **RACER PRO** es un razonador comercial, en consecuencia, tampoco se puede utilizar en el presente proyecto. Los únicos razonadores que satisfacen todas las condiciones establecidas a priori son **KAON2** y **PELLET**, que se describirán en los siguientes apartados.

En la Tabla 3 Razonadores versus Criterios de Selección se muestra una tabla donde se especifica las condiciones identificadas con anterioridad que cumple cada razonador, y cuales no.

	<b>No comercial</b>	<b>Windows</b>	<b>Java</b>	<b>Owl DL</b>
CEREBRA ENGINE	No satisface	Satisface	No satisface	Satisface
FACT++	Satisface	Satisface	No satisface	Satisface
FuzzyDL	Satisface	Satisface	No satisface	No satisface
<b>KAON 2</b>	<b>Satisface</b>	<b>Satisface</b>	<b>Satisface</b>	<b>Satisface</b>
MSPASS	Satisface	Satisface	No satisface	Satisface
<b>PELLET</b>	<b>Satisface</b>	<b>Satisface</b>	<b>Satisface</b>	<b>Satisface</b>
QUONTO	Satisface	Satisface	Satisface	No satisface
RACER PRO	No satisface	Satisface	Satisface	Satisface
CEL	Satisface	No satisface	Satisface	Satisface

Tabla 3 Razonadores versus Criterios de Selección

### 5.3.1 KAON2 [kaon2]

Es un entorno de desarrollo de ontologías basado en Java que permite trabajar con ontologías de tipo OWL DL, SWRL y F-Logic (*Frame Logic*). Es gratuito para uso no comercial. Proporciona:

1. Una API para manipular ontologías de tipo OWL DL, SWRL y F-Logic.
2. Una interfaz para resolver preguntas utilizando la sintaxis SPARQL.
3. Una interfaz DIG (*DL Implementation Group*) que permite el acceso de KAON2 desde otras herramientas, por ejemplo, Protegé.
4. Un módulo que permite extraer instancias de la ontología desde una base de datos relacional.

Las cuestiones abiertas que tiene hoy en día KAON2 son:

1. No soporta nominales. Por ejemplo, si una ontología contiene una clase de tipo owl:oneOf o una restricción de tipo owl:hasValue, el razonador obtendrá como resultado un error.
2. Se han observado problemas hasta en ontologías que contienen como máximo una cardinalidad de dos. En este tipo de ontologías no es posible responder ninguna pregunta.

### 5.3.2 PELLET [pellet]

Pellet es un razonador de código abierto escrito en Java y desarrollado para soportar ontologías de tipo OWL DL. Está basado en algoritmos *tableaux* desarrollados para llevar a cabo inferencias en DL. Soporta toda la expresividad de OWL DL, incluyendo razonador para nominales (o clases enumeradas). Pellet lleva a cabo los siguientes tipos de inferencias:

1. Comprobación de consistencia. Se asegura de que la ontología no contiene ninguna contradicción.
2. Concepto satisfactorio. Comprueba si es posible para una clase tener alguna instancia.

3. Clasificación. Establece la relación entre cada clase con nombre para crear la jerarquía.
4. Comprensión. Encuentra la clase más específica a la que un individuo pertenece. Esto sólo puede realizarse después de la clasificación.

En la Figura 19 [Parsia,Sirin] puede verse la arquitectura del razonador Pellet. Una ontología OWL se transforma en tripletas RDF, que se convierten en aserciones y en axiomas en la base de conocimiento. Si la ontología es de tipo OWL-Full, Pellet utiliza cierto tipo de heurística para reparar la ontología. Pellet guarda los axiomas de clase en la *TBox* y las aserciones acerca de individuos en la *ABox*. El razonador *tableau* utiliza las reglas de *tableau* estándar e incluye varios mecanismos de optimización estándar.

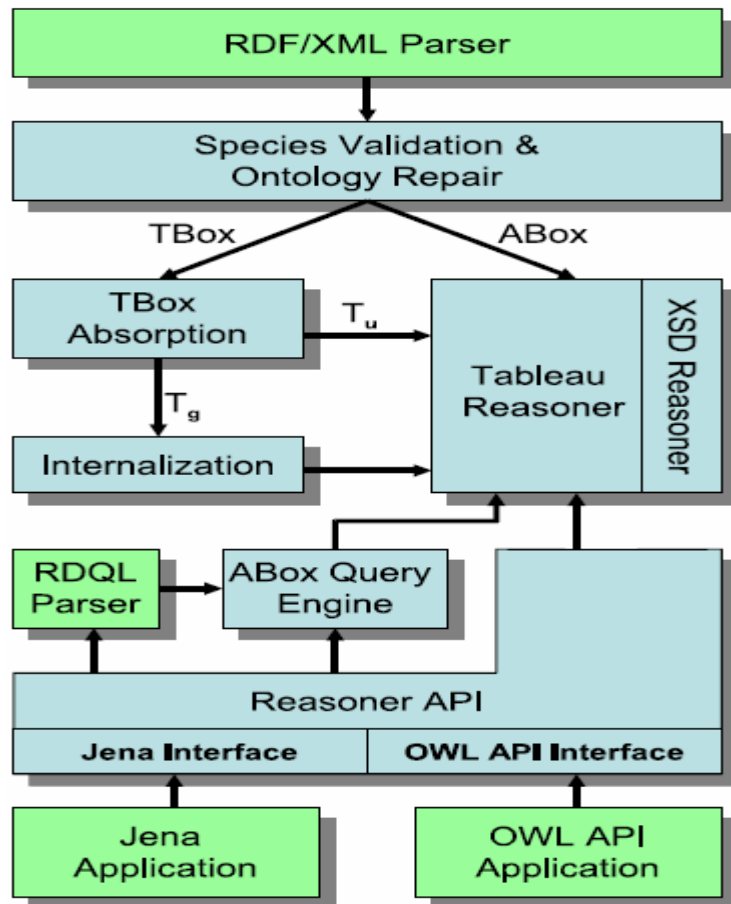


Figura 19 Arquitectura del razonador Pellet [Parsia,Sirin]

En cuanto a la respuesta de preguntas de tipo conjuntivo, Pellet incluye un mecanismo que soporta preguntas para *ABox* con variables de tipo distinguido o no. Las variables de tipo distinguido son aquellas que se introducen en el axioma como ?x y que le indican al servidor que son variables que éste tiene que vincular, mientras que las de tipo no distinguido se introducen como !x y representan para el servidor que no las tiene que vincular. Sin embargo, hay que tener en cuenta que responder a preguntas que contienen ciclos con variables no distinguidas con respecto a OWL DL es un problema que no está resuelto y que aún se está desarrollando. Ciclos con variables distinguidas si están soportados

Las preguntas pueden ser formuladas utilizando el lenguaje SPARQL. Pero Pellet sólo soporta las preguntas conjuntivas para *ABox*. Una consulta *ABox* es aquella acerca de los individuos y sus relaciones con los datos y con otros individuos a través de las propiedades definidas en la ontología. En SPARQL, una consulta *ABox* debe satisfacer tres condiciones:

1. No contener variables en la posición del predicado.

Un ejemplo no válido sería el que se muestra a continuación, ya que el predicado (?f:?n) es variable.

```
PREFIX foaf:<http://XMLns.com/foaf/0.1/>
SELECT ?name
WHERE { ?x ?f:?n ?name }
```

Un ejemplo válido sería:

```
PREFIX foaf: <http://XMLns.com/foaf/0.1/>
SELECT ?x ?name
WHERE { ?x foaf:name ?name }
```

2. Cada propiedad utilizada en la posición del predicado debe ser una propiedad definida en la ontología o una de las siguientes: rdf:type, owl:sameIndividualAs, owl:differentFrom.
3. Si rdf:type se está utilizando en la posición del predicado, se utilizará una URI constante en la posición del objeto para denotar la clase OWL (o la clase de expresión). Por ejemplo:

```
BASE <http://www.w3.org/2004/Talks/17Dec-sparql/data/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
PREFIX foaf: <http://XMLns.com/foaf/0.1/>
SELECT ?person
FROM NAMED <aliceFoaf.ttl>
FROM NAMED <bobFoaf.ttl>
FROM NAMED <celineFoaf.ttl>
FROM NAMED <danFoaf.ttl>
FROM NAMED <eveFoaf.ttl>
WHERE {
  GRAPH ?g { ?person rdf:type foaf:Person } .
}
```

Pellet soporta las consultas que incluyen SELECT, CONSTRUCT y ASK, pero DESCRIBE, OPTIONAL y FILTER no.

XML Schema tiene una gran colección de tipos de datos, incluyendo los numéricos (como enteros y flotantes), los strings y datos de tiempo y fecha. También permite crear nuevos tipos de datos. En este aspecto, Pellet puede testear si los tipos de datos creados por el usuario son o no correctos. Los tipos de datos definidos por el usuario pueden ser descritos en un esquema XML externo.

En cuanto a la depuración que realiza Pellet, ofrece un diagnóstico que permite saber qué axiomas causan, por ejemplo, una inconsistencia o cuáles han sido los conceptos insatisfactorios. Estos resultados se producen en hilos (*threads*) de ejecución distintos. Existe una versión de *Swoop* (<http://www.mindswap.org/2004/SWOOP>), que utiliza Pellet, que integra tanto la explicación como la generación para facilitar al usuario la depuración.

## 6 Diseño

El objetivo del diseño es la obtención de una especificación o modelo conceptual del sistema (lo más detallado posible) que, sobre los resultados obtenidos en el análisis, sirva como base posterior en la implementación y nos permita conocer los principales problemas y las soluciones de desarrollo que se han llevado a cabo durante el proceso de construcción de la aplicación.

El primer paso es identificar los distintos módulos que intervienen en la aplicación y sus principales responsabilidades y dependencias dentro de la misma. De esta forma, se identificaron cinco responsabilidades bien definidas y se organizaron en cinco módulos, que aparecen representados en paquetes en la Figura 20.

- Modelo de datos
- Ejecución de los tests
- Gestión del Razonador
- Persistencia de la información
- Interfaz gráfica de usuario

En el diseño de estos módulos se ha tratado de minimizar el acoplamiento para evitar que cambios en uno de los módulos afecten significativamente a los demás. El paquete “Modelo de datos” es el encargado de gestionar todas las clases que intervienen en la representación y el tratamiento de los datos. El paquete “Ejecución de los tests” es el encargado de gestionar todas las clases que intervienen en el proceso de ejecución de los tests. El paquete “Gestión del razonador” controla las clases encargadas de especificar y controlar el razonador utilizado sobre la ontología. El paquete “Persistencia de la información” se encarga de gestionar el almacenamiento de los datos generados de forma eficiente, y, por último, el paquete “Interfaz gráfica de usuario” controla las clases involucradas en la interfaz gráfica, estableciendo un marco de trabajo sencillo, amigable e intuitivo.

El diagrama de paquetes hace referencia a la distribución de las clases dentro de la aplicación. Nuestras clases han sido organizadas en los módulos que aparecen representados en la Figura 20. Todos ellos, a excepción de *GUI* y *Persistencia* forman parte de un único paquete Java, el *Modelo*, pero han sido divididos y agrupados de esta forma por sencillez y en función de las responsabilidades de cada uno. De esta forma, el *Modelo* está formado tanto por el paquete *EjecucionTests* como por el *InterfazRazonador* y el *Driver*.

El paquete *GUI* hace mención a todo lo referente a la interfaz gráfica de usuario. El paquete *Persistencia* contiene todas las clases necesarias para gestionar la persistencia de los datos en la aplicación. El paquete *Modelo* contiene todas las clases relacionadas con la representación de los tests y de la ontología. El paquete *EjecucionTest* contiene las clases encargadas de controlar y gestionar la ejecución de los tests. El paquete *InterfazRazonador* es el encargado de gestionar la fachada para trabajar con el razonador. Finalmente, está el paquete *Driver* que contiene la clase que implementa los métodos necesarios para trabajar con el razonador.

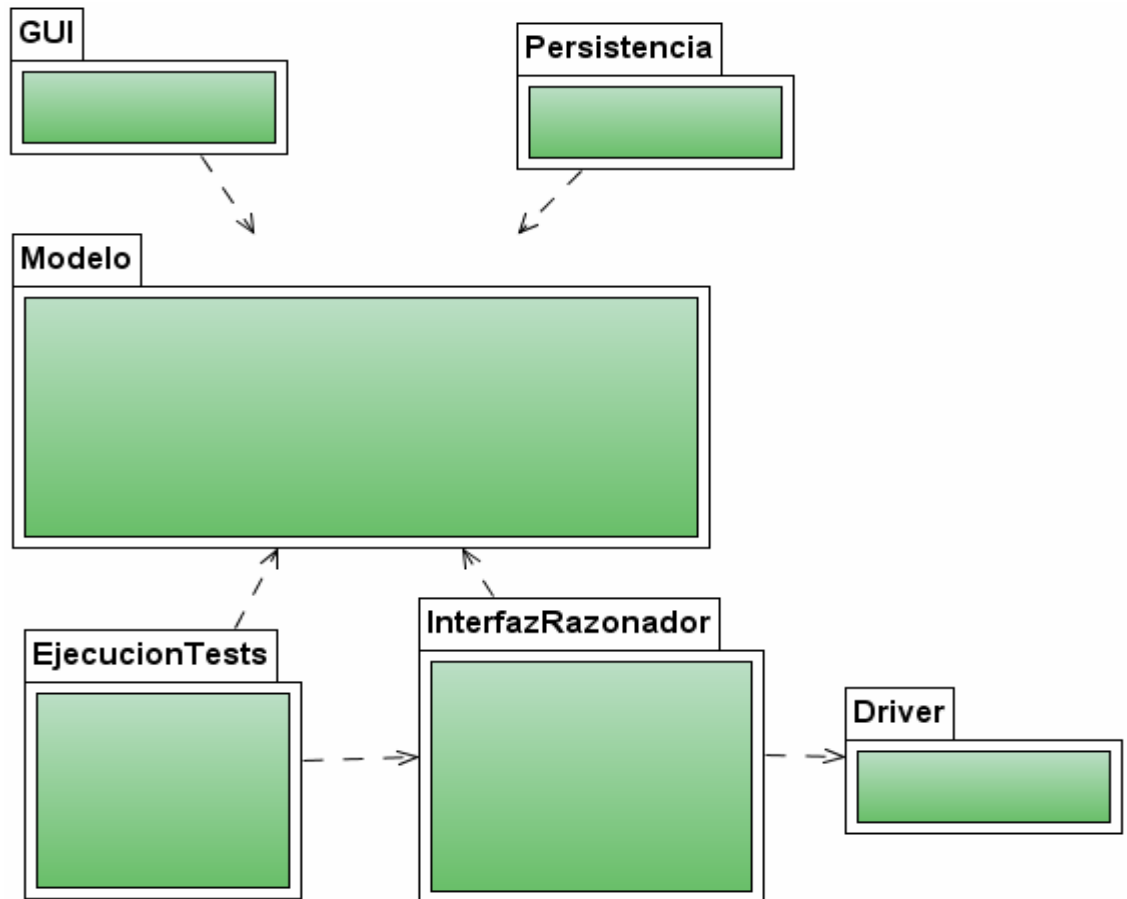


Figura 20 Diagrama de paquetes de la aplicación

A lo largo de este capítulo se irán comentando en más detalle cada uno de éstos módulos, sus responsabilidades dentro de la aplicación, los problemas surgidos en su desarrollo y las soluciones aplicadas para corregirlos. Este diseño formará el esqueleto sobre el que se sustente la herramienta.

## 6.1 Modelo de Datos

La forma en que los datos se representan en el sistema y el tratamiento que se hace de los mismos es fundamental. Una mala representación de los datos puede afectar notablemente al rendimiento de la aplicación.

El modelo de datos es el conjunto de clases que permitirán representar los tests y la ontología en el sistema. Como se explicó con anterioridad y tal y como se ve en la Figura 21, el modelo está formado también por las clases encargadas de la ejecución de los tests, de la gestión del razonador y del driver, pero en este apartado nos centraremos



únicamente en aquellas que hacen referencia a la representación de los tests y de la ontología.

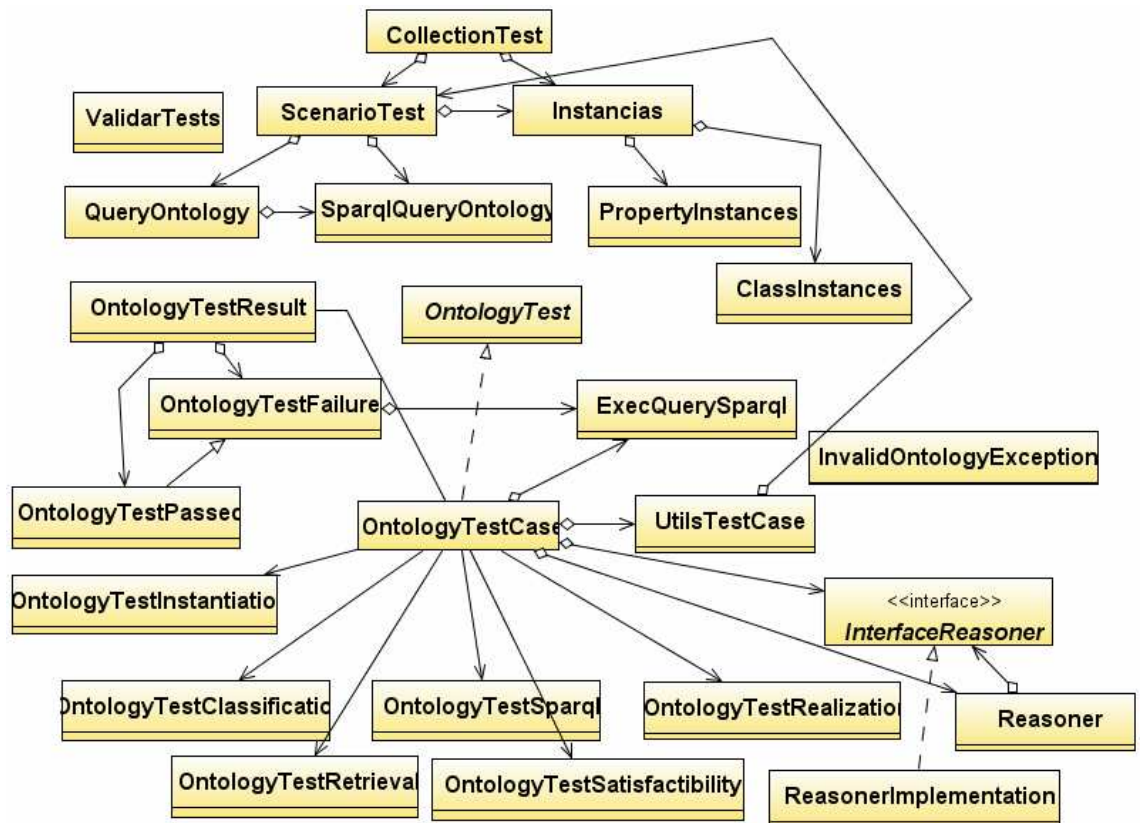


Figura 21 Diagrama de clases del paquete del modelo

### 6.1.1 Representación de los Tests y de la Ontología

La Figura 22 refleja las clases encargadas de representar los tests y la ontología en la aplicación.

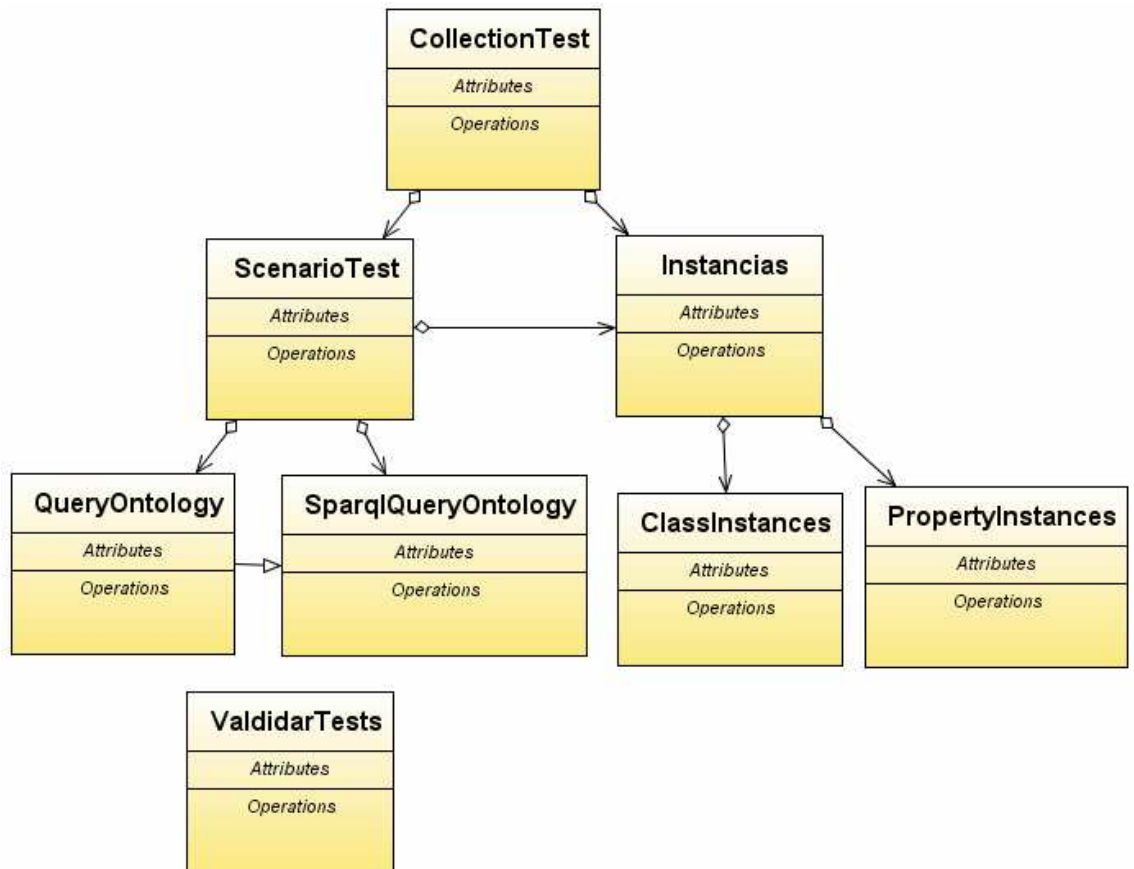


Figura 22 Diagrama con clases que representan los tests y la ontología

Para poder representar los tests, en primer lugar se determinó con qué tipos de tests se iba a trabajar. Éstos se clasificaron en dos grupos, Test Simples y Test SPARQL.

Los **Tests Simples** fueron definidos para realizar tests sin necesidad de utilizar el lenguaje de consultas SPARQL. Esto permite que un usuario poco familiarizado con dicho lenguaje pero que quiera realizar una serie de pruebas a su ontología, pueda hacerlo. Los Tests Simples son: Tests de Instanciación, Test de Recuperación, Test de Realización, Test de Satisfactibilidad y Test de Clasificación. Cada uno de ellos realiza un tipo de prueba sobre la ontología por lo que su comportamiento es diferente. Para poder gestionarlos, se añadió el atributo de *TipoTest* a la clase *ScenarioTest*, como puede verse en la Figura 23.

Los **Tests Sparql** son aquellos que utilizan el lenguaje ontológico de consultas Sparql, un lenguaje complejo y que requiere de conocimientos específicos por parte del usuario.

El siguiente paso, una vez definidos los tipos de tests, fue definir cómo se van a identificar en el sistema. Se tomó la decisión de identificarlos mediante un nombre único dado por el usuario, ya que resulta bastante ilógico y confuso el hecho de tener dos tests con el mismo nombre. De forma intuitiva, el usuario nombrará a sus tests de distinta forma para poder identificarlos posteriormente.

Una vez determinado cómo se van a identificar los tests dentro del sistema, se tiene que establecer la manera en la que se van a almacenar los datos que representan cada test y qué datos son necesarios almacenar para que un test aparezca representado de forma completa.

Un test está formado por el conjunto de consultas que el usuario realizó sobre la ontología y por el conjunto de instancias que el usuario le asoció. Al existir dos posibles tipos de test, también existen dos posibles tipos de consultas, las simples y las sparql, luego el test estará representado por dos tipos distintos de consultas. En la Figura 23 puede verse la clase *ScenarioTest* en dónde se refleja todo lo explicado con anterioridad.

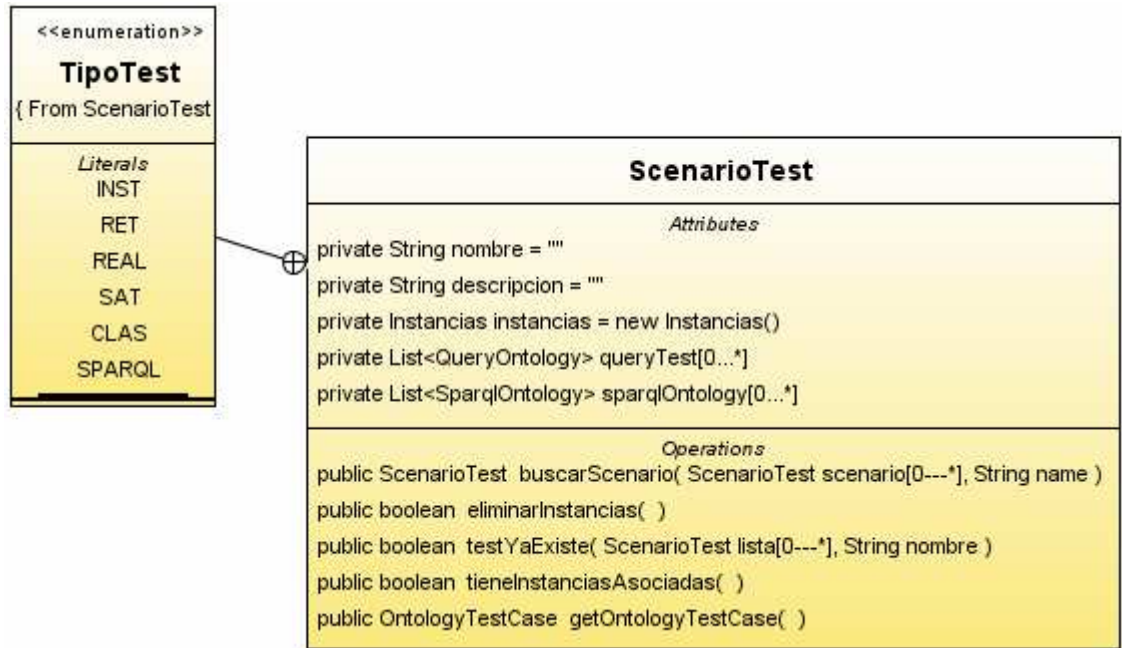


Figura 23 Detalle de la clase ScenarioTest

La definición de las consultas para ambos tipos de tests (simples y sparql) es la misma, es decir, en ambos casos una consulta será representada mediante una cadena de caracteres que contiene la consulta y una cadena de caracteres que contiene el resultado. Además, en el caso de los Tests Simples, será posible añadir un comentario a la consulta, tal y como puede verse en la Figura 24.

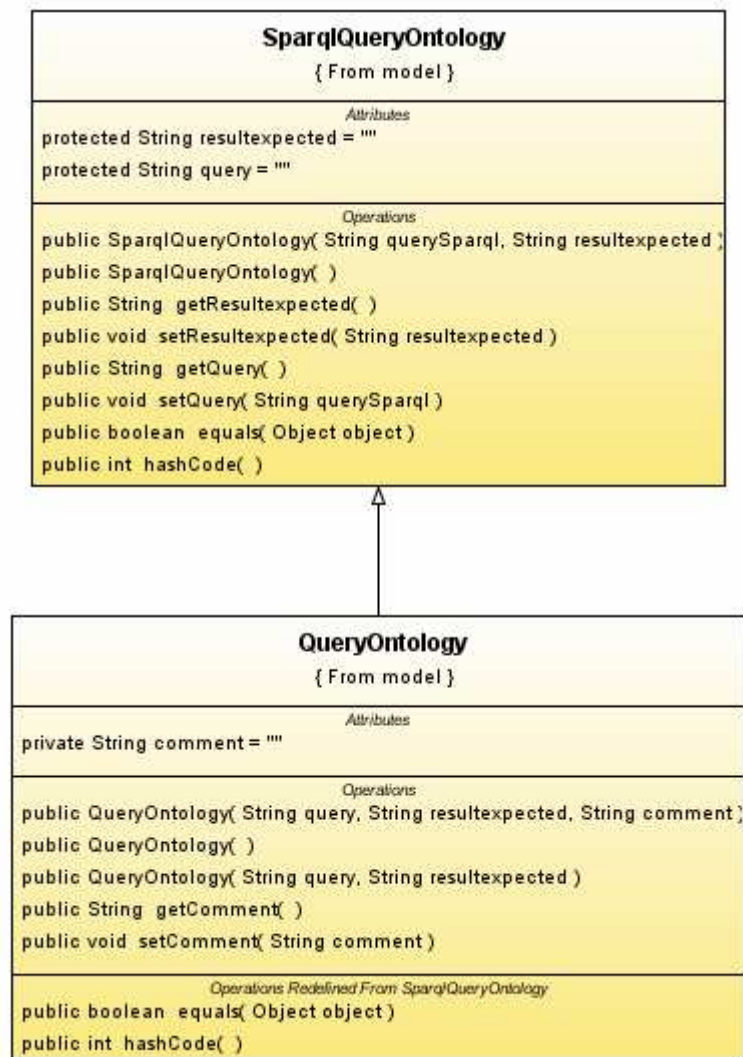


Figura 24 Detalle de las clases SparqlQueryOntology y QueryOntology

Un punto clave a la hora de representar los tests es cómo se va a indicar para cada uno de ellos la base de conocimiento de la ontología sobre la que el test se va a ejecutar. El objetivo de los tests es deducir conocimiento de esta base y para cada test realizado, la base de conocimiento asignado a él puede variar. La forma de asociar esta base de conocimiento a los tests es a través de las instancias.

Las instancias representan la base de hechos de la ontología y son la forma que tiene el usuario de asociar conocimiento a la misma e ir creando la base de hechos. Esto es importante ya que los tests que realice el usuario estarán pensados para evaluar la base de hechos y, para cada test que realice, el conjunto de instancias puede cambiar.

Las instancias se identifican en el sistema de la misma forma que los tests, mediante su nombre, que actuará como identificador único, ya que no tiene sentido y sería confuso para el usuario el hecho de tratar conjuntos de instancias distintos con el mismo nombre. Las instancias, tal y como se ve en la Figura 25 aparecen representadas por un nombre (identificador único), una descripción, un conjunto de instancias de clase y un conjunto de instancias de propiedad. Las instancias de clase y de propiedad son los dos tipos de instancias que se pueden asociar a una ontología.

Instancias
<i>Attributes</i> <pre>private String nombre = "" private String descripcion = "" private List&lt;ClassInstances&gt; classInstances[0...*] private List&lt;PropertyInstances&gt; propertyInstances[0...*]</pre>
<i>Operations</i> <pre>public Instancias( ) public Instancias buscarInstancias( Instancias inst[0...*], String nombre ) public boolean instanciasYaExiste( Instancias inst[0...*], String nombre ) public boolean eliminarInstancias( ClassInstances clasInst[0...*], PropertyInstances propInst[0...*] )</pre>

Figura 25 Detalle de la clase Instancias

Tanto las instancias de clase como las de propiedad, tal y como se ve en la Figura 26, están definidas mediante un atributo que representa el valor que se desea añadir a la ontología para ir creando la base de conocimiento y por un comentario opcional.

ClassInstances	PropertyInstances
<i>Attributes</i> <pre>private String classInstance = "" private String comment = ""</pre>	<i>Attributes</i> <pre>private String propertyInstance = "" private String comment = ""</pre>
<i>Operations</i> <pre>public ClassInstances( )</pre>	<i>Operations</i> <pre>public PropertyInstances( )</pre>

Figura 26 Detalle de las clases ClassInstances y PropertyInstances

Una vez definidas las instancias de clase, de propiedad y las consultas que se pueden realizar, es necesario validarlas. La forma de realizar una consulta o de introducir una instancia debe tener un formato fijo y bien definido, al que debe ceñirse el usuario, y que vendrá dado por el tipo de test que se esté realizando o por el tipo de instancia que se esté creando. Es decir, dependiendo del tipo de test o de instancia, el formato de la consulta será distinto y se necesitará tener un mecanismo de validación para cada uno.

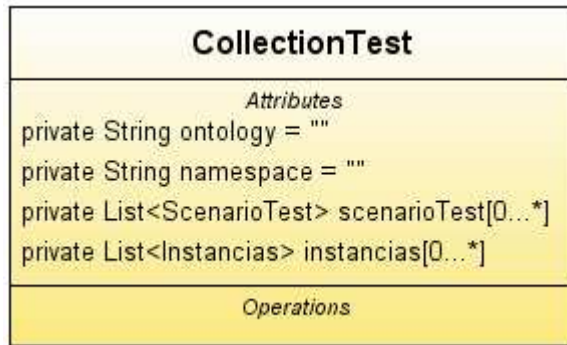
Para solucionar este problema se añadió la clase *ValidarTests* cuya responsabilidad es determinar si la consulta o instancia introducida por el usuario es válida dependiendo del tipo de test al que pertenezca. Para ello la clase contiene una serie de métodos que permiten validar cada tipo de consulta y cada tipo de instancia. La clase *ValidarTests* está representada en la Figura 27.

<b>ValidarTests</b>
<i>Attributes</i>
<i>Operations</i>
public boolean validarQuery( query )
public boolean validarResultado( res )
public boolean validarResultadoInstSatis( res )
public boolean validarQueryInstSatis( query )
public boolean validarInstanciaClase( query )
public boolean validarInstanciaPropiedad( query )
public boolean validarSparqlResult( query )
public boolean validarQuerySparql( query )

**Figura 27** Detalle de la clase *ValidarTests*

Una vez definido cómo se van a representar los tests se debe especificar cómo va a representarse la ontología. Ésta va a quedar definida por su *namespace* y por su ubicación física, atributos de la clase *CollectionTest*. El modelo de datos para la representación de los tests quedará definido, teniendo en cuenta lo explicado con anterioridad, por un conjunto de escenarios, un conjunto de instancias, el *namespace* y la ubicación física de la ontología, tal y como se ve en la Figura 28.





**Figura 28** Detalle de la clase **CollectionTest**

### 6.1.2 Ejecución de los Tests

En este apartado se presenta la parte del modelo correspondiente a la ejecución de los tests. Se explicarán en detalle las clases involucradas en la ejecución de los tests, que se encuentran en el paquete *EjecucionTests* de la Figura 20.

El diseño de este módulo se ha realizando siguiendo parte de la arquitectura de JUnit para realizar pruebas de software, utilizando algunos de los patrones sobre los que se basa JUnit, tales como: *Command*, *Themplate Method* y *Collecting Parameter*, y su diagrama de clases puede verse en la Figura 29.



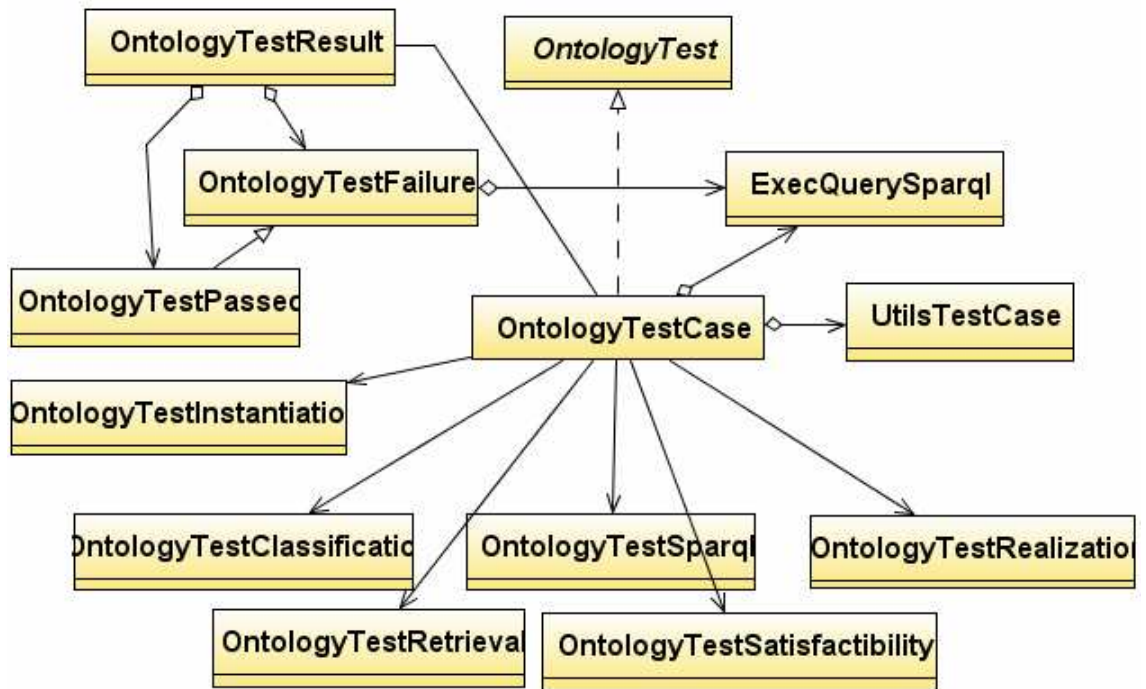


Figura 29 Diagrama de clases del módulo de ejecución de los tests

En nuestro caso serán dos los problemas principales a resolver: la ejecución de los test y el almacenamiento de los resultados obtenidos tras la ejecución. Para resolver el primer problema hemos empleado los patrones *Command* y *Template Method* mientras que en la segunda será el patrón *Collecting Parameter*.

En primer lugar se definió cómo iban a ejecutarse los tests en el sistema. El problema principal es que hay hasta seis tipos distintos de tests en la aplicación y es necesaria una forma uniforme de ejecutarlos todos. Para conseguir esto se ha empleado el patrón *Command*. Este patrón establece el marco de trabajo necesario para poder combinar la ejecución de cualquier tipo de test en un único objeto, que contendrá el comportamiento y los datos necesarios para realizar una acción específica. De esta forma, la aplicación llamará siempre al mismo método para ejecutar un test, el método *run()* en nuestro caso, que contendrá los datos necesarios y el comportamiento para realizar la ejecución de dicho test. La aplicación ya no necesitará, por tanto, conocer cada opción disponible para cada tipo de test, sino que, a través de un único método, podrá realizar la acción oportuna. De esta forma, independientemente del test que se ejecute en un momento dado, siempre se realizará su ejecución. La clase *OntologyTestCase* será la que utilizará éste patrón, encapsulando la acción de ejecutar

un test como un objeto y permitiendo gestionar ejecuciones de tests de forma independiente.

En segundo lugar se definieron los pasos que debe seguir un test para ejecutarse correctamente. El primer paso es añadir las instancias asociadas al tests a la ontología (inicializarla), proceso identificado con el método *setUpOntology* en la clase *OntologyTestCase* de la Figura 30. Éste es un punto clave y muy importante, ya que los resultados de los tests se obtienen en función de la base de conocimiento asociada a la ontología, que viene dada por estas instancias y que debe ser independiente para cada test en cada ejecución.. Es decir, la base de conocimiento que tiene asociada la ontología en el momento de ejecutar un test no debe influir en la ejecución de otro test, que tendrá, por ejemplo, otra base de conocimiento.

Para que esta independencia sea posible, tras cada ejecución de los tests, se deben de eliminar las instancias que fueron asociadas a la ontología para su ejecución y dejarla vacía para la siguiente ejecución, donde serán asociadas las instancias correspondientes. Para ello tras cada ejecución se llamará al método *tearDownOntology()* que será el encargado de realizar este proceso.

Estos pasos son, en realidad, una secuencia estática de métodos en cada ejecución: el método que configura el escenario de ejecución, el método que realiza la ejecución propiamente dicha y el método que devuelve la ontología a su estado inicial. Como puede verse en la Figura 30 se ha aplicado el patrón *Themplate Method* para implementar estas acciones, ya que éste patrón define el esqueleto de un algoritmo en una operación, con lo que se tiene una forma simple y flexible de resolver el problema. En nuestro caso, el esqueleto serán los pasos comentados con anterioridad y la operación donde aparecen definidos será el método ejecutar (método *run()* en la Figura 30) también comentado.

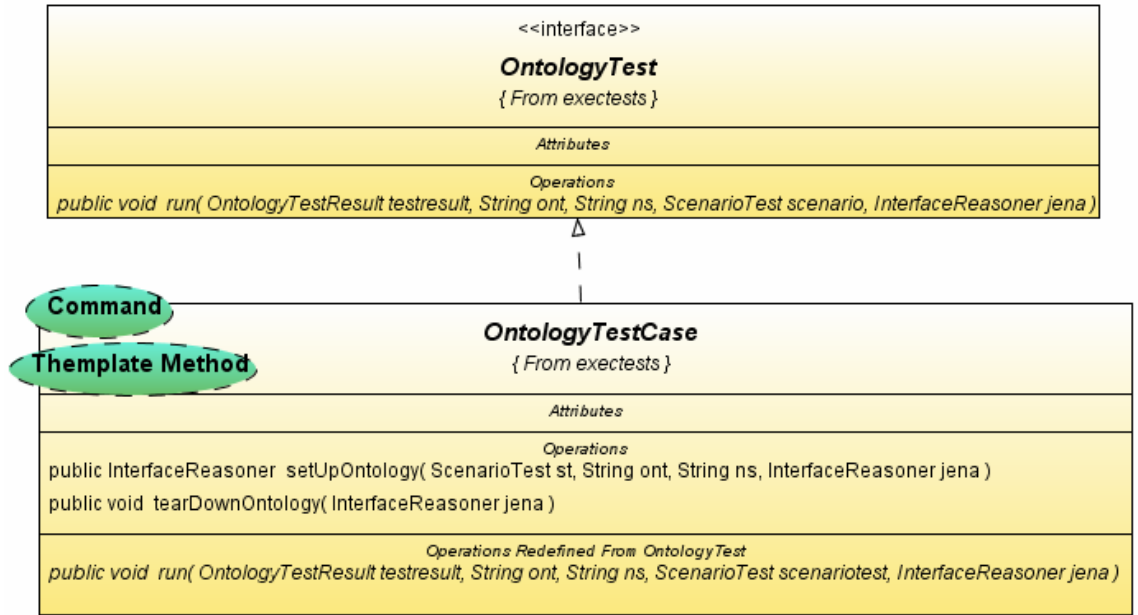


Figura 30 Detalle de la intervención de los patrones Command y Template Method

El tercer problema que se presentó fue cómo recoger y almacenar los resultados de los tests ya que, posteriormente, haría falta emitir un informe sobre el resultado de la ejecución de los tests. Era necesario en cada ejecución saber el tests actual que se estaba ejecutando y qué resultados había producido, es decir, qué consultas habían sido correctas y cuáles habían fallado.

La solución al problema viene dada por el patrón *Collecting Parameter*, tal y como muestra la Figura 31, el mismo utilizado en *JUnit*. Este patrón especifica que se ha de añadir un parámetro al método que ejecuta el test (`run()`) y pasarle un objeto que recogerá los resultados. En nuestro caso, el objeto será de tipo *OntologyTestResult* e irá almacenando los resultados en dos listas distintas, una contenedora de los test que han fallado (*OntologyTestFailure*) y otra con los tests que han sido correctos (*OntologyTestPassed*), tal y como puede verse en la Figura 32. De esta forma, para cada ejecución de un test, dependiendo de si éste ha pasado o ha fallado, el objeto añadirá a su lista correspondiente un elemento, que contendrá el nombre del test, la consulta realizada y el resultado obtenido. Un diagrama representativo de la ejecución de un test puede verse en el Diagrama 1.

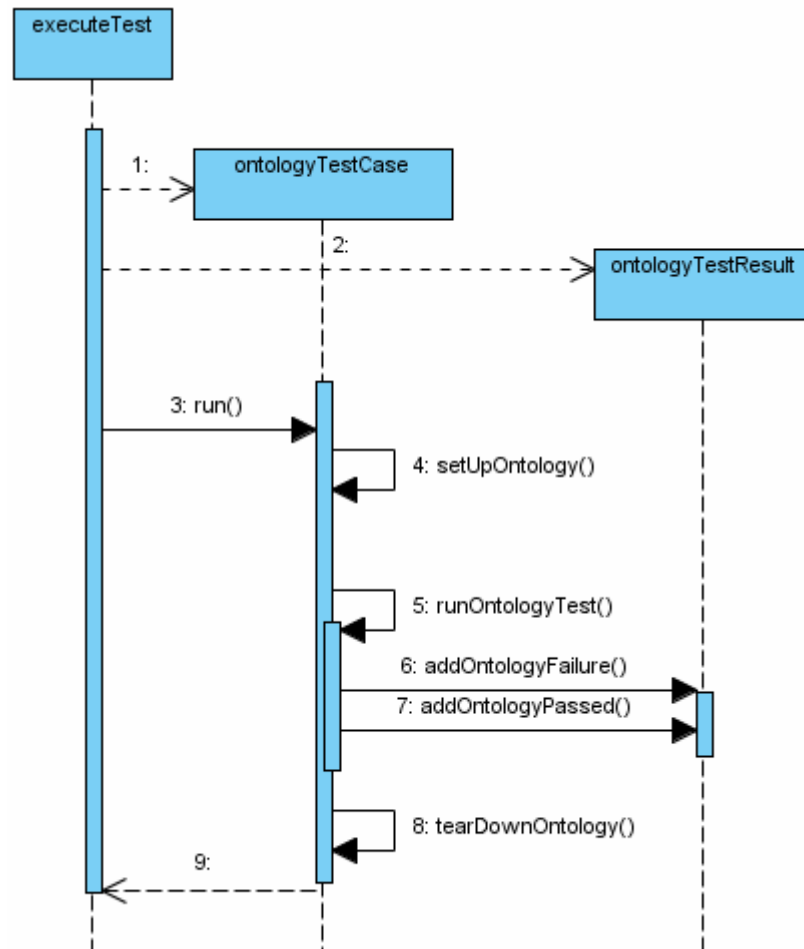


Diagrama 1 Diagrama de Secuencia para la ejecución de los tests

La clase *ExecuteTest* es la encargada de ejecutar los tests. Para ello crea, en primer lugar, dos objetos: *ontologyTestCase* (1: en Diagrama 1) y *ontologyTestResult* (2: en Diagrama 1). El objeto *ontologyTestCase* llama al método *run()* (3: en Diagrama 1) de *OntologyTestCase*, al que se le pasan como parámetros los siguientes datos: el objeto *ontologyTestResult* recientemente creado, la ontología y su *namespace*, y el escenario a ejecutar.

Este método *run()* realiza los pasos necesarios para ejecutar el test, llamando para ello de forma secuencial a los siguientes métodos: *setUpOntology()*, *runOntologyTest()* y *tearDownOntology()* (pasos 4: 5: y 8: de Diagrama 1 respectivamente). El método *setUpOntology()* carga las instancias asociadas al test a la ontología; el método *runOntologyTest()* ejecuta el test correspondiente y evalúa los resultados obtenidos, añadiendo éstos al objeto *ontologyTestResult* en su lista de

aciertos (paso 6 del Diagrama 1), si el test ha sido correcto, o en la de fallos (paso 7 en Diagrama 1) en caso contrario; y el método *tearDownOntology()* elimina las instancias que fueron asociadas a la ontología.

Una vez concluidos estos tres pasos secuenciales, el control de la aplicación vuelve a la clase *ExecTest* (paso 9 del Diagrama 1).

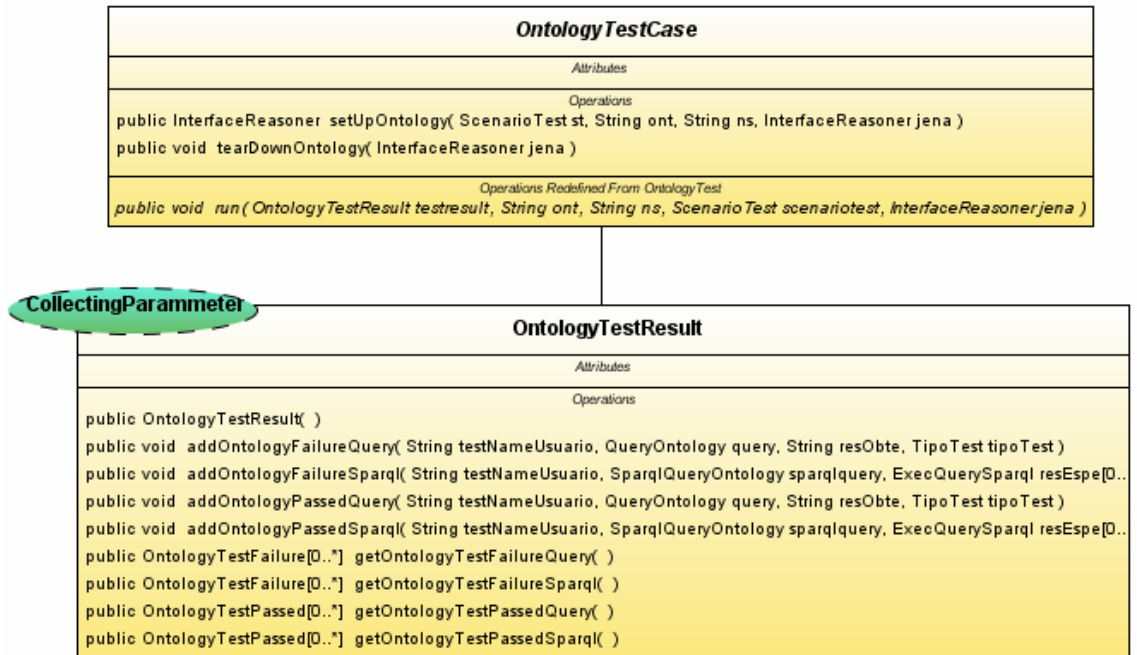


Figura 31 Detalle del patrón Collecting Parameter

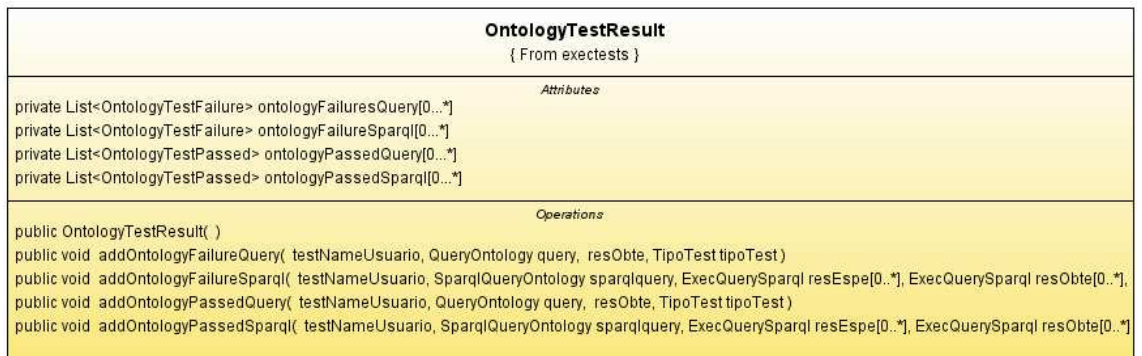


Figura 32 Detalle de la clase OntologyTestResult

Las clases *OntologyTestFailure* y *OntologyTestsPassed* serán las encargadas de almacenar los datos necesarios para identificar de forma completa y correcta los errores o aciertos en los tests realizados. Para ello están formadas por una serie de atributos que identifican el test, las consultas realizadas, los resultados obtenidos y el tipo de test, tal y como se ve en la Figura 33 en el caso de *OntologyTestFailure*.

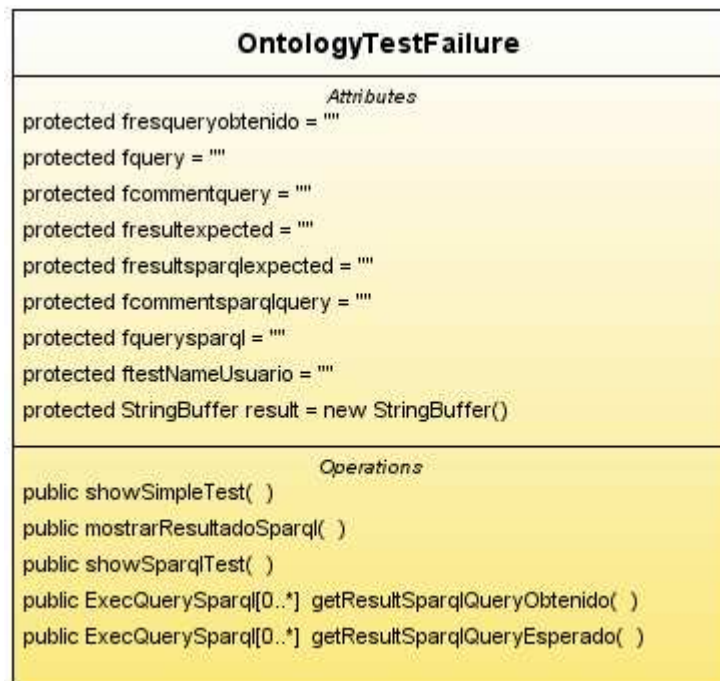


Figura 33 Detalle de la clase *OntologyTestFailure*

En la Figura 33 se observa que el resultado de las consultas Sparql se obtiene en una lista contenedora de la clase *ExecQuerySparql*, que puede verse más en detalle en la Figura 34.

Esto se debe a que a la hora de evaluar si una consulta Sparql ha sido correcta o no, el formato del resultado que proporciona *Jena* (*framework* que utilizamos para realizar consultas sparql en Java) es distinto al formato introducido por el usuario para especificar el resultado esperado, por lo que a la hora de realizar las comparaciones para ver si coinciden o no y así saber si es o no correcta, se presentan dificultades.

Para solucionar este problema se creó la clase *ExecQuerySparql*. Esta clase está formada por una cadena de caracteres que representa el nombre del *SELECT* realizado en la consulta Sparql y por una lista contenedora de cadenas de caracteres que representan los datos recuperados tras la ejecución de la consulta para ese *SELECT*. A través de esta clase, los resultados proporcionados por el razonador al ejecutar la consulta Sparql se van almacenando de la forma descrita y pueden ser comparados con los introducidos por el usuario, al estar estos almacenados de la misma forma.

ExecQuerySparql
<i>Attributes</i> private datos[0..*] = new ArrayList<String>() private nombreSelect = ""
<i>Operations</i> public ExecQuerySparql( ) public [0..*] getDatos( ) public void setDatos( datos[0..*] ) public getNombreSelect( ) public void setNombreSelect( nombreSelect )

Figura 34 Detalle de la clase ExecQuerySparql

### 6.1.3 Gestión del Razonador

Una de las cuestiones más importantes de éste proyecto es el razonador que se va a emplear para poder inferir y deducir el conocimiento necesario sobre la ontología que nos permita realizar los tests. El problema planteado en este punto, es qué ocurriría si uno o varios usuarios quisieran utilizar la herramienta con razonadores distintos. No se debería de restringir el uso de la aplicación a un razonador concreto, puesto que, actualmente, existen un gran número de razonadores ontológicos sobre los que se puede trabajar y el hecho de darle libertad al usuario para que pueda utilizar el que más le interese es una opción muy buena.

La solución ha sido el hecho de permitir al usuario utilizar un razonador y un *framework* distinto a los añadidos a la aplicación por defecto, de forma sencilla. Para ello, la clase *ReasonerImplementation* actúa como un “driver” en la aplicación, para que

si un usuario no desea trabajar a través de *Jena* o de *Pellet*, pueda hacerlo sin que la herramienta deje de funcionar, cambiando el driver original por el que él implemente.

Así, puede ser configurado de forma que sea posible utilizar la implementación del programa con otro *framework* que no sea el seleccionado inicialmente para desarrollar la aplicación y con otro razonador. Los módulos que forman parte de la gestión del razonador se ven en la Figura 35.

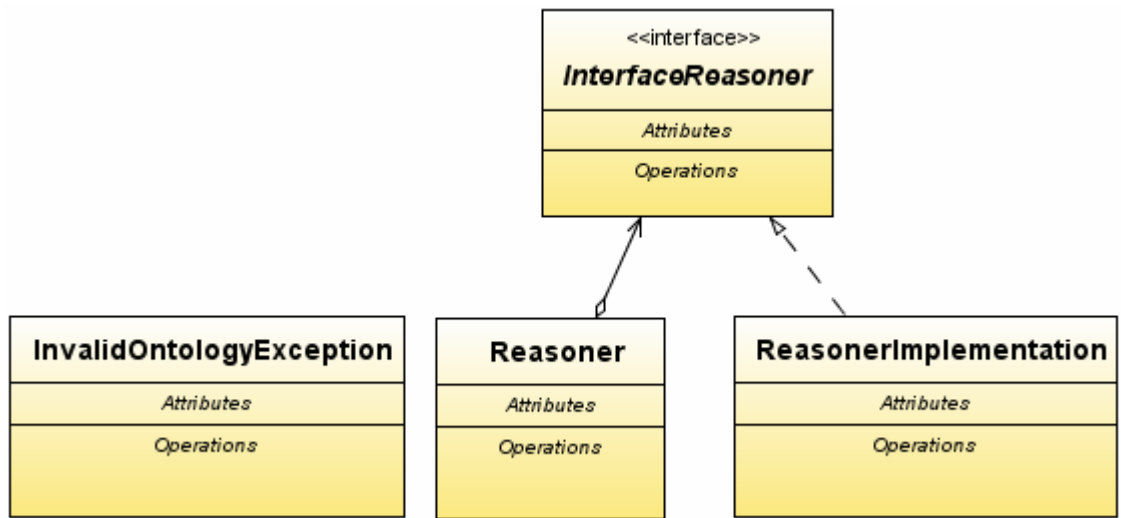


Figura 35 Diagrama de clases del razonador

La clase *ReasonerImplementation*, cuya definición puede verse en la Figura 36, consigue la independencia con el resto de los módulos y así su título de “driver”, gracias al uso de la carga dinámica de clases para crear una instancia de la misma, consiguiendo de esta forma no crear ninguna dependencia con el resto de la aplicación. El diagrama de secuencia para dicha carga dinámica de clases puede verse en el Diagrama 2.



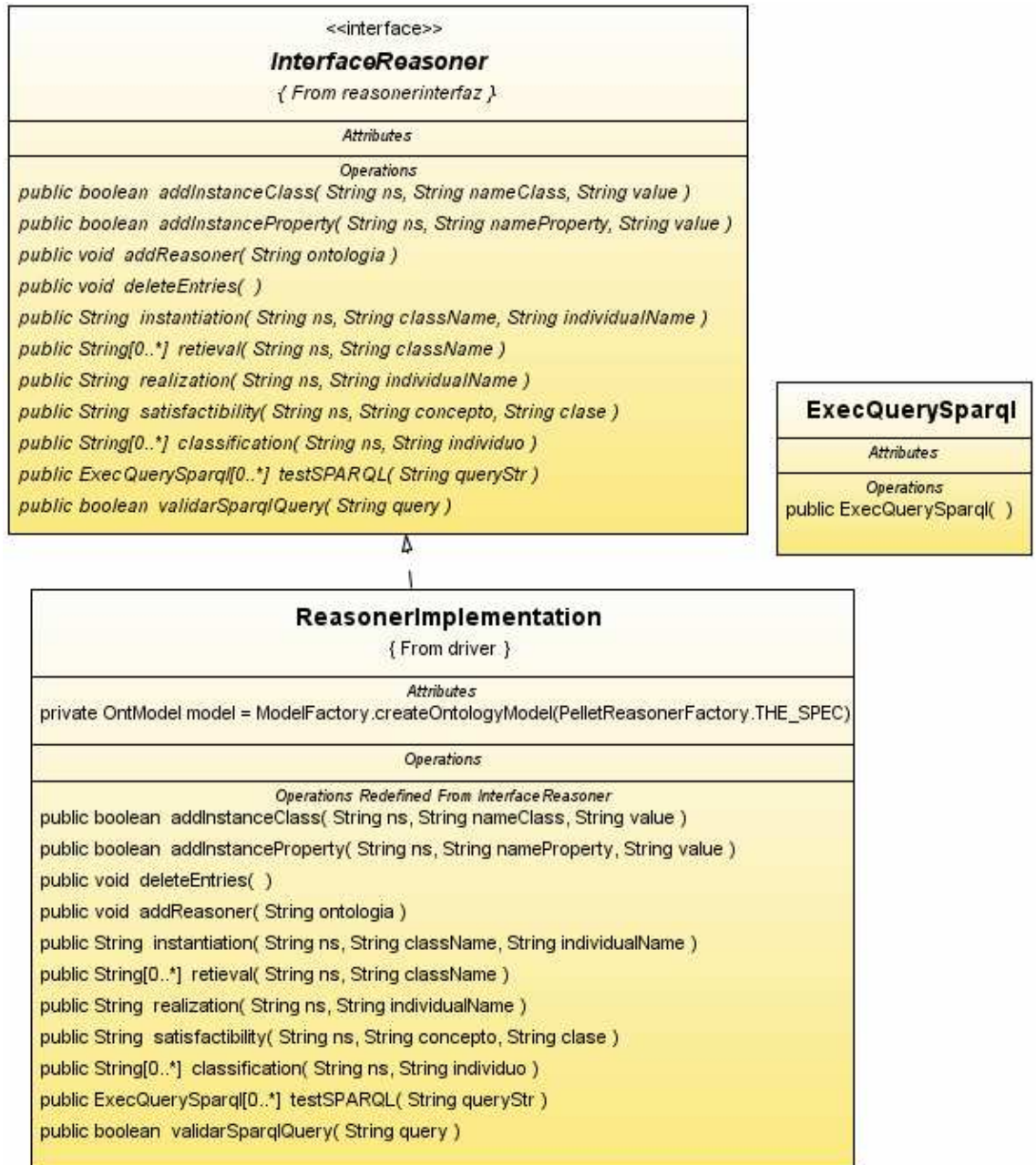


Figura 36 Detalle de las clases que implementan el Razonador

Cualquier clase de la aplicación que lo desee, a través del método `getReasoner()` de la clase *Reasoner* (ver Figura 37), puede obtener una nueva instancia de la clase *InterfaceReasoner*, cuya interfaz es implementada por el driver (la clase *InterfazReasonerImplementation*), sin que se produzca ninguna dependencia con la misma, gracias al uso de la carga dinámica de clases, a través de un fichero de propiedades que contiene el driver de la aplicación. En el Diagrama 2 puede verse representado de forma gráfica lo explicado con anterioridad.

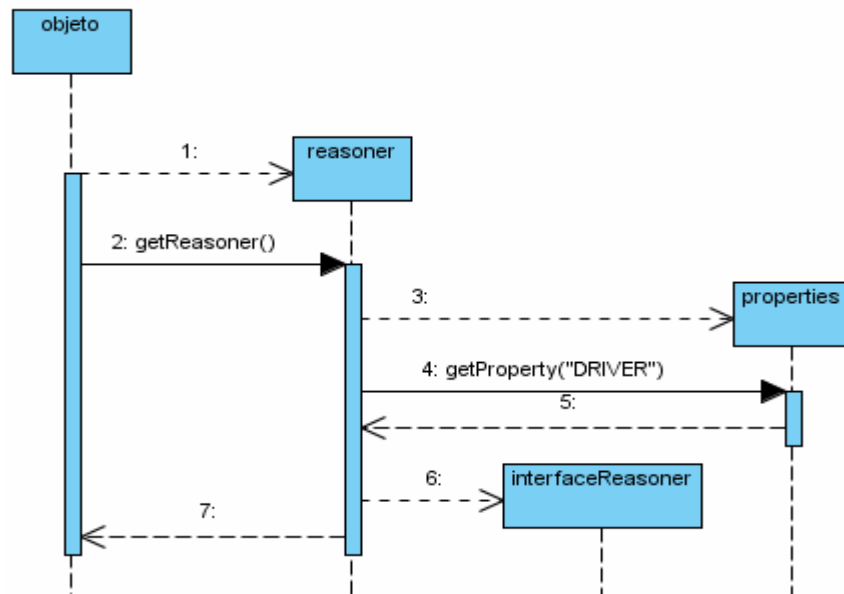


Diagrama 2 Diagrama de secuencia para la carga del razonador

A través del método *getReasoner()* de la clase *Reasoner* puede obtenerse un objeto de la clase *InterfaceReasoner*, a través del cual se tendrá acceso a todos los métodos implementados por el “driver” sin haber creado para ello ninguna dependencia.

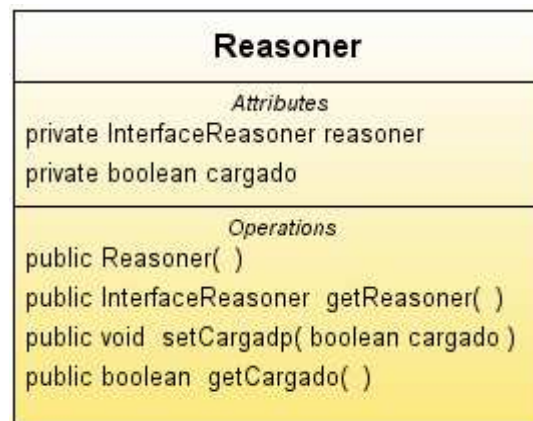


Figura 37 Detalle de la clase Reasoner

Para ello, la clase *Reasoner* utilizará la clase *Properties* y su método *getProperty()* (ver 4 en Diagrama 2) para obtener del fichero de configuración creado por la aplicación la clase de la que se quiere obtener una instancia; es decir, la ruta al driver de la aplicación. A través del método *getProperty()* se obtendrá una nueva instancia de la clase *InterfaceReasoner* (ver 6 en Diagrama 1), que será lo que será devuelto por el método *getReasoner()*.

Esto es, por tanto, un punto de extensión en la aplicación, a través del cual un usuario podrá adaptar la aplicación a su gusto, es decir, añadir un nuevo driver con el que trabajar. Los pasos a seguir para esto son los siguientes:

1. En primer lugar, hay que crear una nueva clase que implemente la interfaz *InterfaceReasoner* (o re-definir según convenga los métodos de la ya existente clase *InterfaceReasonerImplementation*, ver Figura 36). Esta clase define una serie de métodos que han de ser re-definidos según le interese al usuario. Estos métodos son los encargados de interactuar con la ontología, y se describen a continuación, indicando para cada uno de ellos las precondiciones, las poscondiciones y las tareas que deben cumplir.

- ***addReasoner***: este método es el encargado de asociar el razonador a la ontología. Precondiciones: la ontología a la que se asociará el razonador debe ser válida (consistente) y estar desarrollada en *OWL*. A este método se le pasa como parámetro una cadena de caracteres que representa la ontología con la que se va a trabajar. Poscondiciones: si la ontología no es válida se producirá una excepción de tipo *InvalidOntologyException*. En caso contrario, el razonador será asociado con la ontología.
- ***addInstanceClass***: este método es el encargado de añadir instancias de clase a la ontología. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. Este método recibe como parámetros tres cadenas de caracteres. La primera representa el *namespace* de la ontología, la segunda la clase y la tercera el individuo. Poscondiciones: El método devuelve verdadero en caso de que las instancias se añadan con éxito, y falso en caso contrario. No pueden añadirse instancias de clase que no estén definidas en la ontología.
- ***addInstanceProperty***: este método es el encargado de añadir instancias de propiedad a la ontología. Precondiciones: el razonador debe haber sido

seleccionado y asociado a la ontología. El método recibe como parámetros tres cadenas de caracteres que representan, respectivamente, el *namespace* de la ontología, el nombre de la propiedad, y el valor a asignar a la misma. Poscondiciones: las propiedades a las que se añadirán los individuos deben de formar parte de la ontología, en caso contrario se producirá un error. El método devolverá verdadero si las instancias se han añadido con éxito y falso en caso contrario.

- ***deleteEntries***: este método es el encargado de eliminar (desasociar) todas las instancias de la ontología. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. Poscondiciones: la ontología no contendrá instancias asociadas.
- ***instantiation***: este método es el encargado de realizar el Test de Instanciación, que consiste en deducir si un individuo pertenece a una clase. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. El método recibe como parámetros tres cadenas de caracteres que representan, respectivamente, el *namespace* de la ontología, la clase y el individuo implicado. Poscondiciones: El método devuelve “true” (en forma de cadena de caracteres) en caso de que el individuo pertenezca a la clase, o “false” en caso contrario. Si se produce algún error en el modelo a la hora de realizar las consultas, se producirá una excepción de tipo *InvalidOntologyException*.
- ***retrieval***: este método es el encargado de realizar el Test de Recuperación, que consiste en obtener todos los individuos que pertenecen a una clase. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. El método recibe como parámetros dos cadenas de caracteres. La primera representa el *namespace* de la ontología y la segunda la clase de la cual se quieren conocer sus individuos.

Poscondiciones: El método devuelve una lista de cadenas de caracteres, cada una de las cuales representa un individuo al que pertenece la clase.

- **realization**: este método es el encargado de realizar el Test de Realización, que consiste en deducir la clase más exacta a la que pertenece un individuo. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. El método recibe como parámetros dos cadenas de caracteres. La primera representa el *namespace* de la ontología y la segunda el individuo del cual se quiere conocer su clase más exacta. Poscondiciones: El método devuelve una cadena de caracteres que representa la clase más exacta a la que pertenece el individuo.
- **satisfactibility**: este método es el encargado de realizar el Test de Satisfactibilidad, que consiste en deducir si es posible añadir una nueva instancia de clase sin que la ontología deje de ser consistente. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. El método recibe como parámetros tres cadenas de caracteres. La primera representa el *namespace* de la ontología, la segunda el individuo a añadir y la tercera la clase de la que se quiere saber si se puede añadir el individuo. Poscondiciones: El método devuelve “true” (en forma de cadena de caracteres) en caso de que el individuo se pueda añadir a la clase, o “false” en caso contrario. Si se produce algún error en el modelo a la hora de realizar las consultas, se producirá una excepción de tipo `InvalidOntologyException`.
- **classification**: este método es el encargado de realizar el Test de Clasificación, que consiste en deducir todas las clases a las que pertenece un individuo. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. El método recibe como parámetros dos

cadenas de caracteres. La primera representa el *namespace* de la ontología y la segunda el individuo del cuál quieren conocerse todas las clases a las que pertenece. Poscondiciones: El método devuelve una lista de cadenas de caracteres, cada una de las cuales representa una de las clases a las que pertenece el individuo. Si se produce algún error en el modelo a la hora de realizar las consultas, se producirá una excepción de tipo *InvalidOntologyException*.

- ***testSPARQL***: este método es el encargado de realizar el Test Sparql, que consiste en obtener el resultado de una consulta Sparql. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. El método recibe como parámetro una cadena de caracteres que representa la consulta sparql introducida por el usuario. Poscondiciones: El método devuelve una lista de objetos *ExecQuerySparql*. Cada uno de estos objetos está formado por una cadena de caracteres, que representa uno de los datos SELECT de la consulta Sparql, y por una lista de cadenas de caracteres, que representan cada elemento obtenido para ese SELECT. Si la consulta sparql no es válida se producirá una excepción de tipo *InvalidOntologyException*.

Por ejemplo, si la consulta Sparql fuese la siguiente:

```
SELECT ?subject ?object
WHERE { ?subject rdfs:subClassOf ?object }
```

El método *testSPARQL* devolverá, después de haber ejecutado la consulta, una lista con dos elementos de tipo *ExecQuerySparql* (dos elementos ya que hay un SELECT con dos variables). El primer elemento contendrá un atributo, representado mediante una cadena de caracteres, con el valor “subject” y una lista de cadena de caracteres, que tendrá asignados los valores de todos los “subject” que se hayan obtenido de la ontología tras la ejecución de la consulta. El segundo elemento contendrá el valor “object” con sus elementos correspondientes.

- ***validarSparqlQuery***: este método es el encargado de validar la consulta Sparql. Precondiciones: el razonador debe haber sido seleccionado y asociado a la ontología. Este método recibe como parámetro una cadena de caracteres que representa la consulta Sparql a validar. Poscondiciones: Devuelve verdadero en caso de que la consulta sea válida o falso en caso contrario.
2. En segundo lugar, hay que crear una instancia de la clase *InterfaceReasonerImplementation*, de forma que ésta se mantenga independiente al resto de módulos y clases de la aplicación. Esto se ha conseguido cargando dinámicamente dicha clase. Para ello existen dos posibilidades. Si la clase que ha sido redefinida es la ya existente *InterfazReasonerImplementation*, no será necesario hacer nada. Sin embargo, si el usuario ha creado una nueva clase que implementa la interfaz *InterfaceReasoner*, deberá adaptar la carga dinámica de clases a la suya. Para ello deberá modificar el archivo de configuración de la aplicación (*configuracin.properties*), asociando a la clave “*DRIVER*” de dicho archivo el valor de la ruta de su nueva clase.

La clase *InvalidOntologyException* ha sido creada para encapsular las excepciones producidas por el *framework* que ha sido seleccionado por defecto para desarrollar la aplicación, y no crear dependencias con él.

En la Figura 38 se representa, de forma resumida, las dependencias entre la clase encargada de ejecutar los tests (*OntologyTestCase*) con el razonador, que es la forma de comunicar la ejecución de los tests con la interfaz del razonador manteniendo la independencia con el driver seleccionado.

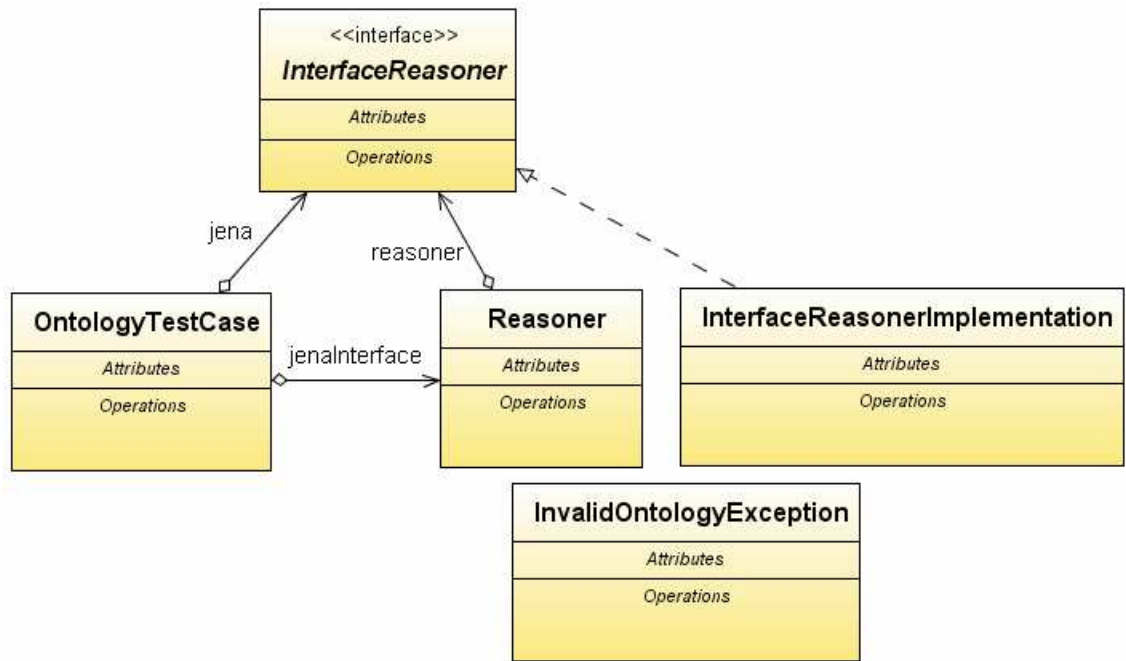


Figura 38 Detalle de la independencia del razonador

## 6.2 Persistencia de la Información

Dado que existe la posibilidad de guardar proyectos y abrirlos posteriormente para trabajar con ellos, es necesario almacenar la información relativa a una sesión de trabajo. Esta información es la contenida en la clase *CollectionTest* (Figura 28) y está formada por:

- La ubicación de la ontología y el *namespace*: estos datos se almacenan para saber con qué ontología se va a trabajar. Son datos imprescindibles por lo que en cada carga se comprueba que sean correctos y si no lo son, se le pide al usuario que los introduzca de nuevo, ya que, por ejemplo, la ubicación física de la ontología ha podido cambiar.
- Las instancias y los escenarios de tests: se almacenan para poder trabajar con ellos tantas veces como se quiera. Por un lado se almacenan las instancias creadas de forma independiente a los tests y por otro lado las asociadas a ellos.



Los resultados de la ejecución de los tests, sin embargo, no son almacenados, ya que ésta información no es relevante para el objetivo que persigue la aplicación. Lo importante es tener almacenado el test y ejecutarlo tantas veces como se quiera, no almacenar los resultados de cada ejecución que, al fin y al cabo, pueden obtenerse de nuevo tras cada ejecución.

El formato para el almacenamiento han sido ficheros de texto plano con extensión .xml. Se podrá guardar así una sesión de trabajo, en nuestro caso un objeto *CollectionTest*, en un archivo .xml para ser utilizado posteriormente por la herramienta.

La clase *IOManagerImplementation* es la encargada de llevar a cabo la persistencia y de agrupar los métodos necesarios para ello. Esta clase puede verse en la Figura 39.

<b>IOManagerImplementation</b> { From persistence }	
Attributes	
Operations	
<pre> public boolean loadProject( String ubicOnto, String namespaceOnto ) public CollectionTest loadProject( ) public void prepareProject( CollectionTest collection ) public boolean saveProject( boolean as, String carpetaProy, String nombreProy, String fichero ) public void saveInstanciasInMemory( Instancias instancias ) public boolean instanciasYaGuardadas( Instancias inst ) public boolean replaceInstanciasLocally( Instancias inst ) public void saveTestInMemory( ScenarioTest scenario ) public boolean testYaGuardado( ScenarioTest scen ) public boolean replaceScenarioLocally( ScenarioTest scen ) public boolean getComo( ) public void setComo( boolean como ) public String getCarpetaProy( ) public void setCarpetaProy( String carpetaProy ) public String getNombreProy( ) public void setNombreProy( String nombreProy ) public String getFichero( ) public void setFichero( String fichero )                     </pre>	

Figura 39 Detalle de la interfaz IOManagerImplementation

Las clases sobre las que trabaja la clase *IOManagerImplementation* son aquellas sobre las que se deberá guardar información, es decir, aquellas clases que serán necesarias para definir un proyecto. Estas clases son: *Instancias*, *ScenarioTest* y *CollectionTest*. Un esquema de éste hecho puede verse en la Figura 40.

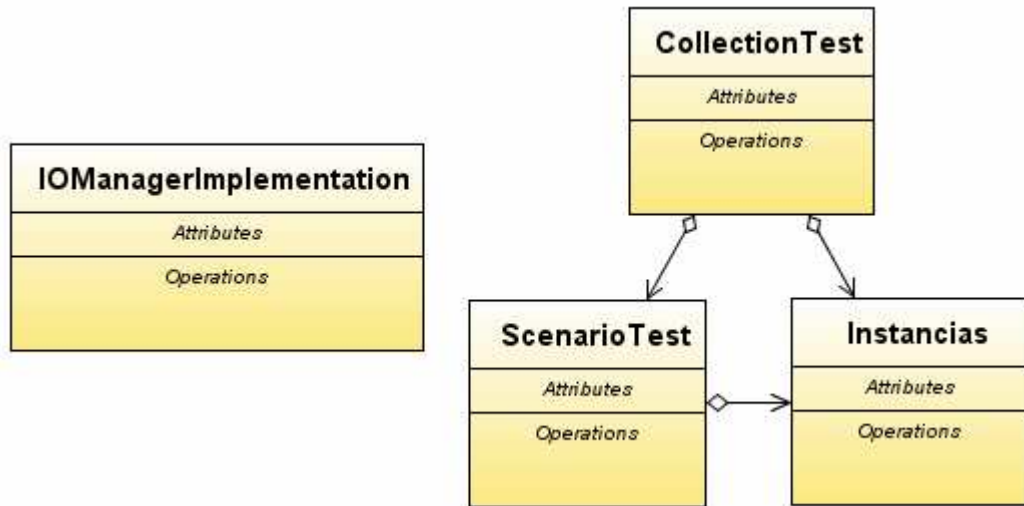


Figura 40 Diagrama de clases para la Persistencia

El proceso de persistencia se ejecutará en un hilo de ejecución diferente al de la interfaz gráfica, impidiendo que ésta no responda durante el proceso. El mecanismo es similar al seguido en la ejecución de los tests.

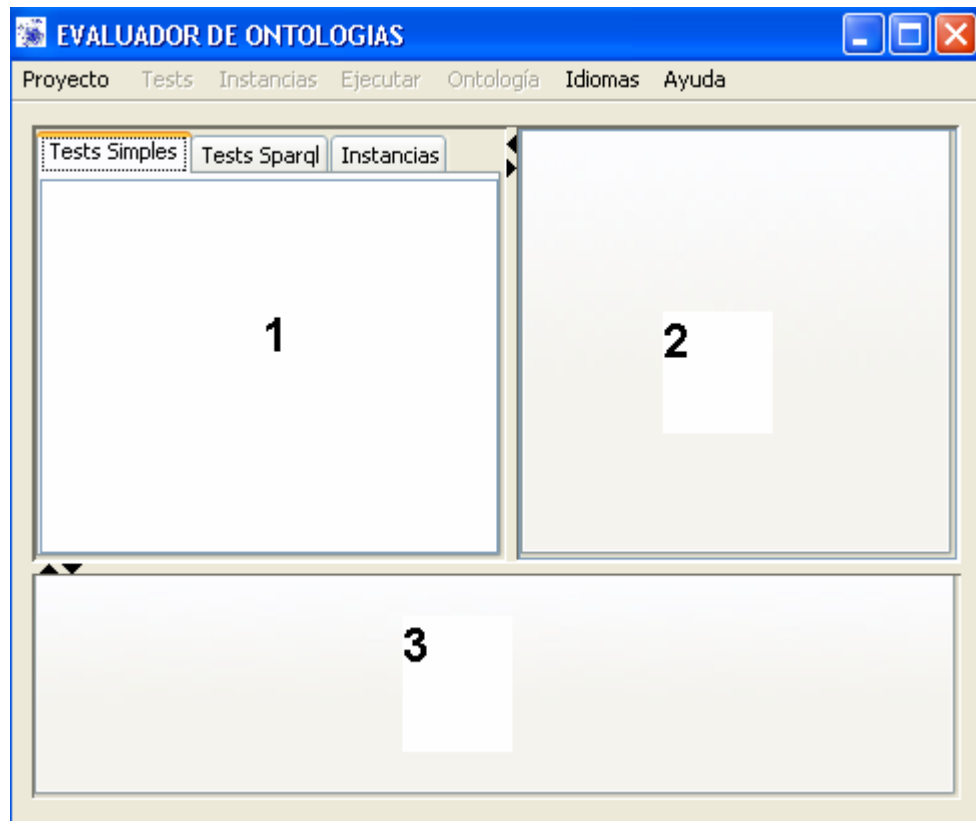
### 6.3 Interfaz gráfica de usuario

En esta sección se explicará el proceso de diseño de la interfaz gráfica de usuario, pensada para que sea lo más intuitiva y amigable posible al usuario.

La herramienta ha sido dividida en tres zonas de trabajo siguiendo el modo de desarrollo de *Netbeans*, para poder trabajar simultáneamente desarrollando tests, visualizando los resultados de las ejecuciones y teniendo acceso a los tests e instancias ya creados y almacenados. Estas zonas se describen a continuación:

1. Zona contenedora de los tests, instancias y ejecuciones guardadas por el usuario (zona marcada en rojo en la Figura 41).
2. Zona de trabajo donde el usuario creará/editará los tests (zona marcada en azul en la Figura 41).

3. Zona contenedora del resultado de la ejecución de los tests (zona marcada en morado en la Figura 41).



**Figura 41 Detalle de la interfaz principal**

En la zona identificada por el número 1, tal y como muestra la Figura 42, se irán almacenando en forma de lista y en sus respectivas pestañas los tests simples, las instancias y los test Sparql; también aparecerán los resultados de los tests una vez ejecutados, añadiendo por cada ejecución una nueva pestaña al panel.

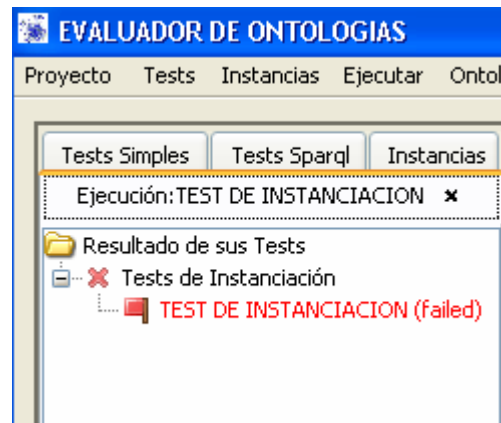


Figura 42 Detalle de la lista de Tests Simples

Un resumen de cómo se representan estos resultados puede verse en la Figura 43.

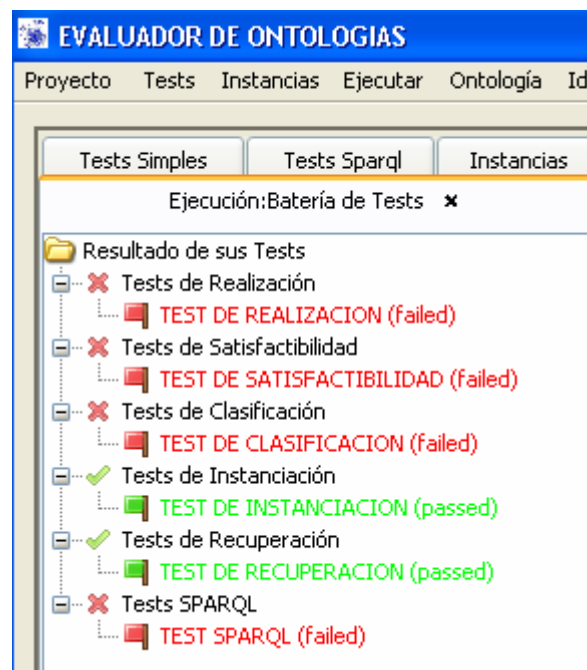
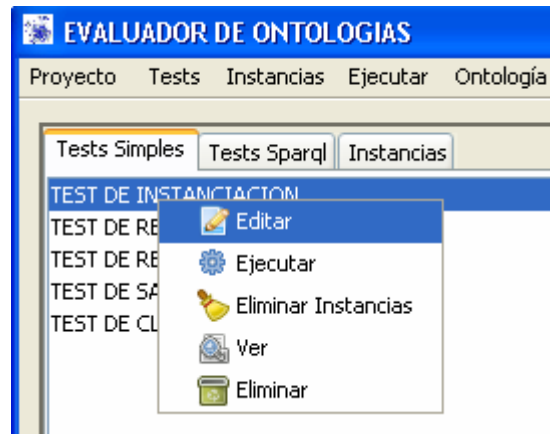


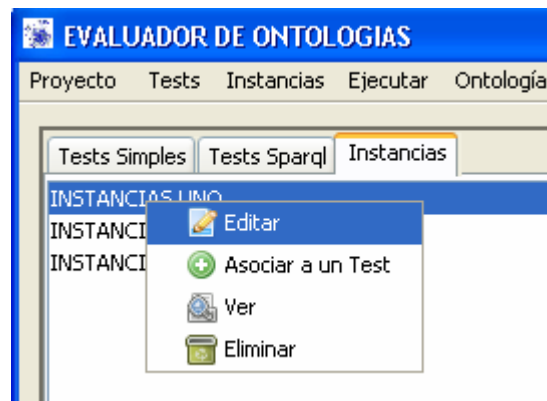
Figura 43 Detalle del resultado de la ejecución de los tests

Cada elemento de la lista de los Tests Simples, los Sparql y las Instancias, tendrá a su vez un menú accesible mediante clic derecho del ratón sobre el elemento. Este menú será distinto para los tests y para las instancias. Las opciones del test serán: editar, ejecutar, ver y eliminar; mientras que las opciones para las instancias serán: editar,

asociar a un test, ver y eliminar. En la Figura 44 y Figura 45 se muestra un detalle de la interfaz que muestra este comportamiento, tanto para los tests como para las instancias respectivamente.



**Figura 44** Detalle del submenú para los tests



**Figura 45** Detalle del submenú para las instancias

La zona identificada por el número 2 será la de trabajo del usuario. En ella se podrán crear tanto tests como instancias, guardarlos y ejecutarlos.

En esta parte de la interfaz gráfica de usuario, se ha utilizado el patrón de diseño *Observer*. Este patrón nos permite tener vistas distintas para el usuario, pero que

trabajan sobre un mismo modelo. En nuestro caso, hemos querido tener dos vistas de cara a la creación de los tests. Una de ellas orientada a un usuario novel, que le permita crear los tests a través de una interfaz guiada, con cajas de texto individuales para introducir las consultas, borrarlas, duplicarlas o añadirles un comentario (ver Figura 46); y otra orientada a un usuario experto, que prefiera introducir las consultas directamente en un cuadro de texto, así como los comentarios (ver Figura 47). El patrón *Observer* nos permite que, pese a que las vistas sean distintas de cara al usuario, el modelo sobre el que trabajan sea el mismo, en nuestro caso, el único con el que contamos para el desarrollo de la aplicación.

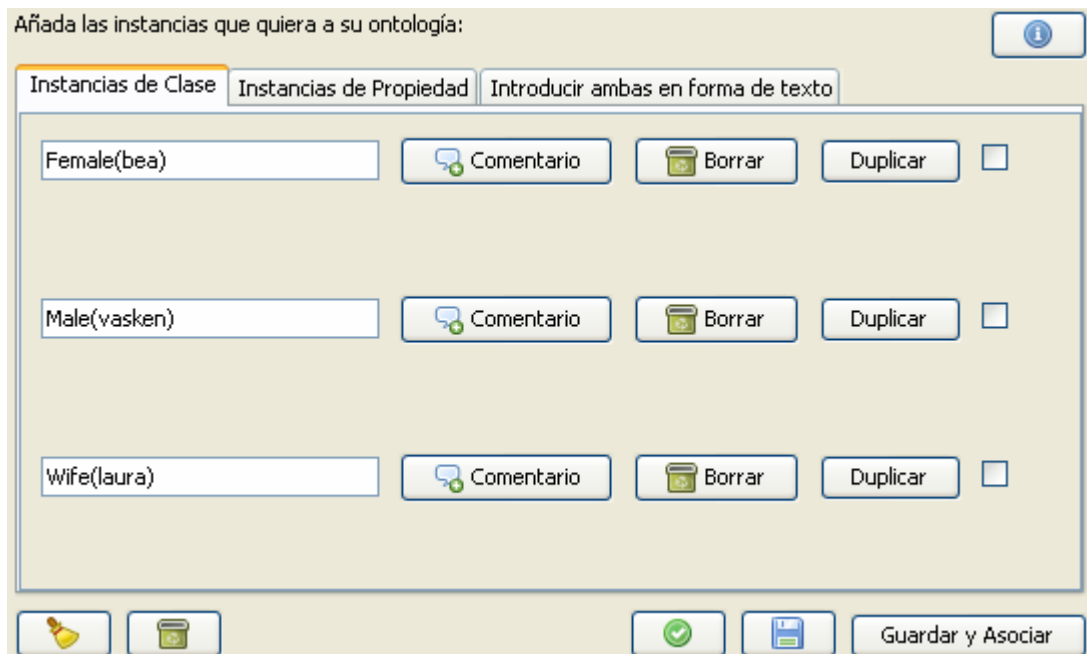
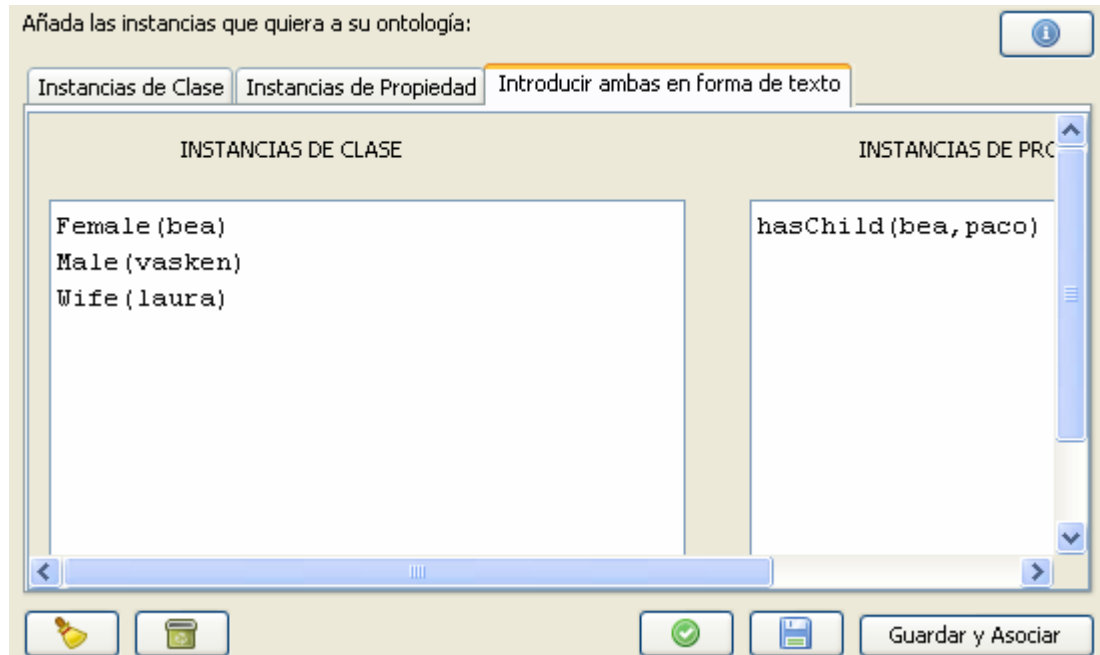


Figura 46 Detalle del proceso de creación de un test para un usuario novel



**Figura 47 Detalle del proceso de creación de un test para un usuario experto**

La zona identificada por el número 3 contendrá el resultado explicativo de la ejecución de los tests, es decir, explicará cuales son las consultas que han fallado, indicando cual era el resultado esperado por el usuario y cuál ha sido el obtenido, así como las consultas que han sido correctas. Por cada ejecución de un test o de un conjunto de tests, una pestaña contenedora de los resultados será añadida a este panel. Esta representación puede verse en la Figura 48.



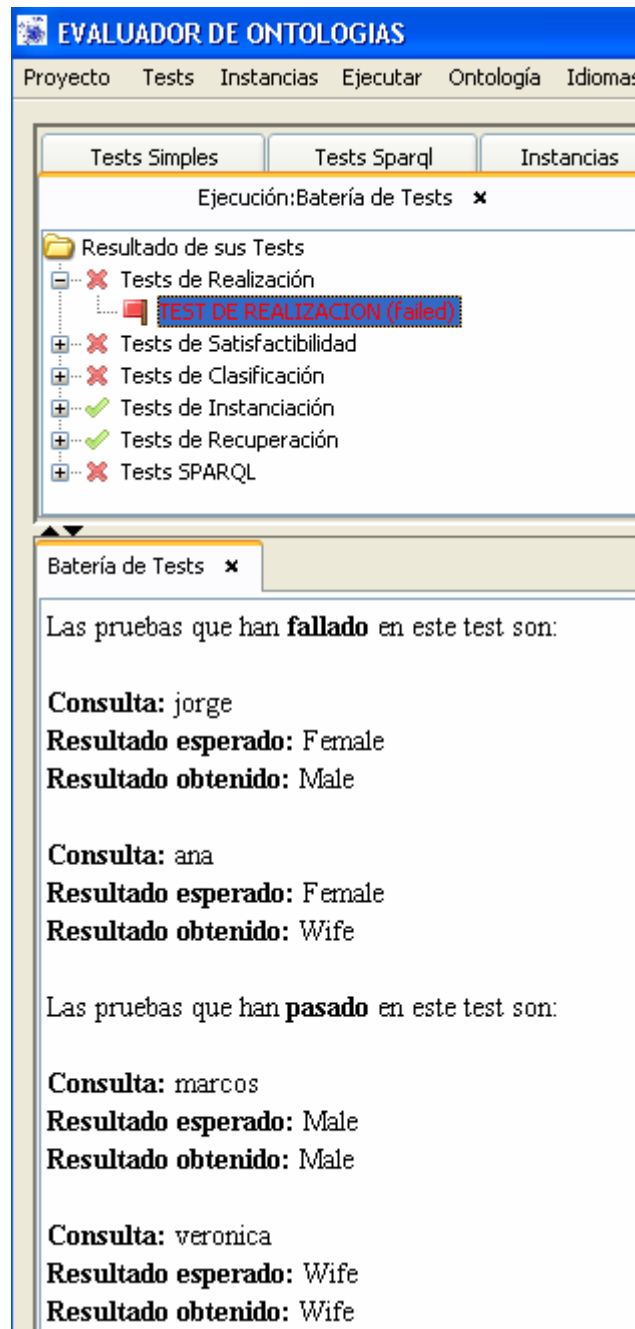


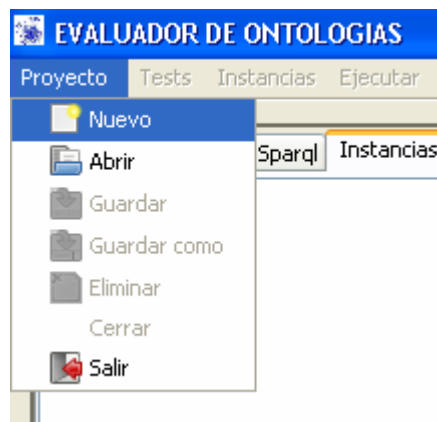
Figura 48 Detalle de la representación gráfica del resultado de la ejecución de los tests

### 6.3.1 Estructura de los menús

A continuación se detallará la estructura de menús diseñada para hacer el acceso a la funcionalidad lo más intuitivo posible.

Se han establecido cuatro menús principales:

- *Archivo*. Típico en todas las aplicaciones de escritorio. Nos permitirá crear un nuevo proyecto, cargar uno existente, guardar, guardar como y salir de la aplicación (Figura 49)



**Figura 49 Detalle de la opción Nuevo Proyecto**

- *Tests*. Mediante este menú podremos crear, importar, editar y ver tests. Se podrán crear dos tipos posibles de tests, simples (a su vez aquí se podrán crear cinco tipos de tests simples) o Sparql. Podrán editarse tests de la sesión actual de trabajo, así como importar tests (de proyectos ya existentes y realizados con la herramienta) y visualizarlos (Figura 50)

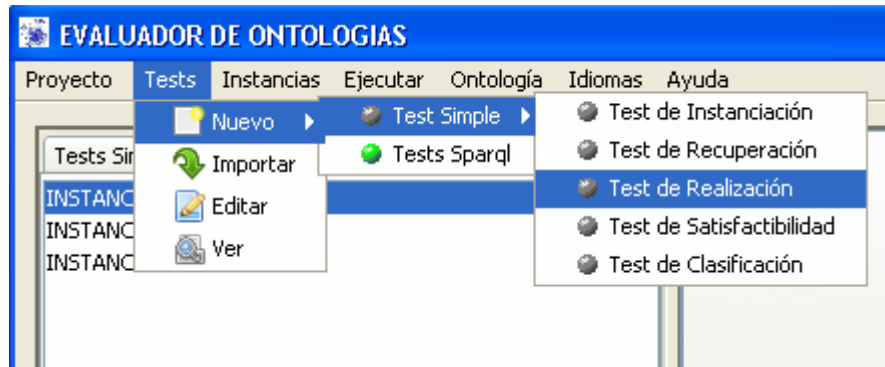


Figura 50 Detalle de la opción Crear Test

- *Instancias.* Mediante este menú se podrán crear instancias, importarlas (de proyectos ya existentes creados con esta herramienta), editarlas y visualizarlas (Figura 51)

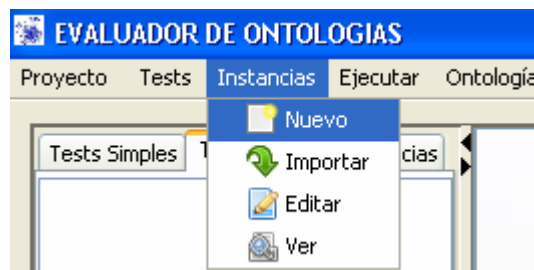


Figura 51 Detalle de la opciones para las instancias

- *Ejecutar.* Este menú nos permitirá ejecutar los tests. Se podrá elegir un conjunto de tests ha ejecutar o se podrá indicar directamente que se quieren ejecutar todos los test realizados y guardados hasta el momento (Figura 52)



**Figura 52** Detalle de las opciones para la Ejecución

## 7 Implementación

En este apartado se describirán aquellos puntos de la implementación que, por su relevancia o complejidad, resultan de especial interés. No se detallarán todas y cada una de las clases y métodos públicos sino que se explicarán los puntos concretos que han requerido de un mayor esfuerzo y dedicación. Durante el proceso de codificación se hizo un esfuerzo en documentar el código, por lo que el Javadoc generado automáticamente a partir del código fuente constituye una documentación más exhaustiva y detallada del código fuente.

### 7.1 Especificación del Framework y del Razonador

El *Framework Jena* utilizado en el desarrollo de la aplicación así como el razonador seleccionado para inferir y deducir conocimiento sobre las ontologías, son elementos que pueden cambiar en función de las necesidades. Para ello ha sido necesario implementar un mecanismo que permita que un usuario adapte el código a sus necesidades, añadiendo de forma sencilla el *framework* y el razonador que quiera utilizar. Para conseguir esto se han creado las clases *InterfaceReasoner*, *ReasonerImplementation* y *Reasoner*. La clase *InterfaceReasoner* es la que se deberá implementar en caso de querer utilizar otro *framework* u otro razonador. En nuestro caso dicha implementación está contenida en *ReasonerImplementation* que utiliza la API de *Jena* para realizar todas las operaciones necesarias sobre la ontología y que por lo tanto contiene los detalles del código más importantes y significativos que explicaremos a continuación. El diagrama de paquetes que refleja ésta explicación puede verse en la Figura 53.

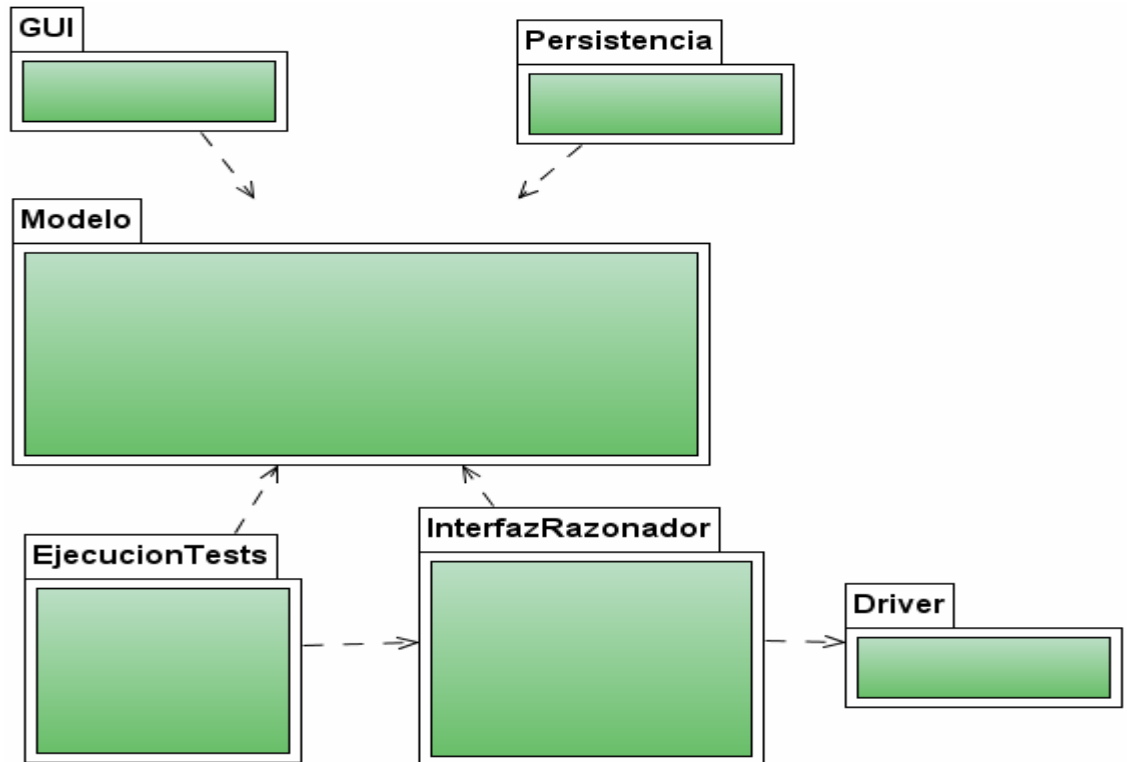


Figura 53 Diagrama de paquetes

### 7.1.1 Creación del modelo

El primer paso es crear el modelo sobre el que se va a trabajar, especificándole un razonador, en nuestro caso Pellet. Este modelo no se refiere al modelo de datos del que se habla en el diseño del software, en el capítulo anterior. Es un modelo específico para trabajar sobre la ontología con un razonador. Es el modelo que va a contener toda la definición de la ontología. La forma de especificar el razonador aparece definida en el Código Fuente 1. La forma de indicarle al modelo que el razonador es Pellet es mediante la línea 2, que es la forma establecida por el *framework Jena* para asociar el razonador Pellet al modelo ontológico sobre el que se va a trabajar.

```

1. private final OntModel model = ModelFactory.createOntologyModel(
2. PelletReasonerFactory.THE_SPEC );
  
```

Código Fuente 1. Creación del modelo asociado al razonador Pellet

Una vez creado el modelo hay que asociarlo con la ontología, tal y como muestra el Código Fuente 2, mediante el método `addReasoner`, al que se le pasa como parámetro la ontología, y mediante los métodos de la API de *Jena* `read`, prepare y

validate ésta es asociada la modelo, preparada y validada respectivamente. Con esto nos aseguramos que la ontología es válida.

```

1. public void addReasoner(String ontologia) throws InvalidOntologyException{
2. ontologia = "file:".concat(ontologia);
3. try {
4. model.read(ontologia);
5. model.prepare();
6. model.validate();
7. }catch(Exception e){
8. throw new InvalidOntologyException();
9. }}

```

Código Fuente 2. Asociación del razonador a la ontología

#### 7.1.1.1 Cargar Instancias

Por carga de instancias se entiende el proceso de añadir instancias a la ontología sobre la que se va a trabajar, es decir, crear o ampliar la base de hechos de la ontología. En el Código Fuente 3 se ve el detalle de implementación del método que permite añadir instancias de clase a la ontología, *addInstanceClass*. A este método se le pasa como argumento el namespace de la ontología, el nombre de la clase y el individuo a añadir a dicha clase.

En la línea 2 se listan, a través del modelo creado con anterioridad, todas las clases que contiene la ontología, para así poder comprobar que la clase a la que se quiere asociar el individuo existe. En la línea 6, se utiliza el símbolo “#” para separar las clases del *namespace*, ya que el formato obtenido en la línea 2 al listar las clases es: namespace#clase. En la línea 12 se crea una nueva instancia para la clase a la cual se quiere añadir un nuevo individuo y en la línea 13 se crea el individuo para esa clase.

En el caso de las instancias de propiedad, el proceso será el mismo a diferencia de los métodos *listNamedClasse* y *createClass*, que serán sustituidos respectivamente por *listObjectProperties* y *createProperty*.

```

1 public boolean addInstanceClass(String ns, String nameClass, String value) {
2 Iterator it = model.listNamedClasses();
3 String[] clas = null;
4 ArrayList<String> clases = new ArrayList<String>();
5 while (it.hasNext()) {

```

```
6 clas = it.next().toString().split("#");
7 for (int i=0; i<clas.length; i++) {
8     clases.add(clas[i]);
9 }
10 if (clases.contains(nameClass)) {
11     nameclass = model.createClass(ns + nameClass);
12     model.createIndividual(ns + value,nameclass);
13     return true;
14 }else{
15     return false;
16 }
17 }
```

Código Fuente 3. Clase para añadir instancias de clase a la ontología

#### 7.1.1.2 Eliminar la Base de Hechos de la Ontología

Una vez añadida una base de hechos para un test y ejecutado este, es necesario eliminar de la ontología dicha base de hechos para añadir la del siguiente test. Este proceso se realiza mediante el método de Código Fuente 4. En la línea 2 de Código Fuente 4 se eliminan todos los recursos asociados al modelo.

```
1. public void deleteEntries() {
2.     model.removeAll();
3. }
```

Código Fuente 4. Reseteo del modelo de la ontología

#### 7.1.2 Descripción de los Tests

Cada test será el encargado de obtener, dependiendo del tipo de test, una serie de datos sobre la ontología. Estos datos serán comparados posteriormente, en otra clase, con los introducidos por el usuario en el campo “Resultado Esperado”, para así poder comprobar si la consulta es o no correcta (si ambos resultados coinciden).

##### 7.1.2.1 Test de Instanciación

Este test es el encargado de, dado un individuo y una clase, deducir si éste pertenece o no a la dicha clase. Para ello se obtiene en primer lugar la clase del modelo de la ontología de la que se quiere conocer el individuo, como se ve en la línea 3 del Código Fuente 5. Si ésta no es vacía, entonces se listan todas sus instancias, como se ve en la línea 7, y se van recorriendo. Si el nombre de alguna de estas instancias coincide con el pasado como argumento, se devuelve *true*, en caso contrario *false*.



```
1. public String instantiation(String ns, String className,
2. String individualName) throws InvalidOntologyException{
3. OntClass ontClass = model.getOntClass(ns + className);
4. if (ontClass==null) return "La clase introducida no es una instancia para el modelo";
6. try{
7. Iterator it = ontClass.listInstances();
8. while(it.hasNext()){
9. String instanceName = it.next().toString();
10. instanceName = instanceName.substring(instanceName.indexOf("#")+1);
11. if(individualName.equals(instanceName)) {
12. return "true";
13. }.}
14. }catch(InconsistentOntologyException in){
15. throw new InvalidOntologyException();}
16. return "false";}
```

Código Fuente 5. Test de instanciación

### 7.1.2.2 Test de Recuperación

Este test deduce, dada una clase, todos los individuos que pertenecen a la misma. Para ello se ha creado el método *retrieval* que se ve en Código Fuente 6. Para ello primero se obtiene la clase del modelo de la cual se quieren listar los individuos, tal y como muestra la línea 3 del Código Fuente 6. Después, si ésta existe, se obtienen todos sus individuos (línea 6). Éstos se van recorriendo y se van añadiendo a un array, que será devuelto por el método y que contendrá todos los individuos pertenecientes a la clase.

```
1. public ArrayList<String> retieval(String ns, String className){
2. ArrayList<String> rval = new ArrayList<String>();
3. OntClass ontClass = model.getOntClass(ns + className);
4. if(ontClass==null) return rval=null;
5. try{
6. Iterator it = ontClass.listInstances();
7. while(it.hasNext()){
8. String instanceName=it.next().toString();
9. instanceName=instanceName.substring(instanceName.indexOf("#")+1);
10. rval.add(instanceName);
11. return rval;
12. }catch(InconsistentOntologyException in){
13. throw new InvalidOntologyException(); }}
```

Código Fuente 6. Test de recuperación

### 7.1.2.3 Test de Realización

Este test es el encargado de deducir, dado un individuo, cual es la clase más exacta a la que pertenece. El código que realiza esta acción está descrito en Código Fuente 7. Para ello, en primer lugar se obtiene el individuo concreto del modelo del cual

se quiere saber su clase más exacta (línea 2). Si éste no es nulo, se utiliza el método *getRDFType()* para obtener su recurso más específico (línea 6) y lo manipularemos para quedarnos en concreto con la clase (línea 7), que será lo que devolverá el método.

```

1. public String realization(String ns, String individualName){
2. Individual individual = model.getIndividual(ns + individualName);
3. if(individual==null) return "El individuo introducido no es una instancia para el
4. modelo";
5. try{
6. Resource resource = individual.getRDFType(true);
7. String className=resource.toString().substring(resource.toString().indexOf("#")+1);
8. return className;
9. }catch(InconsistentOntologyException in){
10. throw new InvalidOntologyException();} }

```

Código Fuente 7. Test de realización

#### 7.1.2.4 Test de Clasificación

Este test es el encargado de, dado un individuo, obtener todas las clases a las que pertenece. En primer lugar se obtiene el individuo del cual queremos obtener todas sus clases (línea 3 de Código Fuente 8). En la línea 7 obtenemos un iterador para recorrer todas las clases de la ontología. Tras realizar una serie de operaciones para eliminar redundancias, llamamos al método *instantiation* (línea 22) para saber si el individuo pertenece a la clase. Si pertenece, se añade a una lista que devolverá el método (línea 24).

```
1. public ArrayList<String> classification(String ns, String individuo)
2. throws InvalidOntologyException{
3. Individual individual = model.getIndividual(ns + individuo);
4. String pertenece;
5. ArrayList<String> clases = new ArrayList<String>();
6. if(individual==null) return clases = null;
7. Iterator it = model.listNamedClasses();
8. Iterator itaux = model.listObjectProperties();
9. ArrayList<String[]> arrayProp = new ArrayList<String[]>();
10. while(itaux.hasNext()){
11. arrayProp.add(itaux.next().toString().split("#"));
13. while(it.hasNext()){
14. int aux=0;
15. String[] instanceName = it.next().toString().split("#");
16. for(int i=0; i<arrayProp.size(); i++){
17. String[] deProp = arrayProp.get(i);
18. if(deProp[0].equals(instanceName[0]) && deProp[1].equals(instanceName[1])){
19. aux=1;
20. }}
21. if(aux==0){
22. pertenece = instantiation(ns, instanceName[1], individuo);
23. if (pertenece.equals("true")) {
24. clases.add(instanceName[1]);
25. }}}
26. return clases;}
```

Código Fuente 8. Test de clasificación

#### 7.1.2.5 Test de Satisfactibilidad

Este test comprueba si una nueva instancia de clase puede ser añadida a la base de hechos de la ontología sin que ésta pase a ser inconsistente. En primer lugar, se obtiene la clase concreta del modelo a la cual queremos saber si se le puede añadir un individuo (línea 3 de Código Fuente 9). Después se listan las clases disjuntas para esa clase (se listan todas aquellas clases con los que no puede tener ningún individuo en común, línea 6) y se añaden a una lista (línea 11). Después se utiliza el método *classification*, explicado con anterioridad, para obtener todas las clases a las que pertenece el individuo (línea 13). Tras esto, se van recorriendo los arrays de clases disjuntas y el de individuos dados por el método *classification*. Si el individuo pertenece a una clase declarada como disjuntas, el método devuelve la cadena de caracteres “false”, en caso contrario “true”.

```

1. public String satisfactibility(String ns, String concepto, String clase)
2. throws InvalidOntologyException{
3. OntClass ontClass = model.getOntClass(ns+clase);
4. if(ontClass==null) return "La clase introducida no es una instancia para el modelo";
5. try{
6. Iterator it = ontClass.listDisjointWith();
7. ArrayList<String> conjuntoDisj = new ArrayList<String>();
8. while(it.hasNext()){
9. String disjunta = it.next().toString();
10. disjunta = disjunta.substring(disjunta.indexOf("#")+1);
11. conjuntoDisj.add(disjunta);
12. }
13. ArrayList<String> clasesConcepto = classification(ns,concepto);
14. if(clasesConcepto==null){
15. return "true";
16. }else{
17. for(int i=0;i<clasesConcepto.size();i++){
18. for(int k=0;k<conjuntoDisj.size();k++){
19. if(clasesConcepto.contains(conjuntoDisj.get(k)))
20. return "false"; } }
21. return "true";
22. }catch(InconsistentOntologyException in){
23. throw new InvalidOntologyException(); } }

```

Código Fuente 9. Test de satisfactibilidad

#### 7.1.2.6 Test Sparql

Con el test Sparql lo que se pretende es evaluar una consulta Sparql, tratando los resultados obtenidos de la misma para poder compararlos posteriormente con los esperados por el usuario y así poder emitir un informe. Para poder tratar la consulta Sparql es necesario que esta sea válida, es decir, correcta gramaticalmente y sea de tipo SELECT. Para realizar esta validación se ha creado el método *validarSparqlQuery* (Código Fuente 10).

Este método crea la consulta, como se ve en la línea 2. Si la consulta no es válida, se produce una excepción que es capturada en el lugar que corresponda. Si no es de tipo SELECT (línea 3), devuelve false indicando que no es válida.

```

1. public boolean validarSparqlQuery(String query){
2. Query queryStr = QueryFactory.create(query);
3. if(!queryStr.isSelectType()){
4. return false;
5. }return true; }

```

Código Fuente 10. Validación de consulta Sparql

En la línea 2 de Código Fuente 10 se crea un objeto de tipo *Query* que será la consulta en forma de cadena de caracteres introducida por el usuario convertida en una consulta de tipo *Query* para poder trabajar con ella a través de *Jena*. En la línea 15 se crea una instancia de la *Query* en el modelo con el razonador de Pellet y en la línea 16 se ejecuta la consulta de tipo SELECT. Los resultados se obtienen en una lista (línea 16), cuyo contenido será tratado y almacenado de una forma concreta en un objeto de la clase *ExecQuerySparql*, clase encargada de tratar los resultados obtenidos de la ejecución de las consultas Sparql. La clase *ExecQuerySparql* está formada por una lista de Strings y un nombre. El nombre indica el SELECT realizado por el usuario y la lista contiene todos los resultados obtenidos para ese SELECT; luego los resultados obtenidos de la ejecución de la consulta se van almacenando de esta forma en el objeto de tipo *ExecQuerySparql*. El método devuelve la lista de tipo *ExecQuerySparql*.

```

1. public ArrayList<ExecQuerySparql> testSPARQL(String queryStr, boolean formatHTML){
2. Query query=null;
3. String expReg = "([\\?]{1}[a-zA-Z]+)";
4. ArrayList<String> sel = new ArrayList<String>();
5. try{
6. query = QueryFactory.create(queryStr);
7. Element patern = query.getQueryPattern();
8. String p = patern.toString();
9. String[] consulta = p.split("\\s");
10. for(int i=0; i<consulta.length;i++){
11. if(consulta[i].matches(expReg)){
12. sel.add(consulta[i].substring(1));
13. } }
14. try{
15. QueryExecution qexec = new PelletQueryExecution(query, model);
16. ResultSet results = qexec.execSelect();
17. List resultVars = query.getResultVars();
18. ArrayList<ExecQuerySparql> lista = new ArrayList<ExecQuerySparql>();
19. while(results.hasNext()){
20. QuerySolution binding = results.nextSolution();
21. for(int i = 0; i < resultVars.size(); i++) {
22. ExecQuerySparql e = new ExecQuerySparql();
23. String var = (String) resultVars.get(i);
24. RDFNode result = binding.get(var);
25. if(result!=null){
26. String aux = result.toString();
27. String dato = aux.substring(aux.indexOf("#")+1);
28. if(this.perteneceALista(var,lista)==false){
29. e.setNombreSelect(var);
30. lista.add(e);
31. lista.get(i).getDatos().add(dato);
32. }else{
33. ExecQuerySparql eq = this.seleccionarLista(var, lista);
34. eq.getDatos().add(dato);
35. } } }
36. return lista;
37. }catch(InconsistentOntologyException in){
38. throw new InvalidOntologyException();
39. }catch (QueryParseException ex){
40. throw new InvalidOntologyException();
41. } }

```

Código Fuente 11. Test Sparql

### 7.1.3 Ejecución de los Tests

La ejecución de los tests se realiza en un hilo de ejecución distinto al de la interfaz de usuario, para evitar ésta deje de responder durante un proceso que puede llegar a tardar varios minutos. Por ello se creó la clase *ExecuteTest* que extiende a la clase *SwingWorker* y que permite crear el hilo independiente de ejecución. La clase *SwingWorker* es una utilidad integrada en Java SE 6 que facilita enormemente la gestión de estos hilos de ejecución. La clase *ExecuteTest* es la encargada de gestionar el hilo

independiente en el que se ejecutarán los tests. Para ello, se crea un nuevo objeto de dicha clase (ver línea 3 del Código Fuente 12) en la clase que iniciará el proceso de ejecución del test. El objeto creado llamará al método *execute()* de la clase *SwingWorker*. Con la línea 4 del Código Fuente 12, asociamos el hilo a una ventana de progreso con un botón “Cancelar”, para permitir que el usuario pueda cancelar la ejecución de los tests en cualquier momento.

```

1  try{
2    TreeResults.setTestSeleccionado(scenario.getNombre());
3    ExecuteTest execTest = new ExecuteTest(scenario);
4    ProgressControlJDialog progres = new ProgressControlJDialog(execTest);
5    JProgressBar progresBar = progres.getProgressBar();
6    progresBar.setValue(0);
7    execTest.addPropertyChangeListener(new ProgressListener(progresBar,progres,true));
8    execTest.execute();
9    progres.setVisible(true);
10 }catch (InvalidOntologyException ex){
11   JOptionPane.showMessageDialog(MainApplicationJFrame.getInstance(),"No se pudo ejecutar el
12   test. Ontología no válida.",
13   "Error Message",JOptionPane.ERROR_MESSAGE);
14 }
```

Código Fuente 12. Inicio del proceso de ejecución

La llamada al método *execute()* invocará al método *doInBackground()* de la clase *ExecuteTest* (ver Código Fuente 13) en un hilo de ejecución diferente al de la interfaz de usuario. Este método lanzará la ejecución de los tests llamando al método *execBaterlyTest(ListaScenariotest)* (ver Código Fuente 13).

```

1  @Override
2  protected OntologyTestResult doInBackground() throws Exception {
3    OntologyTestResult treeResult = new OntologyTestResult();
4    setProgress(0);
5    if(scenario!=null){
6      treeResult = execOneTest(scenario);
7      setName(scenario.getNombre());
8    }else if(listScenariotest!=null){
9      treeResult = execBaterlyTest(listScenariotest);
10     setName("Batería de Tests");
11   }
12   setProgress(100);
13   return treeResult;
14 }
```

Código Fuente 13. Método *doInBackground()* de la clase *ExecuteTest*

Mientras la ejecución de los tests no sea cancelada por el usuario (ver línea 7 del Código Fuente 14), para cada escenario contenido en la colección (ver línea 6 del

Código Fuente 14), se realizará una llamada al método *run()* de la clase *OntologyTestCase*, que será la encargada de ejecutar los tests (ver línea 8 del Código Fuente 14).

```

1. private OntologyTestResult execBatoryTest(List<ScenarioTest> listScenario){
2.   OntologyTestCase testCase = new OntologyTestCase();
3.   testResult = new OntologyTestResult();
4.   int size = listScenario.size();
5.   int div = 100/size;
6.   for(int i=0;i<size;i++){
7.     if(this.isCancelled()==false){
8.       testCase.run(testResult, CollectionTest.getInstance(), listScenario.get(i));
9.       setProgress(getProgress()+div);
10.    }
11.   return testResult;}

```

Código Fuente 14. Ejecución de los tests

El método *run()* que se ve en la línea 8 de Código Fuente 14 realiza las llamadas necesarias para inicializar los datos del test, ejecutarlo y dejarlo listo para la siguiente ejecución. Estas llamadas se muestran en el Código Fuente 15.

```

1   InterfaceReasoner j = setUpOntology(scenariotest, ont, ns);
2   testInst.run(testresult, ns, ont, scenariotest,j);
3   testRet.run(testresult, ont, ns, scenariotest,j);
4   testRealiz.run(testresult, ont, ns, scenariotest,j);
5   testSatis.run(testresult, ont, ns, scenariotest,j);
6   testClas.run(testresult, ont, ns, scenariotest,j);
7   testSparql.run(testresult, ont, ns, scenariotest,j);
8   tearDownOntology(j);

```

Código Fuente 15. Algoritmo secuencial para la ejecución de los tests

La línea 1 del Código Fuente 15 inicializa el test, es decir, asocia a la ontología las instancias del test. Tal y como se ve en las líneas 5 y 6 del Código Fuente 16, se crean dos patrones gramaticales para poder tratar las instancias introducidas por el usuario. En las líneas 7 y 8 se obtiene una instancia del razonador al que se le añade la ontología en la línea 10. Una vez realizados estos pasos, se van recorriendo por separado las instancias de clase asociadas al test (línea 16) y las de propiedad (línea 23) y se añaden a la ontología (línea 28) de una en una.



```

1  protected InterfaceReasoner void setUpOntology(ScenarioTest st, String ont, String ns) throws
   InvalidOntologyException{
2  ListIterator liClass;
3  ListIterator liProperties;
4  String[] ciClas,ciInd,piClas,piInd;
5  patron1 = "[\\(|,\\|n| ]";
6  patron2 = "[\\n| \\|) ]";
7  jenaInterface = new Reasoner();
8  jena = jenaInterface.getReasoner();
9  if(JenaInterface.isCargado()==true){
10 Jena.addReasoner(ont);
11 Instancias instancias = st.getInstancias();
12 List<ClassInstances> classInstances = instancias.getClassInstances();
13 List<PropertyInstances> propertyInstances = instancias.getPropertyInstances();
14 liClass = classInstances.listIterator();
15 liProperties = propertyInstances.listIterator();
16 while (liClass.hasNext()) {
17 ClassInstances cla = (ClassInstances) liClass.next();
18 String ci = cla.getClassInstance();
19 ciClas = ci.split(patron1);
20 ciInd = ciClas[1].split(patron2);
21 jena.addInstanceClass(ns, ciClas[0], ciInd[0]);
22 }
23 while (liProperties.hasNext()) {
24 PropertyInstances p = (PropertyInstances) liProperties.next();
25 String pi = p.getPropertyInstance();
26 piClas = pi.split(patron1);
27 piInd = piClas[1].split(patron2);
28 jena.addInstanceProperty(ns, piClas[0], piInd[0]); } }
29 return jena;

```

Código Fuente 16. Preparación del Test (método *setUpOntology*)

La líneas de la 2 a la 7 del Código Fuente 15 ejecutan el test. Para ello, se realiza la llamada correspondiente al test que se tenga que ejecutar. En el caso de ser el test de instanciación, se realizará la llamada de la línea 1 del Código Fuente 17. Se le pasa el namespace de la ontología, la clase y el individuo. Si se tratase del test de recuperación, se realiza la llamada de la línea 2, a la que se le pasa el namespace y la clase de la que se quiere conocer los individuos pertenecientes, y devuelve la lista con todos los individuos de dicha clase. Si es el test de realización, se realiza la llamada de la línea 3 del Código Fuente 17, y se le pasa el namespace de la ontología y el individuo del que se quiere conocer la clase más exacta, y se devuelve dicha clase. Si se tratase del test de satisfactibilidad, se llamaría a la línea 4 y se le pasaría como argumento al test el namespace de la ontología, el nuevo individuo a añadir y la clase. Si fuese el test de clasificación, se realizaría la llamada de la línea 5 y se le pasaría al test el namespace de la ontología y el individuo del cual se quieren conocer todas las clases a las que pertenece, por lo que devuelve una lista de Strings que contienen dichas clases. Si fuese

el test Sparql, se le pasa la consulta realizada (línea 6) y se devuelve una lista de *ExecQuerySparql*.

```

1 String resObtenidoInst = Jena.instantiation(ns, clasF, indF)
2 List<String> resObtenidoRet = Jena.retrieval(ns, query);
3 String resObtenidoRealiz = Jena.realization(ns, query);
4 String resObtenidoSatisf = Jena.satisfiability(ns, concepto, loincluye);
5 List<String> resObtenidoClas = Jena.classification(ns, query);
6 List<ExecQuerySparql> listaResultObtenida = Jena.testSPARQL(sparqlQuery, true);

```

Código Fuente 17. Llamadas a la ejecución de cada test. Método *runOntologyTest*.

Todos los tests devuelven una serie de datos que serán comparados por los que el usuario indicó que esperaba obtener para así saber si el test ha pasado o ha fallado. Los resultados se irán añadiendo al objeto *testResult*. Si el resultado coincide, se añadirá a la lista de resultados correctos, es decir, se añadirá un *addOntologyPassedQuery* mientras que si es distinto, se añadirá a la de resultados fallados, es decir, a la *addOntologyFailureQuery*. Si se tratase de una consulta Sparql, se añadirían respectivamente a *addOntologyPassedSparql* y a *addOntologyFailureSparql*. Un ejemplo para el test de Instanciación puede verse en el Código Fuente 18.

```

1 resObtenidoInst = Jena.instantiation(ns, clasF, indF);
2 if(!resObtenidoInst.equals(resQueryExpected)){
3 testresult.addOntologyFailureQuery(nombreTestUsuario, qo, resObtenidoInst, tip);
4 }else{
5 testresult.addOntologyPassedQuery(nombreTestUsuario, qo, resObtenidoInst, tip);}

```

Código Fuente 18. Detalle del proceso de captura de resultados en la clase *OntologyTestCase*

A cada lista de resultados se le pasa como argumento el nombre del test con el que se está trabajando, la consulta realizada al test, el resultado obtenido tras la ejecución del test y el tipo de test.

La línea 3 del Código Fuente 15 elimina las instancias de la ontología mediante un método de la API de *Jena* (Código Fuente 19).

```

1 protected void tearDownOntology(InterfaceReasoner Jena){
2 jena.deleteEntries();}

```

Código Fuente 19. Método *tearDownOntology*

## 7.2 Persistencia de la Información

La persistencia se ha realizado mediante las clases *XMLEncoder* y *XMLDecoder* de Java. Éstas permiten convertir un objeto bean o una colección de objetos beans a un archivo xml y viceversa, respectivamente.

```
1  if(!fichero.endsWith(".xml")){
2    XMLEncoder e = new XMLEncoder(new BufferedOutputStream(new
    FileOutputStream(fichero+".xml")));
3  }else{
4    XMLEncoder e = new XMLEncoder(new BufferedOutputStream(new FileOutputStream(fichero)));
5  }
6  e.writeObject(CollectionTest.getInstance());
7  e.close();
```

Código Fuente 20. Detalle del método *saveProject*

En la línea 2 del Código Fuente 20 se crea una instancia de la clase *XMLEncoder* y en la línea 6 se escribe sobre ella el objeto que se quiere persistir, en este caso, el proyecto en su totalidad, el objeto *CollectionTest*

El proceso de abrir un proyecto ya creado, viene dado en el Código Fuente 21 y utiliza la clase *XMLEncoder* para extraer el contenido de un documento xml (formato en el que quedó guardado el proyecto) en un objeto de tipo *CollectionTest* y devolverlo.

```
1  decoder = new XMLDecoder(new BufferedInputStream(new
    FileInputStream(FileChooserSelector.getPathSelected())));
2  return (CollectionTest) decoder.readObject();
```

Código Fuente 21. Detalle del método *loadProject*

Ambos procesos, el de guardar o abrir un proyecto, también han sido realizados en un hilo de ejecución distinto al de la interfaz de usuario, utilizando, al igual que para la ejecución de los tests, la clase *SwingWorker*.

## 7.3 Gestión del Razonador

Una de las cuestiones más importantes de éste proyecto es qué razonador se va a emplear para poder inferir y deducir el conocimiento necesario sobre la ontología que nos permita realizar los tests y cómo se va a trabajar con razonador en Java, para poder implementar la herramienta. Se ha elegido el razonador *Pellet* y se ha realizado la implementación sobre el *framework* de *Jena*.

Esta decisión ha sido tomada por varios motivos. En el caso de *Jena*, porque es un *framework* en Java de licencia libre, que provee todos los elementos básicos que se necesitan y que incluye muchas APIs para poder trabajar en aplicaciones semánticas sobre Java, como en el caso de este proyecto. En el caso de *Pellet*, porque cumple con todas las condiciones especificadas en el caso del razonador (aquí pondría una referencia al apartado donde se dieron estas razones). Sin embargo, el proyecto ha sido planteado para que tanto *Jena* como *Pellet* puedan ser cambiados en función de las necesidades del usuario.

Esto se ha logrado creando una interfaz para el razonador que sea independiente del resto de la aplicación. Dicha interfaz es implementada por la clase *ReasonerImplementation*, tal y como se ven en la Figura 54, y en ella es dónde aparecen las dependencias con las librerías del *framework* de *Jena* y el razonador *Pellet*.

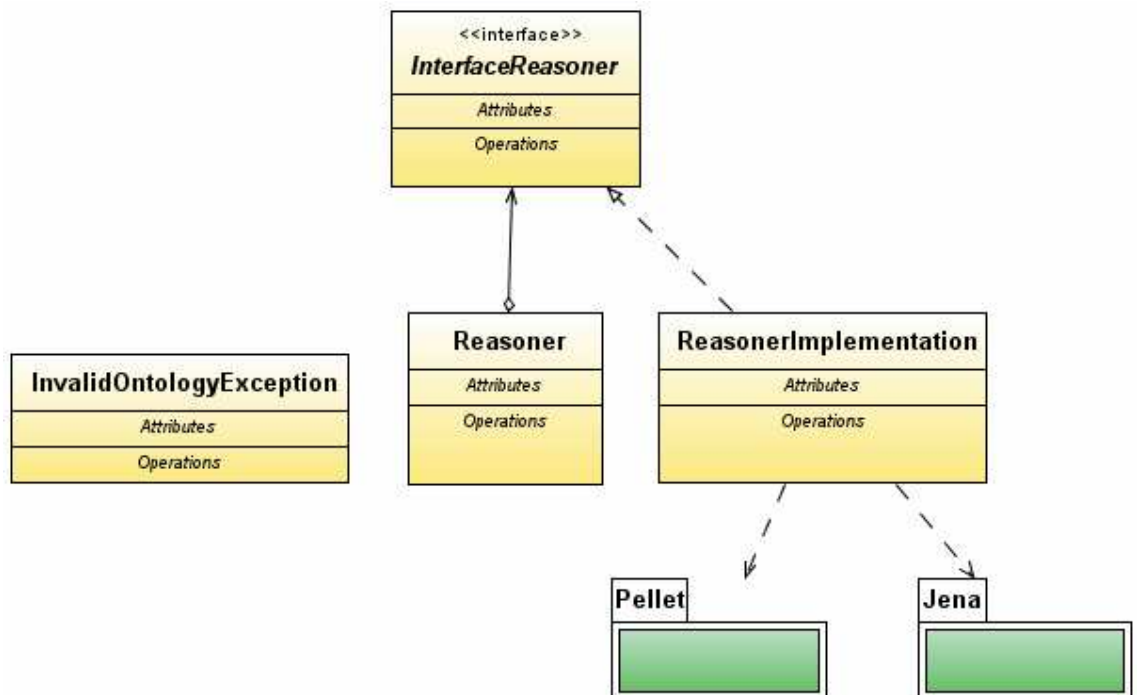


Figura 54 Detalle de las clases encargadas del Razonador

Para que éstas dependencias no afecten al resto de la implementación, se ha utilizado la carga dinámica de clases en la clase *Reasoner* (Código Fuente 22), que es la

encargada de obtener una instancia de la clase *InterfaceReasoner*, mediante el método *getReasoner()*, tal y como se ve en la Figura 55

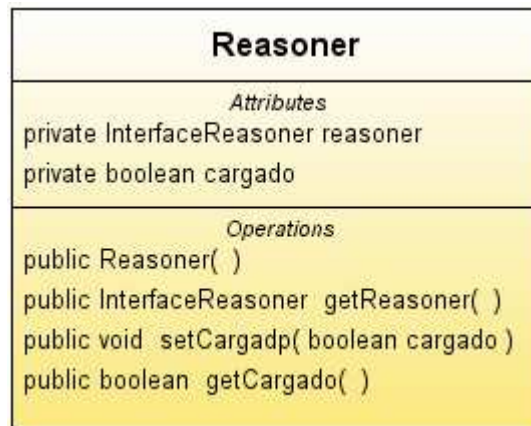


Figura 55 Detalle de la clase Reasoner

El método *getReasoner()* es el encargado de obtener una instancia del razonador seleccionado sin crear dependencias. Para ello utiliza la clase *Configuration* y su método *cargarDriver()*. Como se puede ver en la línea 2 del Código Fuente 22, el driver se obtiene a partir de un objeto de tipo *Properties*, el cual creará una nueva instancia de la clase *InterfaceReasoner()* empleando el driver indicado en el archivo de configuración, tal y como se ve en la línea 4 del Código Fuente 22.

```

1  public InterfaceReasoner getReasoner(){
2  Properties propiedades = Configuration.getInstance().cargarDriver();
3  try {
4  reasoner = (InterfaceReasoner)
   Class.forName(propiedades.getProperty("DRIVER")).newInstance();
5  this.setCargado(true);
6  } catch (InstantiationException ex) {
7  System.out.println("Instantiation Exception");
8  } catch (IllegalAccessException ex) {
9  System.out.println("Illegal Access Exception");
10 } catch (ClassNotFoundException ex) {
11 System.out.println("ClassNotFoundException");
12 }
13 return reasoner;
14 }
```

Código Fuente 22. Detalle del método *getReasoner()*.

El método *cargarDriver()* se muestra en el Código Fuente 23. La línea 4 comprueba si existe un archivo de configuración con nombre "configuration.properties" en el directorio *home* del usuario. Si no existe, crea uno por defecto (línea 7 del Código Fuente 23), asignando como driver por defecto la clase *ReasonerImplementation*. Si el archivo ya existe o si acaba de ser creado, la línea 20 del Código Fuente 23 lo carga en el objeto *Properties*, que será devuelto por el método (línea 24 del Código Fuente 23).

Si se quisiera establecer otro razonador que no fuese Pellet u otro *framework* distinto a *Jena*, se implementarían los métodos indicados en la clase *InterfaceReasoner* en una nueva clase, que se añadiría como driver al archivo de configuración, y a través de la línea 4 del Código Fuente 22 se crearía una instancia de dicha clase. La clase compilada deberá estar en el CLASSPATH de la máquina virtual con la que se lance la aplicación para que ésta sea capaz de localizarla.

```

1  public Properties cargarDriver(){
2  archivo = new File(rutaDelArchivo);
3  try {
4  if (!archivo.exists()) {
5  Properties tmp = new Properties();
6  tmp.setProperty("HOME",home);
7  tmp.setProperty("DRIVER",
   "code.google.com.p.ontologytesting.model.reasonerinterfaz.driver.ReasonerImplementation")
   ;
8  tmp.setProperty("IDIOMA", "code.google.com.p.ontologytesting.gui.internacionalization.Spanish");
9  File directorio_file = new File(home+"/.OntologyTest/");
10 try{
11 directorio_file.mkdir();
12 }catch(SecurityException ex){
13 System.out.println("No se pudo crear el directorio");
14 }
15 FileOutputStream out = new FileOutputStream(home+"/.OntologyTest/"+arch);
16 tmp.store(out, "Configuracion de OntologyTest");
17 out.close();
18 }
19 FileInputStream in =new FileInputStream(archivo);
20 propiedades.load(in);
21 }catch (IOException ex) {
22 System.out.println("Error durante la configuracion");
23 }
24 return propiedades;
25 }

```

Código Fuente 23. Método cargarDriver()

La clase *InvalidOntologyException* ha sido creada para encapsular las excepciones propias de *Jena*, y eliminar también así esas dependencias.

## 7.4 Gestión de los Menús en la interfaz gráfica

En muchas ocasiones, la interfaz nos proporcionará puntos de acceso a una misma funcionalidad desde lugares distintos, por ejemplo, desde los paneles contenedores de nuestra sesión de trabajo. Si implementáramos el código necesario para acceder a esta funcionalidad en cada componente (*JList*, *JItem*, etc.) se generaría una cantidad de código redundante muy elevada. Para evitar esto se han utilizado *Actions*, que permiten unificar el código de acceso a la funcionalidad. En *Swing* los componentes destinados a lanzar funcionalidades se pueden inicializar a través de estos *Actions*. El componente tomará aquellos elementos que necesite, descartando el resto.

Estos *Actions* se han utilizado en los submenús que aparecen al hacer clic derecho sobre las listas de Tests Simples, Tests Sparql e Instancias que se encuentran en la parte izquierda de la interfaz de usuario.

En el Código Fuente 23 se presenta el caso de la lista de de Tests Simples. Si se hace clic sobre la misma se ejecuta éste código. En la línea 2 se obtienen la lista de elementos contenidos en el panel TestsSimples. En la línea 4 se fija el nombre del test seleccionado en la clase *PopMenuTests*, a través del objeto *popTest*, para poder realizar las acciones en el test seleccionado posteriormente. En la línea 5 se crea un evento *MouseListener* sobre los elementos creados a través del método *createPopupMenuForTests()* de la clase *PopMenuTests*. Será en ésta clase dónde se traten los eventos, en función de la selección hecha sobre la lista.

```

1 private void testSimplesListValueChanged(javax.swing.event.ListSelectionEvent evt) {
2     JList lista = (JList) evt.getSource();
3     if(modeloSimples.getSize()>0){
4         popTest.setTestSelec(modeloSimples.get(lista.getLeadSelectionIndex()). toString());
5         MouseListener popupListener = new PopupListener(popTest.createPopupMenuForTests());
6         lista.addMouseListener(popupListener);
7     }}

```

Código Fuente 23. Detalle del método *testSimplesListValueChanged*

En el Código Fuente 24 se ve como en la línea 2 se obtiene el elemento seleccionado de la lista, y en las líneas 6,7,8,9 y 10 se realiza la acción oportuna en función de qué elemento haya sido seleccionado.

```

1 public void actionPerformed(ActionEvent e) {
2   JMenuItem source = (JMenuItem)e.getSource();
3   ScenarioTest scenario = s.buscarScenario(collection.getScenariotest(), this.getTestSelec());
4   if(source.getText().equals("Editar")){
5   }else if(source.getText().equals("Eliminar Instancias")){
6   }else if(source.getText().equals("Ejecutar")){
7   }else if(source.getText().equals("Ver")){
8   }else if(source.getText().equals("Eliminar")){
9   }

```

Código Fuente 24. Detalle del método actionPerformed de clase PopMenuTests

## 7.5 Creación de un .jar con la aplicación empaquetada

El empaquetar la aplicación en un único archivo ejecutable para poder distribuirla de forma sencilla no es una tarea simple debido al gran número de dependencias (librerías externas) de la aplicación. Para resolver este problema se ha utilizado la herramienta One-Jar [one-jar]. A través de ésta, una aplicación Java puede empaquetarse en un archivo ejecutable siguiendo los siguientes pasos:

- 1 Descargarse el paquete [one-jar-boot-0.96.jar](http://one-jar.sourceforge.net/index.php?page=downloads&file=downloads) de la página web: <http://one-jar.sourceforge.net/index.php?page=downloads&file=downloads>
- 2 Crear un archivo ejecutable JAR que contenga los ficheros CLASS de la aplicación mediante, por ejemplo, el siguiente comando: `jar cfm main.jar MANIFEST.MF *.class`. El archivo `MANIFEST.MF` indica la clase principal de la aplicación así como la referencia a las librerías necesarias para que ésta funcione.



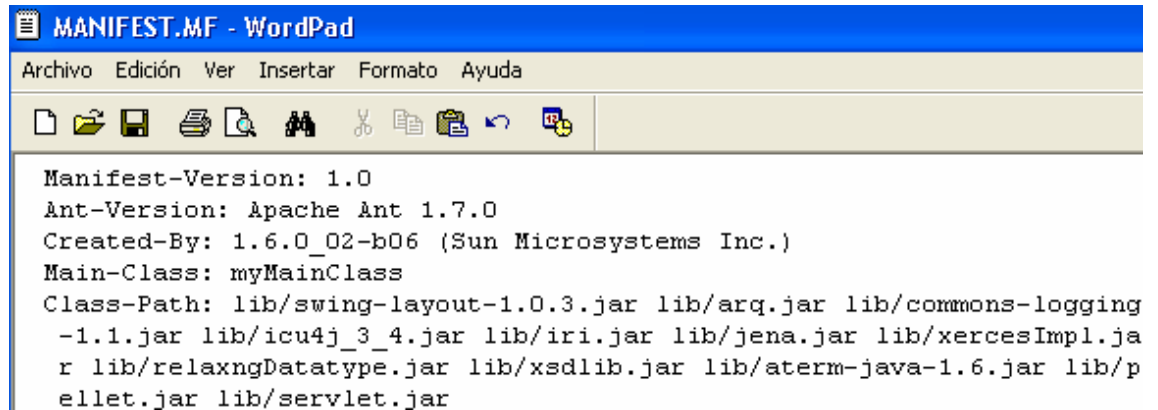


Figura 56 Detalle del archivo MANIFEST.MF

- 3 Crear tres directorios llamados: main, lib, y boot, dentro de una carpeta.

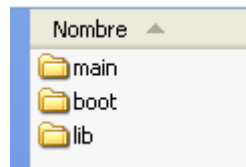


Figura 57 Estructura de directorios a crear

- 4 Colocar el archivo main.jar creado en el apartado 2 dentro de la carpeta main.
- 5 Colocar todas las librerías de las que depende la aplicación dentro de la carpeta lib.
- 6 Crear un nuevo archivo JAR en el mismo directorio de las carpetas main, boot y lib, tal y como se ve en la Figura 58, con el contenido de las carpetas main y lib, mediante el siguiente comando: `jar cf application.jar main lib`

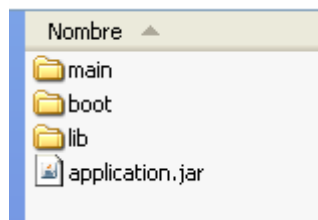
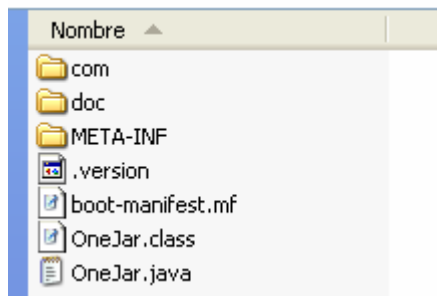


Figura 58 Creación del archivo application.jar

- 7 Extraer el contenido del archivo descargado [one-jar-boot-0.96.jar](#). Para ello hay que descomprimir dicho archivo con cualquier herramienta capaz de descomprimir archivos .zip, procediendo como si de un archivo .zip se tratara. El contenido de la extracción se muestra en la Figura 59. Éste contenido hay que copiarlo y guardarlo dentro de la carpeta boot.



**Figura 59** Detalle del contenido del archivo one-jar-boot-0.96.jar

- 8 Situar en el directorio boot y actualizar el archivo `application.jar` creado en el punto 6, mediante el comando: `jar -uvfm ../application.jar boot-manifest.mf *.*`

El archivo `application.jar` contiene ahora nuestra aplicación Java junto con todas sus dependencias.

## **7.6 Internacionalización**

La herramienta ha sido desarrollada para que el usuario pueda seleccionar el idioma con el que se le presentará la interfaz gráfica, bien en inglés o en español. Para esto se ha utilizado la herramienta de internacionalización que tiene *Netbeans 6.1*. Los pasos a seguir han sido los siguientes:

Desde la barra de herramientas seleccionar Tools→Internationalization→Internationalization Wizard (ver Figura 60).

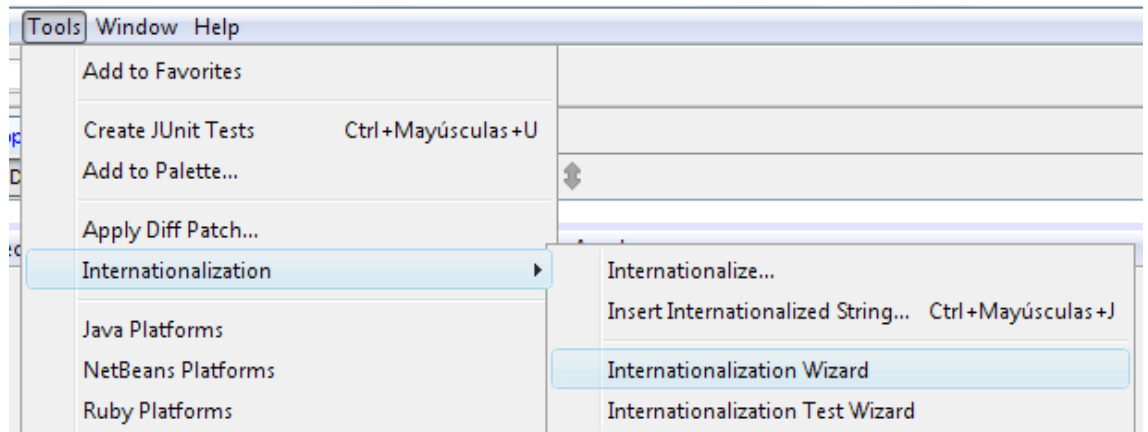


Figura 60 Primer paso del asistente de Netbeans para la internacionalización

Esto nos conduce a la pantalla de la Figura 61 donde tendremos que añadir todas las clases que se quieran internacionalizar. Es decir, todas las clases que contengan cadenas de caracteres que se quieran traducir, deberán añadirse en este paso.

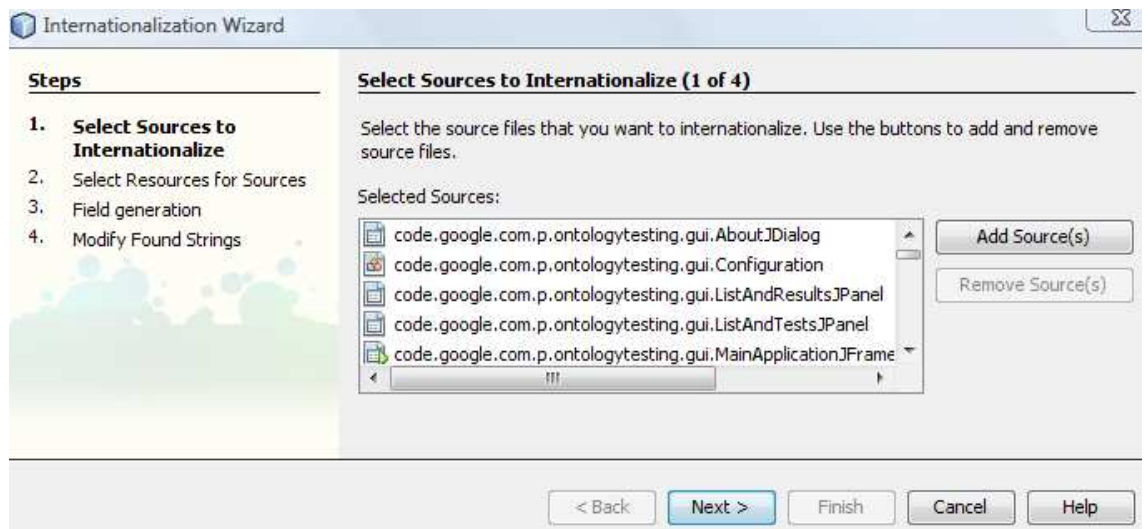
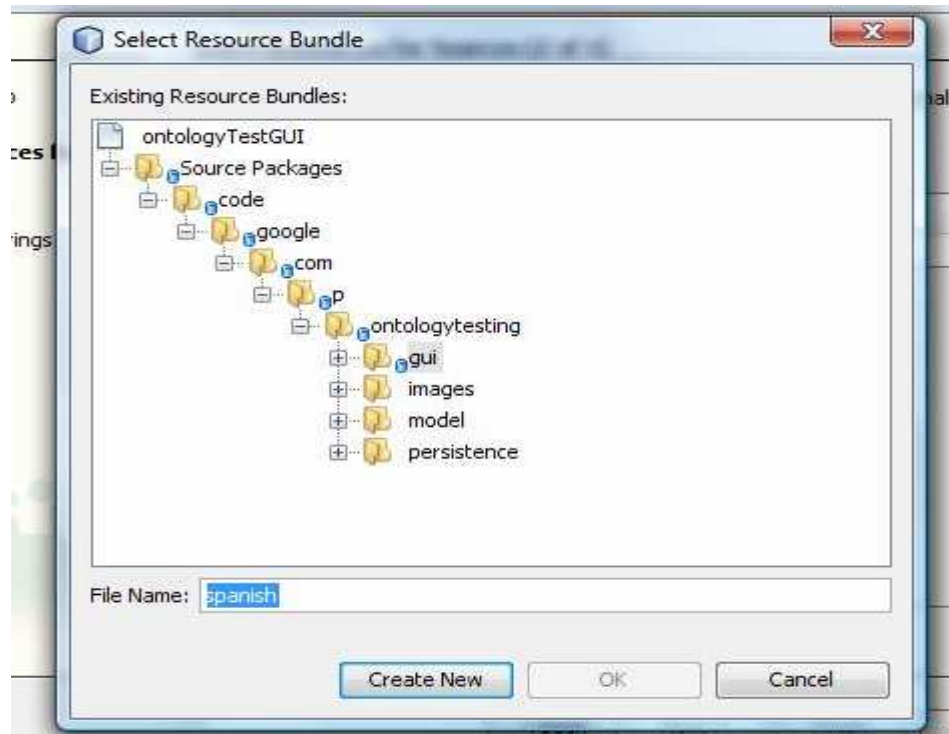


Figura 61 Selección de las clases a internacionalizar

Si avanzamos al siguiente paso se pedirá que se seleccionen los recursos del código dónde se generará el archivo de internacionalización. Si se selecciona “*Select All*” se abrirá una pequeña pantalla de navegación por el proyecto para que seleccionemos donde guardar el archivo. En nuestro caso se ha creado archivo llamado “spanish” (ver Figura 62).



**Figura 62** Detalle del proceso de internacionalización

En el último paso se pedirá, para cada clase que se haya indicado que se quiere internacionalizar, qué cadenas de caracteres concretas se desean internacionalizar. Se tendrán que seleccionar/deseleccionar las que interesen (ver Figura 63). Tras completar este proceso y darle a Finalizar, *Netbeans* habrá creado un archivo llamado *spanish.properties* con todas las cadenas de caracteres que se hayan seleccionado en el paso cuatro del asistente, y habrá sustituido en el código fuente, donde aparecen dichas cadenas, por un recurso del tipo: `java.util.ResourceBundle.getBundle("code/google/com/p/ontologytesting/gui/spanish")`. Éste será sustituido por el que aparece en el Código Fuente 25, para automatizar el proceso y que se cargue desde un archivo de propiedades.

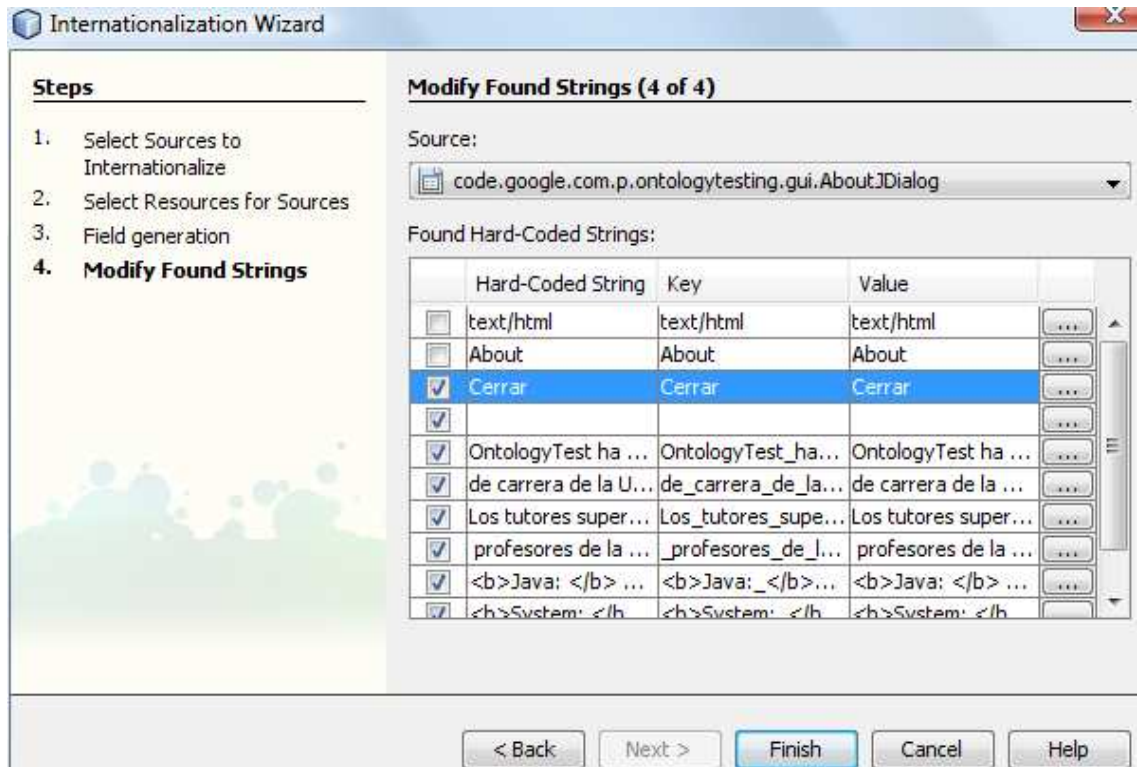


Figura 63 Último paso del proceso de internacionalización

El siguiente paso es copiar el archivo *spanish.properties* y renombrar la copia, modificando el contenido al inglés. Por ejemplo, se tendrán dos archivos, un *spanish.properties* (ver Figura 64) y otro llamado *englishUk.properties* (ver Figura 65).

```
EVALUADOR_DE_ONTOLOGIAS=EVALUADOR DE ONTOLOGIAS
Proyecto=Proyecto
Ubicación_de_la_Ontología_y_el_Namespace=Ubicación c
Instancias=Instancias
Tests_Sparql=Tests Sparql
Tests_Simples=Tests Simples
Test_Simple=Test Simple
Test_Sparql=Tests Sparql
Test_de_Clasificación=Test de Clasificación
Test_de_Satisfactibilidad=Test de Satisfactibilidad
Test_de_Realización=Test de Realización
Test_de_Recuperación=Test de Recuperación
Test_de_Instanciación=Test de Instanciación
```

Figura 64 Detalle del archivo *spanish.properties*

```
EVALUADOR_DE_ONTOLOGIAS=ONTOLOGY EVALUATOR
Proyecto=Project
Ubicación_de_la_Ontología_y_el_Namespace=Ontology
Instancias=Instances
Tests_Sparql=Tests Sparql
Test_Sparql=Tests Sparql
Test_Simple=Test Simple
Tests_Simples=Tests Simples
Test_de_Clasificación=Classification Test
Test_de_Satisfactibilidad=Satisfactibility Test
Test_de_Realización=Realization Test
Test_de_Recuperación=Retrieval Test
Test de Instanciación=Instantiation Test
```

**Figura 65** Detalle del archivo english.properties

En el caso de esta aplicación, el recurso creado por defecto (*java.util.ResourceBundle.getBundle("code/google/com/p/ontologytesting/gui/spanish")*) ha sido modificado para que sea cargado del archivo de propiedades, y así seleccionar un idioma u otro dependiendo de las preferencias del usuario. Cuando el usuario seleccione el idioma en la aplicación, éste se almacenará en el archivo de propiedades con el nombre de IDIOMA:

*IDIOMA=code.google.com.p.ontologytesting.gui.internacionalization.EnglishGB*

La próxima vez que se arranque la aplicación, todas las cadenas de caracteres internacionalizadas se leerán a través de la línea del Código Fuente 25.

<pre>java.util.ResourceBundle.getBundle(Configuration.getProperties().getProperty("IDIOMA"),new Locale(Configuration.getProperties().getProperty("LOCALE"))).getString("Guardar");</pre>
--

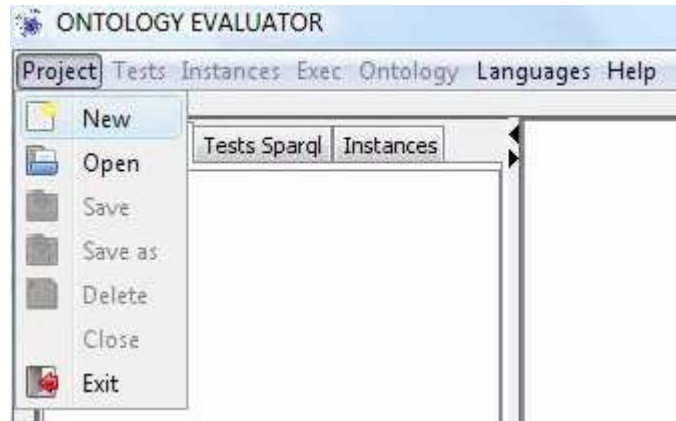
Código Fuente 25. Detalle de la carga del idioma desde el archivo de propiedades.

También se ha añadido al archivo de propiedades la propiedad LOCALE. Esta propiedad permite que las cadenas de caracteres pertenecientes a la API de Java se traduzcan de forma automática al idioma establecido a través de la propiedad *setDefaultLocale()*. En este caso, dicha propiedad se ha establecido como “es” para el español, o “uk” para el inglés (ver Código Fuente 25).

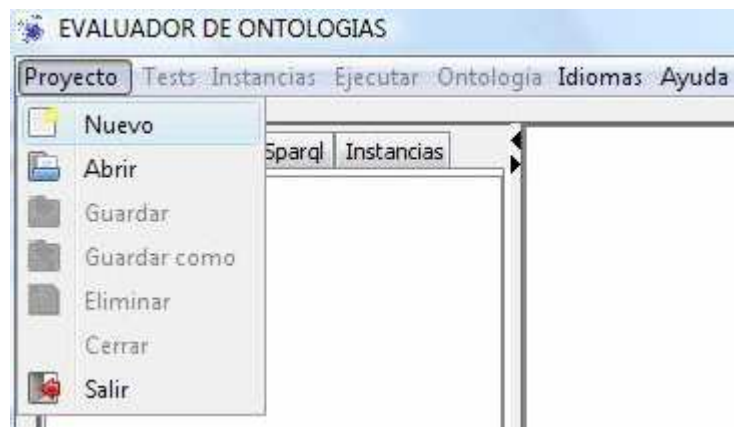
```
Locale.setDefault(new Locale(Configuration.getProperties.getProperty("LOCALE")));
```

Código Fuente 26. Detalle de la asignación del idioma a la propiedad LOCALE

En la Figura 66 puede verse el la pantalla principal de la interfaz gráfica de OntologyTest en español, y en la Figura 67 la misma interfaz en inglés.



**Figura 66** Detalle de la interfaz gráfica en inglés



**Figura 67** Detalle de la interfaz gráfica en español

## 8 Verificación, Validación y Pruebas

La verificación del código desarrollado y la validación del modelo es esencial en cualquier desarrollo *software*. Proporcionar un código de calidad, minimizando el número de *bugs*, es básico en el desarrollo de un producto para que no fracase. Si el usuario final encuentra comportamientos no esperados por parte del producto de manera regular como consecuencia de errores en la implementación, probablemente dejará de utilizarlo pese a que su diseño sea el correcto. En el otro extremo, se debe comprobar que el resultado del producto cumple con lo que espera el usuario, ya que un producto con un código de calidad estará condenado al fracaso si no cumple con las expectativas del usuario.

Por tanto la verificación (¿Estamos desarrollando correctamente el producto?) y la validación (¿El producto se adecua a las necesidades para las que fue diseñado?) son indispensables.

La verificación se ha realizado mediante *code review* (ver página) y el uso de herramientas de análisis estático de código [wikipedia.org b]. La validación se llevó a cabo realizando tests sobre ontologías desde prácticamente el primer día. El desarrollo de estos tests ha sido llevado a cabo casi en su totalidad por el desarrollador de OntologyTest.

A continuación se describirán más en detalle cómo se llevaron a cabo estas tres prácticas que garantizan la calidad del *framework* desarrollado.

### 8.1 Code review

El experto en el dominio de aplicación en cuanto al desarrollo software se refiere (el profesor Abraham Otero) posee conocimientos de ingeniería de software y de Java, por lo que se involucró activamente en el proyecto participando en las revisiones de código. Estas revisiones fueron regulares, siempre dependiendo de las necesidades del alumno, favoreciendo las correcciones continuas y evitando que aquellos fallos identificados se reflejen en otras partes del código de una manera indirecta.



Para que la revisión de código se pueda realizar de una manera eficiente la última versión del código fuente del proyecto debe estar siempre accesible tanto a desarrolladores como revisores y debe existir facilidad para la comunicación entre ambas partes.

Para el acceso al código se ha utilizado un sistema de control de versiones. Para facilitar el acceso al mismo desde diferentes ubicaciones se decidió la creación de un proyecto de *software* alojado en el repositorio de google [code.google]. Gracias a esto se ha tenido soporte para utilizar un sistema de control de versiones Subversion accesible desde Internet [Bar03]. La actualización del código alojado en el Subversion se realizaba de modo inmediato una vez se había desarrollado una nueva funcionalidad.

Con una periodicidad aproximadamente de un par de veces cada dos semanas, se realizaban reuniones con ambos expertos con el fin de que cada uno en su campo, expusiesen los resultados de sus revisiones con más detalle y aclarase las dudas del equipo de desarrollo. Estas reuniones, por lo tanto, sirvieron tanto para la verificación como para la validación.

## **8.2 Análisis estático de código**

Para el análisis estático de código se ha utilizado la herramienta *Findbugs* [sourceforge a], desarrollada por la universidad de *Maryland*, con la financiación, entre otros, de la propia *Sun Microsystems*, propietaria de la plataforma *Java*.

Esta herramienta ayuda a detectar un número elevado de puntos en el código fuente que sean *bugs* potenciales. A lo largo del desarrollo del código, se han ido detectando una serie de *bugs* que han sido corregidos. Sin embargo, la versión definitiva de la aplicación contiene un *bug* que no ha sido corregido, ya que es un error. Este es:

**1. Un método invoca a `System.exit(...)`.** Para cerrar la aplicación, es necesario invocar a éste método. Findbug advierte de que la máquina virtual de java se cerrará, pero es el comportamiento que se espera.

### 8.3 Validación

La única forma de validar una aplicación es desarrollando modelos reales para la misma. Sólo entonces se puede comprobar si el modelo de programación que impone el la aplicación se adecua a los problemas que trata de resolver y a las expectativas del usuario.

Al contar en el equipo con un experto en el campo de las ontologías, el proceso de validación de la herramienta ha sido realizado por él. Esto ha permitido detectar una serie de errores no detectados con anterioridad y que han sido corregidos en la versión final del software. También han surgido mejoras que podrían llevarse a cabo en versiones futuras.

Para realizar la validación, el experto ha preparado un conjunto de meta-pruebas a las que ha sometido a la herramienta. Las ontologías con las que ha realizado las meta-pruebas se ven en la , si bien únicamente forman parte de las que se expondrán en detalle las relacionadas con la ontología *family.owl*.

Ontología	URL	Clases	Propiedad es objeto	Propiedad es dato	Numero axiomas
ULMS.owl	<a href="http://www.phinformatics.org/Assets/Ontology/ulms.owl">http://www.phinformatics.org/Assets/Ontology/ulms.owl</a>	135	0	5	133
chemicalSubstance.owl	local	19	10	0	26
Family.owl	<a href="http://www.owl-ontologies.com/family.owl">http://www.owl-ontologies.com/family.owl</a>	4	1	0	4
Ontosem.owl	<a href="http://morpheus.cs.umbc.edu/aks1/ontosem.owl">http://morpheus.cs.umbc.edu/aks1/ontosem.owl</a>	7596	604	0	7992

**Tabla 4 Ontologías para el desarrollo de meta-pruebas**

Una síntesis de las pruebas para el Proyecto, las Instancias y los Tests ha los que ha sido sometido OntologyTest puede verse en las tablas respectivamente. Los errores detectados como de tipo Medio han sido corregidos para la versión final del software.

Objeto	Operación	Resultado	Gravedad del defecto detectado
<b>Proyecto</b>	Crear	Si la ontología tiene algún defecto, no se especifica cuál es éste.	Leve
	Modificar nombre	No hay opción para mostrar el nombre	Leve
	Mostrar nombre	No hay opción para mostrar el nombre	Leve
	Mostrar ontología	Falta la posibilidad de “buscar” dentro de la ontología.	Leve
	Cerrar		OK
	Cargar	Si el usuario intenta reutilizar el directorio de un proyecto ya borrado, el sistema responde que no se ha podido crear un directorio para el proyecto.	<u>Media</u>
<b>Instancias</b>	Asociar		OK
	Modificar	Cuando se modifica el conjunto de instancias asociado a un test, al guardarlo (pulsando sólo “guardar”, no “guardar y asociar”) se crea una copia nueva sin vincular. Es decir, acaba habiendo dos copias: (1) la que se había asociado al test y (2) la modificada.	<u>Media</u>
	Mostrar		OK
	Borrar	No se cierra la ventana que muestra los datos del conjunto de instancias borrado.	<u>Media</u>

**Tabla 5 Síntesis de las meta-pruebas para el Proyecto y las Instancias**

<i>Objeto</i>	Operación	Resultado	Gravedad del defecto detectado
<b>Tests</b>	Crear	La pestaña de “Nueva Instanciación” aparece en español en la versión en inglés.	<u>Media</u>
		No hay copiar y pegar (para pegar términos de la ontología en los tests).	Leve
		En el caso del Test SPARQL, si se produce un error de sintaxis al realizar la consulta, no se indica de qué tipo de error se trata.	<u>Media</u>
		No cambia el nombre del test al guardarlo si se mantiene abierto.	Leve
	Modificar	OK	OK
	Ejecutar	OK	OK
		En la versión en inglés la etiqueta “RESULT EXPECTED” del formulario de tests es incorrecta. Debe poner “EXPECTED RESULT”.	Leve
	Mostrar	Errores gramaticales y de traducción inglés/español	<u>Media</u>

	Borrar	No se cierra la ventana que muestra los datos del test.	<b><u>Media</u></b>
--	--------	---	---------------------

**Tabla 6 Síntesis de las pruebas para los Tests**

## **9 Evolución del proyecto y Conclusiones**

Hasta el momento, no existía ninguna herramienta que permitiese elaborar tests para probar dinámicamente los requerimientos funcionales de las ontologías. Para solucionar éste problema, se ha desarrollado la herramienta *OntologyTest*, que ha sido descrita a lo largo de este documento. Ésta permite al usuario definir una serie de tests y, para cada uno, establecer un escenario inicial (la base de hechos de la ontología para ese test), una serie de consultas y sus resultados. Los resultados de la ejecución de los tests se presentan en un formato similar al de JUnit, indicando para cada consulta de cada test cuáles han pasado y cuáles han fallado y por qué.

*OntologyTest* permite elaborar tests de forma sistemática y probarlos cuántas veces sea necesario, pudiendo ejecutar todas las pruebas automáticamente cada vez que se hacen cambios sobre la ontología.

Con esto un usuario puede ir, a medida que va desarrollando una ontología, probando si es conforme a sus especificaciones, ya que puede ir realizando pruebas durante todo el proceso de desarrollo. Lo ideal sería que el usuario crease una serie de tests antes de desarrollar la ontología, y una vez terminada ésta, los ejecutase para comprobar los resultados, de igual modo que se realizan los tests en JUnit, a priori del desarrollo del software.

Este proyecto final de carrera ha supuesto un avance tecnológico significativo en el desarrollo de ontologías en OWL DL, pues, hasta ahora, herramientas de este estilo existían para el desarrollo de software convencional, por ejemplo, en Java, pero no para el desarrollo de ontologías en OWL.

*OntologyTest* está implementado en Java y tiene una interfaz similar a la de *Netbeans*.

La herramienta tiene que trabajar con conjuntos de instancias añadidos por el usuario directamente a través de la propia herramienta. No soporta el hecho de leer directamente del esquema de definición de la ontología las instancias que ya podía tener asociadas, que hubiesen sido añadidas al crear la ontología, con otras herramientas

como Protégé.

Otra limitación a tener en cuenta es el hecho de que OntologyTest trabaja sobre un razonador concreto, Pellet. Existen otros muchos razonadores hoy en día capaces de inferir y deducir conocimiento sobre las ontologías, con los que OntologyTest podría trabajar, tales como KAON2, RacerPro, etc. OntologyTest está implementado de tal forma que el uso del razonador Pellet es un driver para la aplicación, por lo que el hecho de implementar otra serie de *drivers* no es un proceso complicado.

Otro punto de extensión de OntologyTest es el hecho de aumentar la batería de tests que soporta. Hasta ahora implementa seis tipos de tests: instanciación, recuperación, realización, satisfactibilidad, clasificación y SPARQL, pero este número podría crecer en función de las necesidades del usuario. Conjuntos de tests adaptados a las nuevas necesidades o a la evolución del campo de las ontologías podrían ser desarrollados para aumentar la eficiencia y la utilidad de la herramienta.

## 10 Tutoriales

### 10.1 Crear y trabajar con un proyecto

En este apartado se va a explicar de forma detallada cómo crear un proyecto de trabajo mediante OntologyTest y cómo utilizarlo para evaluar ontologías.

Si se ejecuta la aplicación, se accede a una pantalla que muestra un menú principal con el aspecto que muestra la Figura 68.

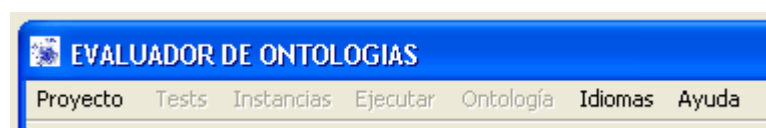


Figura 68 Menú principal de la aplicación

Si nos colocamos sobre “Archivo”, podremos seleccionar la acción a realizar, bien crear un nuevo proyecto o bien abrir un proyecto existente, tal y como se ve en la Figura 69.

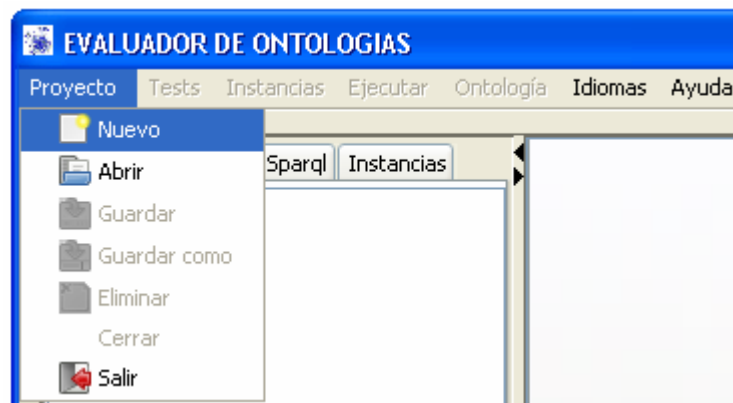
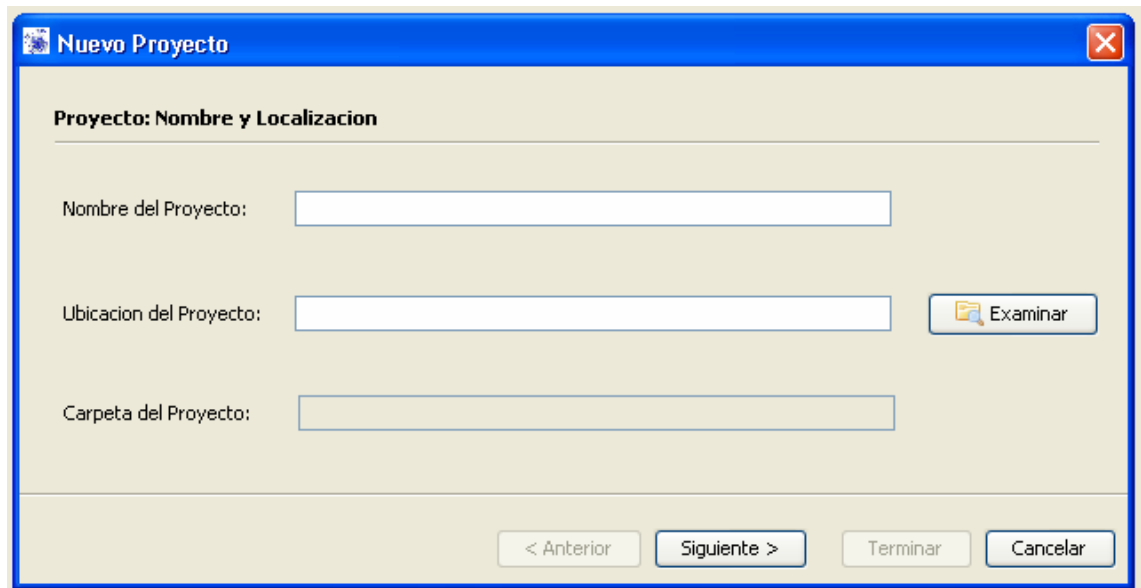


Figura 69 Opciones del menú Archivo

### 10.1.1 Crear un proyecto nuevo

Si seleccionamos la opción “Nuevo” de la Figura 69, accedemos a la pantalla que se ve en la Figura 70, en la que se nos solicitan dos campos: el nombre que le queremos dar al proyecto y su ubicación, dónde queremos guardarlo en nuestro ordenador.



**Figura 70 Paso 1 de la creación de un proyecto**

Una vez completados los pasos anteriores, si hacemos clic en “Siguiente”, accedemos a la pantalla que muestra la Figura 71, dónde se nos solicitan dos campos más: la ubicación física de la ontología a evaluar y el *namespace* de la misma.



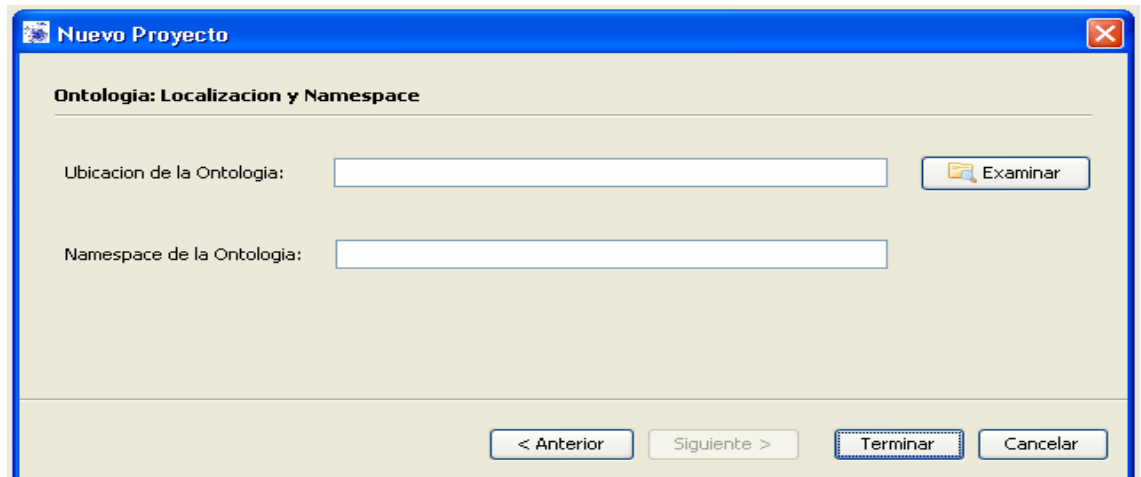


Figura 71 Paso 2 de la creación de un proyecto

Una vez completados estos datos, si pinchamos sobre “Terminar” el proyecto se creará y se activarán el resto de opciones del menú principal, como muestra la Figura 72, que hasta este momento aparecían como no permitidas tal y como se mostraba en la Figura 68. Si la ontología seleccionada no es válida, aparecerá un mensaje de error indicando éste hecho. Sólo es posible trabajar con ontologías bien formadas.

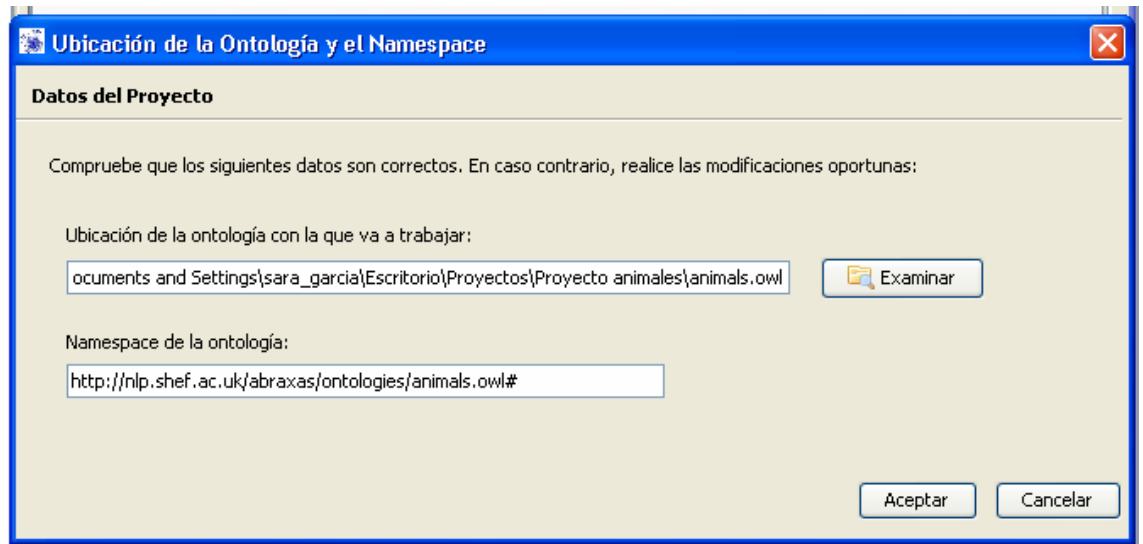


Figura 72 Menú principal una vez cargado el proyecto

### 10.1.2 Abrir un proyecto existente

Si se selecciona la opción “Abrir” de la Figura 69, se nos abrirá un explorador de archivos para que seleccionemos qué proyecto queremos abrir. Éste proyecto tiene que haber sido creado previamente mediante ésta herramienta y tener una extensión .xml.

Una vez seleccionado el proyecto, se nos pedirá que confirmemos la ubicación física de la ontología con la que vamos a trabajar y su namespace, tal y como se ve en la Figura 73.



**Figura 73** Proceso de carga de un proyecto

Si hacemos clic en “Aceptar” y la ontología seleccionada es válida, el proyecto se cargará.

### 10.1.3 Crear un Test Simple

Para crear un test simple, por ejemplo, un test de instanciación, sobre el menú principal seleccionamos la opción Tests→Nuevo→Test Simple→Test de instanciación, como se ve en la Figura 74.

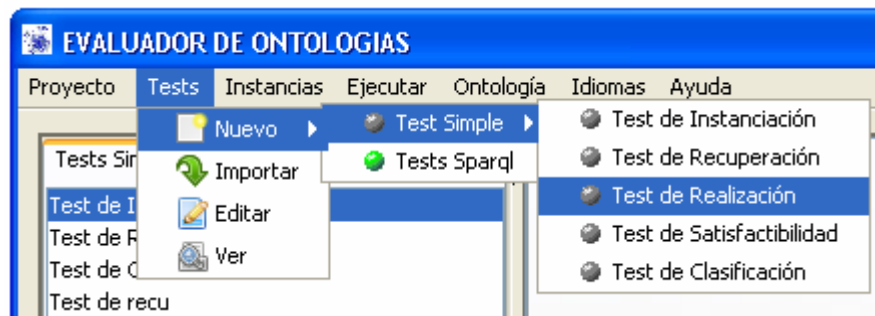


Figura 74 Creación de un test simple

En la zona derecha de la aplicación, la dedicada al desarrollo de tests e instancias, aparecerá un nuevo test de instanciación, para que sea completado por el usuario y que se muestra en la Figura 75.

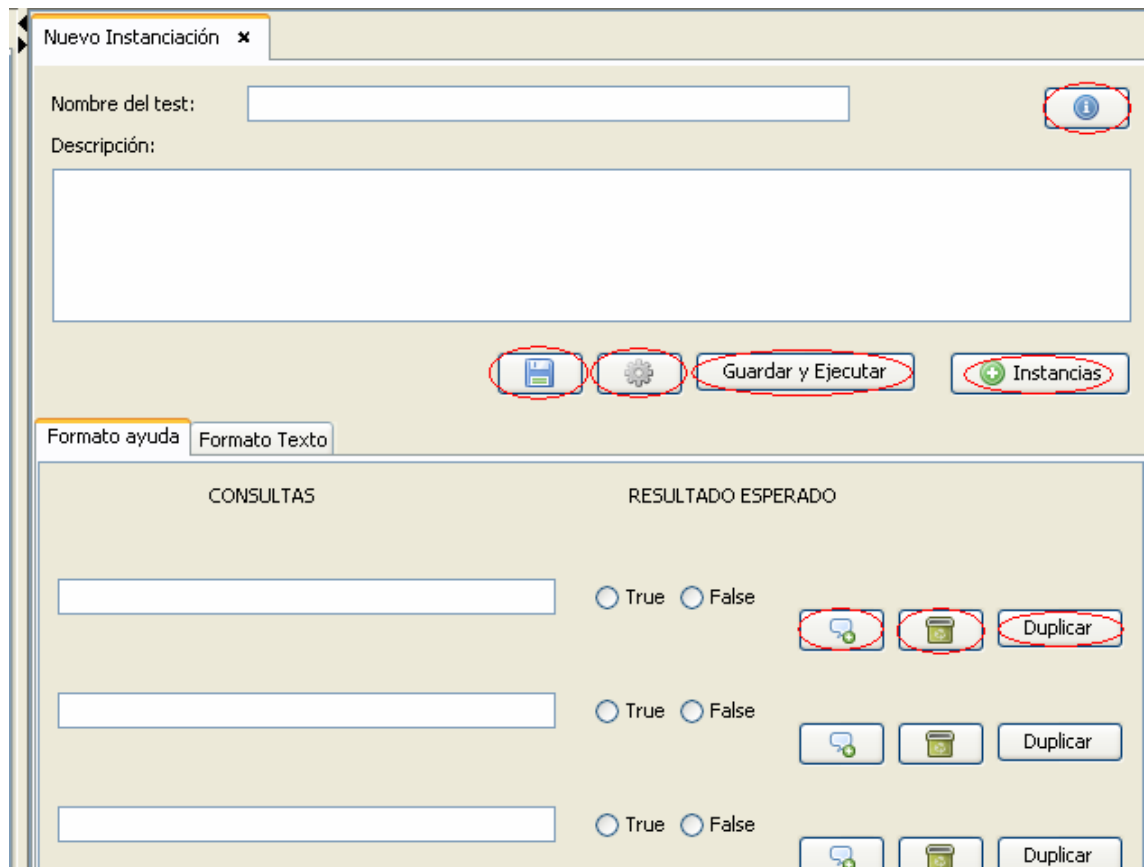




Figura 75 Detalle de la creación de un test de instanciación

Para un nuevo test, se introducirá el nombre del mismo (campo obligatorio y único), una breve descripción (opcional) y las consultas que se desean realizar junto con los resultados que se espera obtener. Otra posible vista para el test y para la forma de introducir las consultas sería mediante el “Formato Texto”, cuya interfaz se ve en la Figura 76.

**Figura 76 Formato Texto para la creación de un test de instanciación**

El botón  de la Figura 75 nos permite acceder a una pantalla de ayuda, que nos indica el formato de las consultas.

El botón  nos permite guardar el test. Si guardamos el test, éste aparecerá automáticamente contenido en la parte derecha de la aplicación, dedicada a almacenar los tests y las instancias realizados, en forma de lista, tal y como se ve en la Figura 77.

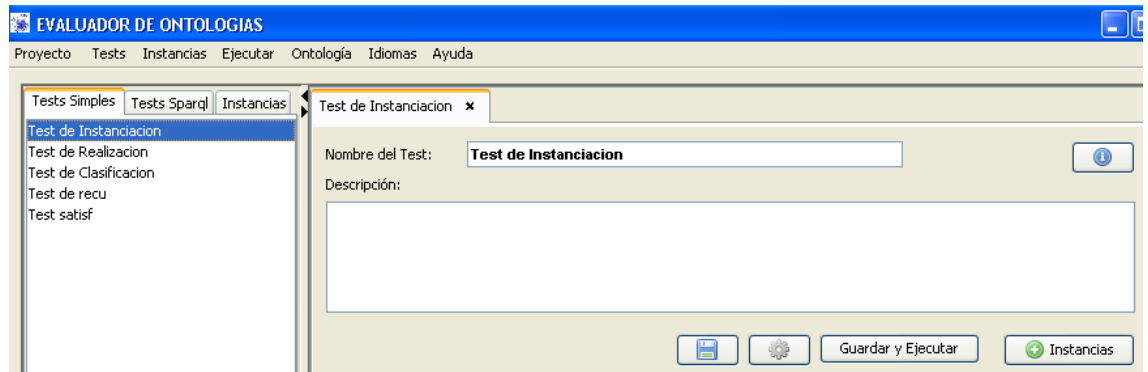




Figura 77 Detalle del almacenamiento de un test simple

El botón  ejecuta el test.

El botón  ejecuta y guarda el test.

El botón  asocia instancias al test.

El botón  de la Figura 75, abre la pantalla que muestra la Figura 78 y que permite añadir un comentario a la consulta.

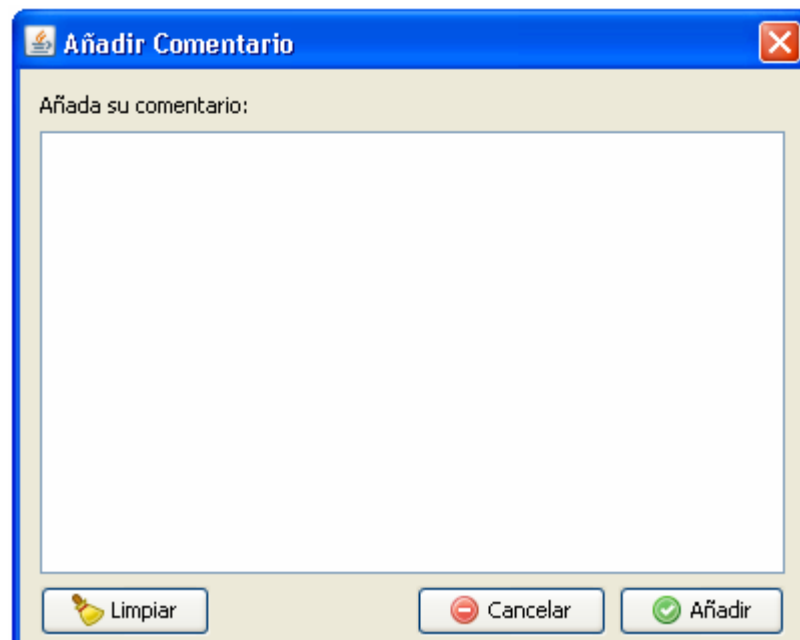



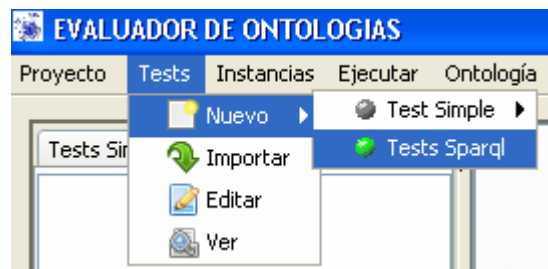
Figura 78 Panel para añadir un comentario a una consulta

El botón  borra la consulta.

El botón  duplica la consulta.

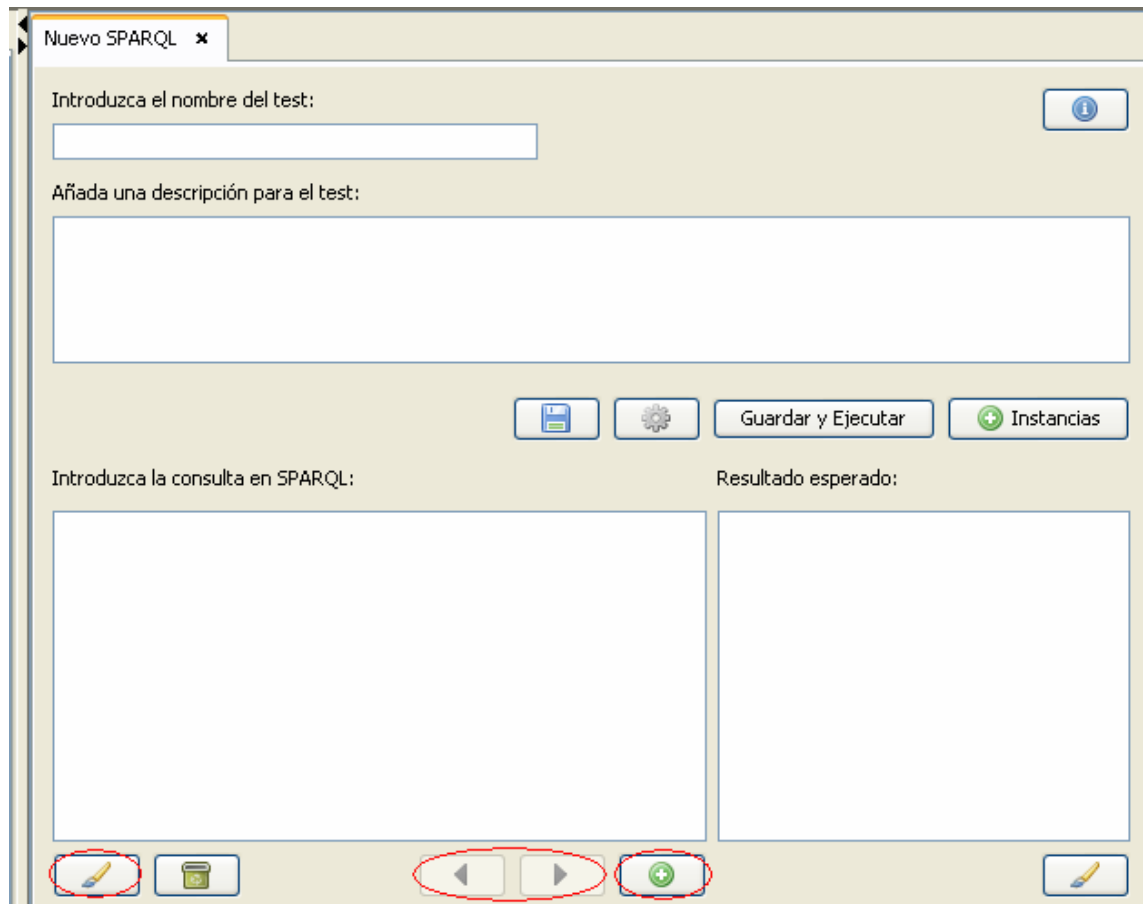
#### 1.10.4 Crear un Test Sparql

Para crear un test sparql, sobre el menú principal seleccionamos la opción Tests→Nuevo→Test Sparql, como se ve en la Figura 79.



**Figura 79 Creación de un test sparql**


Esta acción abrirá en la zona de desarrollo de tests e instancias un test sparql vacío, que se ve en la Figura 80.





**Figura 80 Nuevo Test Sparql**

Para el test sparql también se introducirá un nombre (obligatorio y único) y una descripción (opcional). Las consultas se añadirán en el área de texto de la izquierda y el resultado esperado para la misma en el de la derecha.

Los botones que ya aparecían en la creación de los tests simples tienen el mismo significado (ver 10.1.3 Crear un Test Simple).

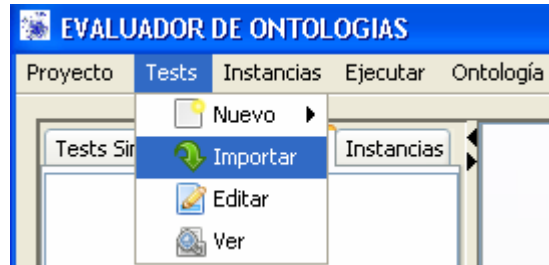
El botón  que aparece en la Figura 80, limpia la consulta.

Los botones  nos permiten avanzar por todas las consultas creadas para ese test.

El botón  nos permite añadir una nueva consulta al test.

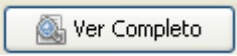
### 10.1.5 Importar un Test

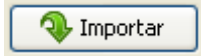
Para importar un test, nos colocaremos en el menú principal y seleccionaremos Tests→Importar, como muestra la Figura 81.



**Figura 81 Importar un test**

Esta acción nos llevará a la pantalla que se ve en la Figura 82 y que nos pedirá que seleccionemos, mediante un explorador de archivos, el proyecto del cuál queremos importar los tests. Una vez seleccionado el proyecto, se cargarán en el panel de la izquierda, todos los tests que contiene dicho proyecto, y en el de la derecha aparecerá el nombre y la descripción que tiene cada tests, para así poder hacernos una mejor idea de que test se trata.

El botón  nos abrirá una nueva ventana que contendrá el test al completo, incluyendo sus instancias (ver Figura 83).

El botón  importará los tests seleccionados a nuestro proyecto actual.

El botón  cancelará la acción.



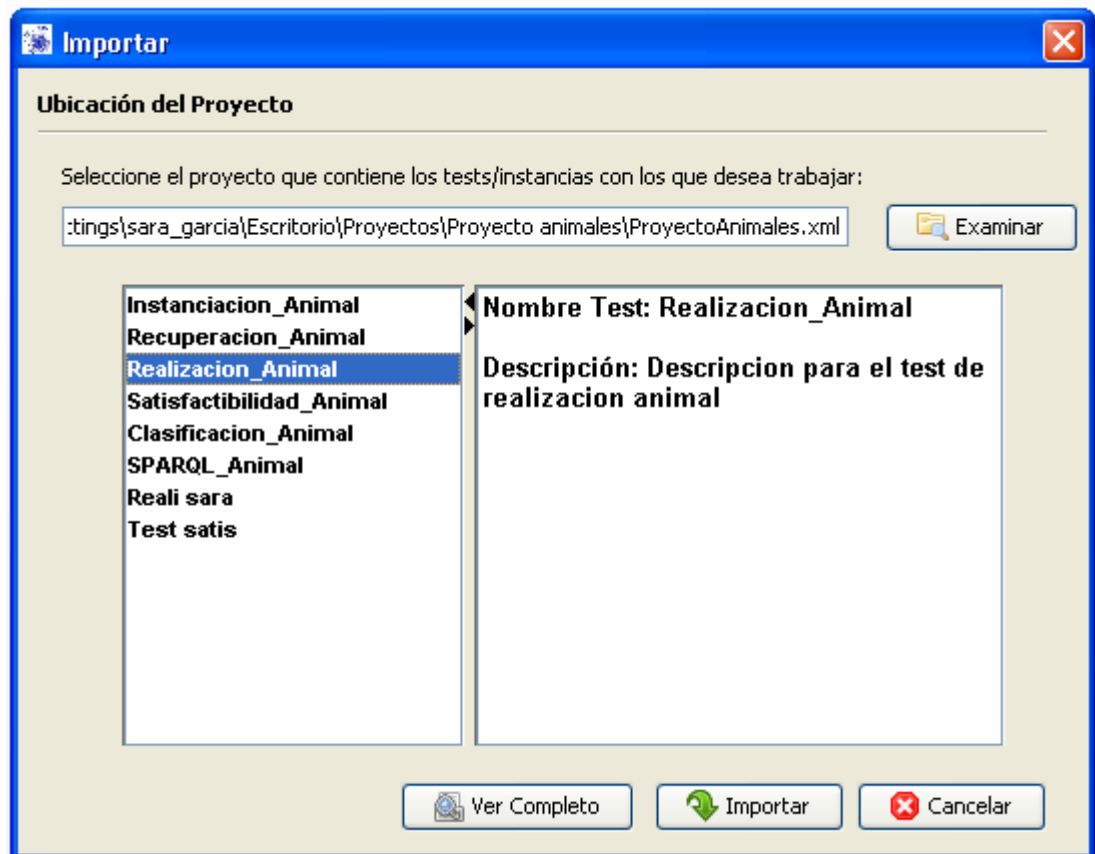


Figura 82 Proceso de importar un test

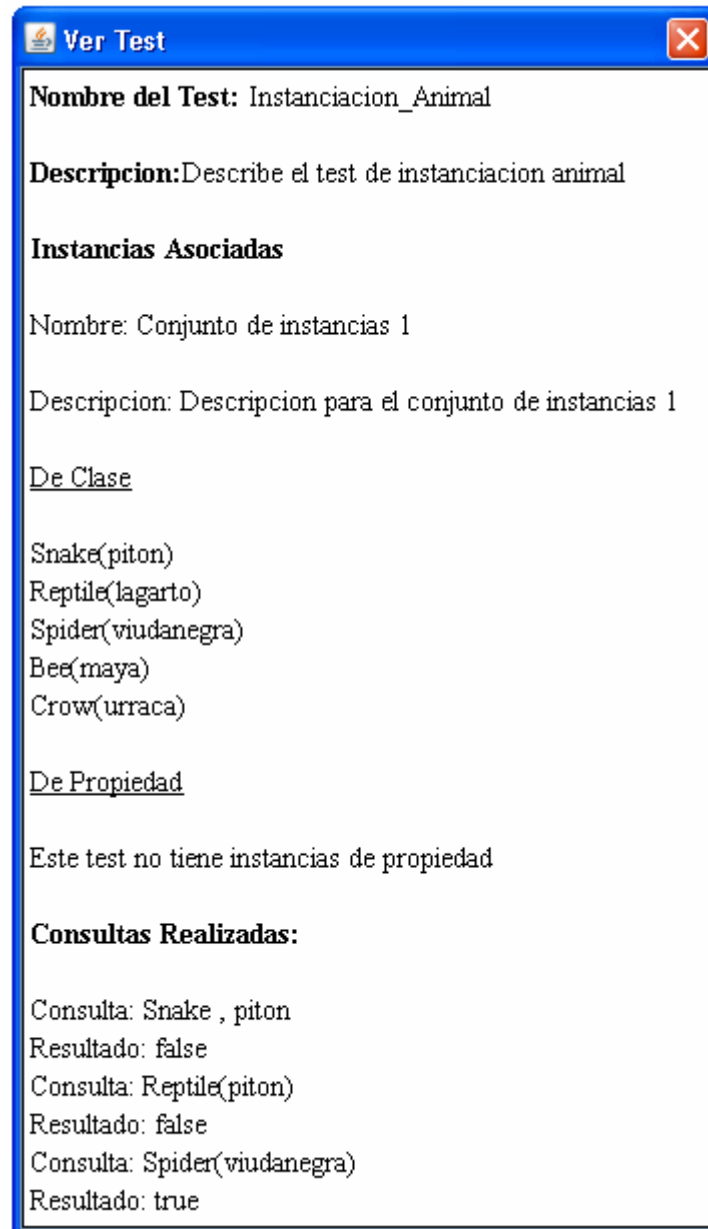
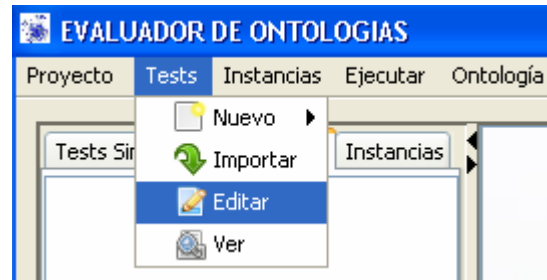


Figura 83 Vista completa de un test

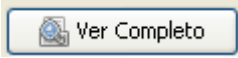
#### 10.1.6 Editar un test


Para editar un test, seleccionaremos del menú principal la opción Tests→Editar, tal y como muestra la Figura 84.



**Figura 84 Editar un test**

Esta acción nos llevará a la pantalla de la Figura 85, que nos mostrará, en la parte izquierda, todos los tests que tenemos actualmente guardados en la sesión de trabajo, y, en la derecha, una breve descripción del test.

Con el botón  podemos ver una descripción más extensa del test, tal y como se ve en la Figura 83.

Con el botón  abriremos el test para su edición.

Con el botón  cancelaremos la acción.

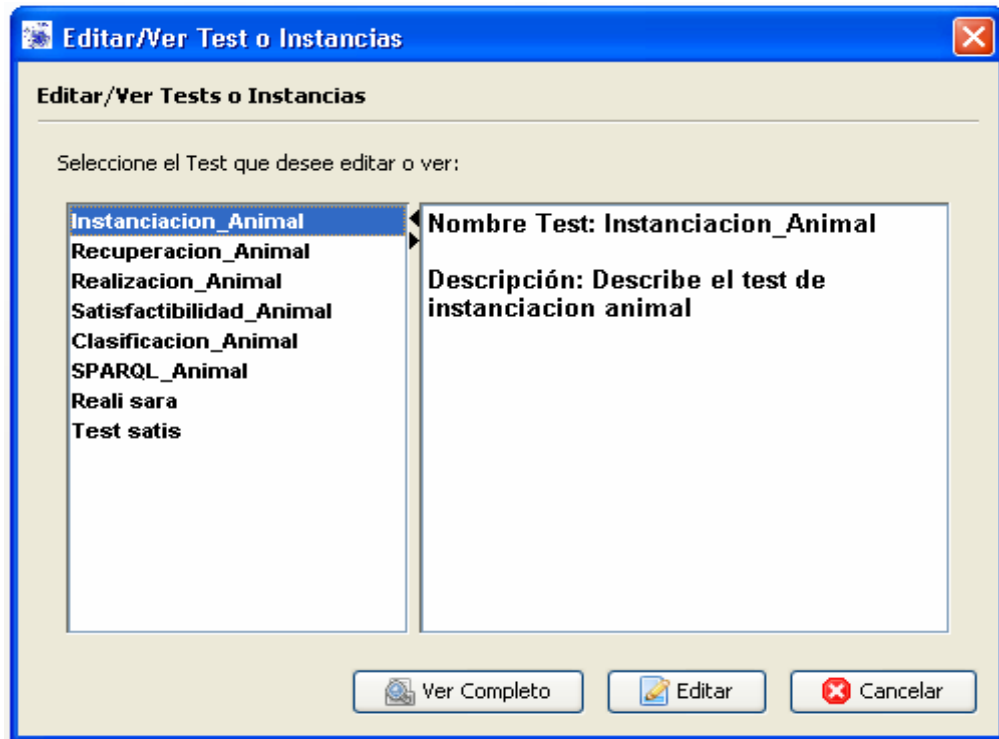


Figura 85 Selección del test a editar

### 10.1.7 Ver un test

Para poder ver un test completo, seleccionaremos, desde el menú principal Tests→Ver, tal y como muestra la Figura 86.

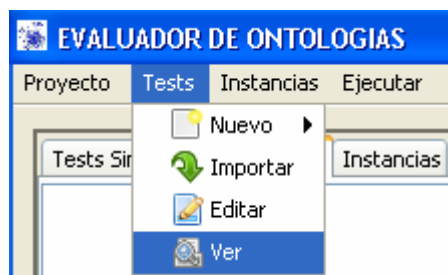


Figura 86 Ver Test

Esta acción abrirá la pantalla que se ve en la Figura 85 Selección del test a editar, y nos permitirá, entre otras acciones, ver el test. La vista del test será similar a la que se muestra en la Figura 83 Vista completa de un test.

#### 10.1.8 Crear instancias

Para crear un conjunto de instancias, seleccionaremos en el menú principal Instancias→Nuevo, tal y como se ve en la Figura 87.



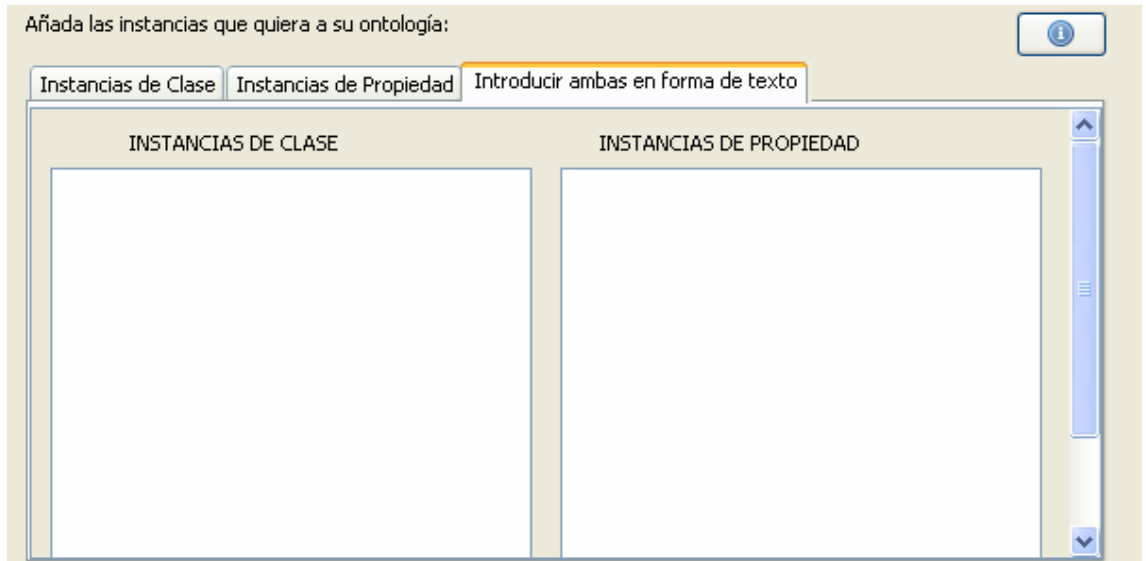
**Figura 87 Crear nuevo conjunto de instancias**

Esta acción abrirá en la zona de la izquierda de la aplicación una nueva plantilla para crear instancias, que se ve en la Figura 88.



Para el conjunto de instancias introduciremos un nombre único e identificativo (campo obligatorio), una descripción (opcional), y el conjunto de instancias, bien de clase o de propiedad, en sus respectivas pestañas indicativas.



**Figura 88 Creación de un conjunto de instancias**


Será posible introducir las instancias en cajas de texto y de forma menos amigable, tal y como muestra el detalle de la Figura 89, algo que para un usuario más experto y especializado puede resultar útil.

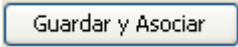



**Figura 89 Detalle del proceso de creación de instancias**

Con el botón  se limpiará la casilla de las instancias seleccionadas mediante .

Con el botón  se borrarán las instancias seleccionadas mediante .

Con el botón  se guardarán las instancias en la sesión de trabajo.

Con el botón  se guardarán y asociarán las instancias en la sesión de trabajo.

Con el botón  se asociarán las instancias a un test. Con ésta acción accederemos a una nueva pantalla que nos pedirá que seleccionemos el test al cual queremos asociar éstas instancias, tal y como muestra la Figura 90.

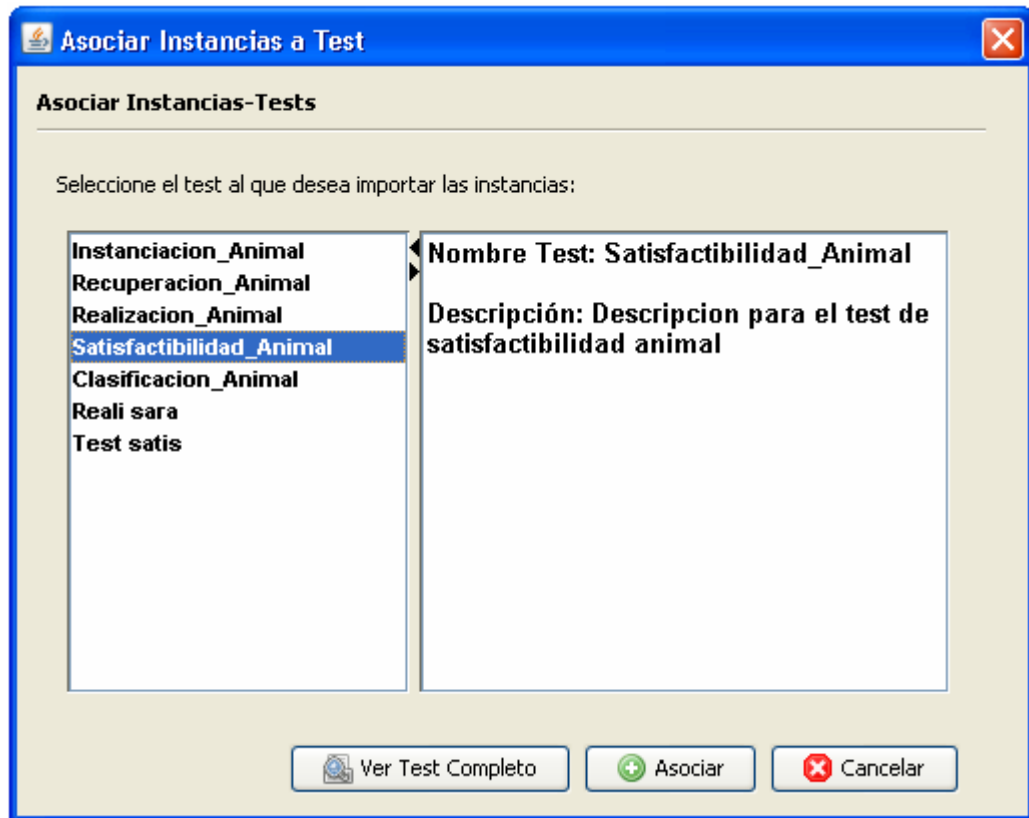

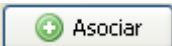
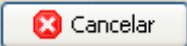


Figura 90 Asociar instancias a un test

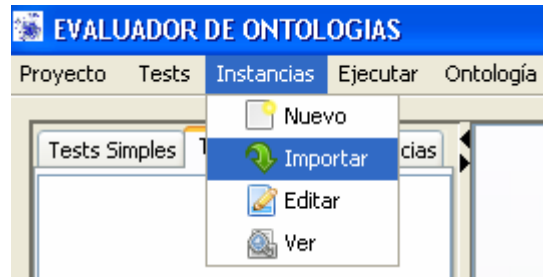
En esta pantalla tendremos la opción de ver una descripción del test, mediante

, de asociar las instancias, mediante , o de cancelar .

#### 10.1.9 Importar Instancias

Para importar instancias desde otros tests debemos de seleccionar Instancias→Importar en el menú principal de la aplicación, tal y como muestra la Figura 91.





**Figura 91 Importar instancias**

Esta acción se comporta igual que 10.1.5 Importar un Test, sólo que en vez de trabajar con tests, trabaja con instancias.

#### *10.1.10 Editar instancias*

Para editar un conjunto de instancias, seleccionaremos del menú principal la opción Instancias→Editar, tal y como muestra la Figura 92.



**Figura 92 Editar instancias**

Esta opción se comporta igual que 10.1.6 Editar un test, sólo que abriendo las instancias para su edición, en vez de los tests.

#### *10.1.11 Ver Instancias*

Para poder ver la descripción de un conjunto de instancias, seleccionaremos, desde el menú principal Instancias→Ver, tal y como muestra la Figura 93.



**Figura 93 Ver instancias**

Esta acción se comporta igual que 10.1.7 Ver un test, excepto que muestra las instancias almacenadas en la sesión de trabajo con su descripción, para que se seleccione cuál se desea ver.

En el panel izquierdo aparecerán las instancias guardadas en la sesión de trabajo, y en el derecho una descripción, que consistirá en las instancias de clase y de propiedad asociadas (ver Figura 94).

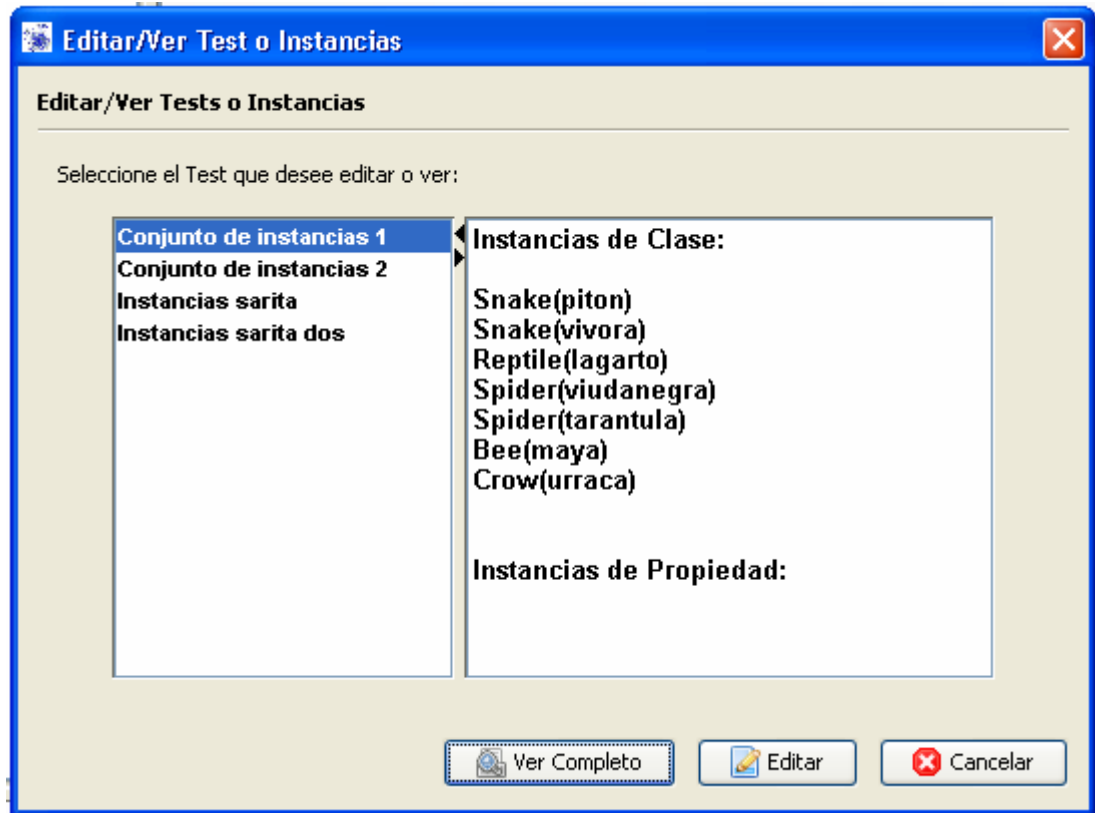


Figura 94 Selección del conjunto de instancias a Ver

La vista de las instancias puede verse en la Figura 95.

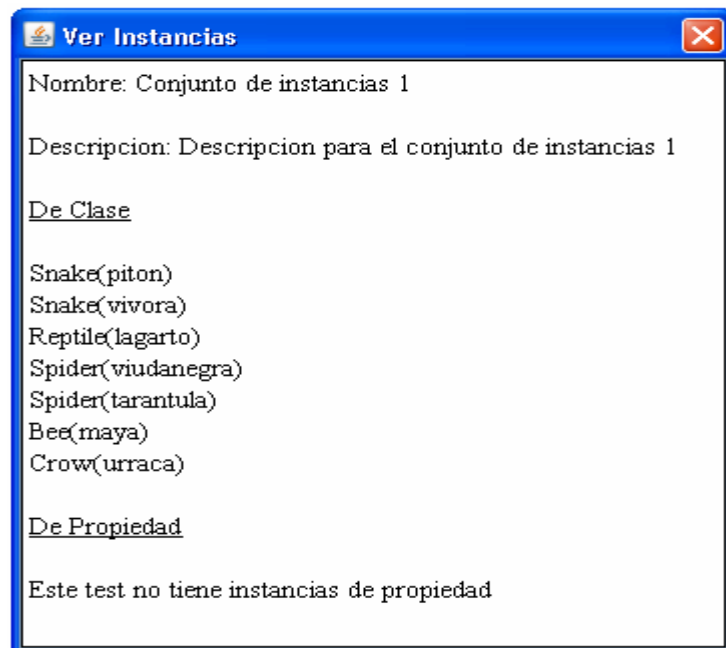


Figura 95 Vista de un conjunto de instancias

#### 10.1.12 Ejecutar un test / Conjunto de tests

Desde el menú principal, podemos seleccionar que tests queremos ejecutar mediante Ejecutar→Seleccionar Test o Ejecutar→Seleccionar Conjunto de Tests, tal y como muestra la Figura 96.



**Figura 96 Ejecutar tests**

Si elegimos la opción de “Seleccionar test”, se nos abrirá una nueva pantalla (ver Figura 97) en la que se nos preguntará qué tests de los almacenados en la sesión de trabajo queremos ejecutar.

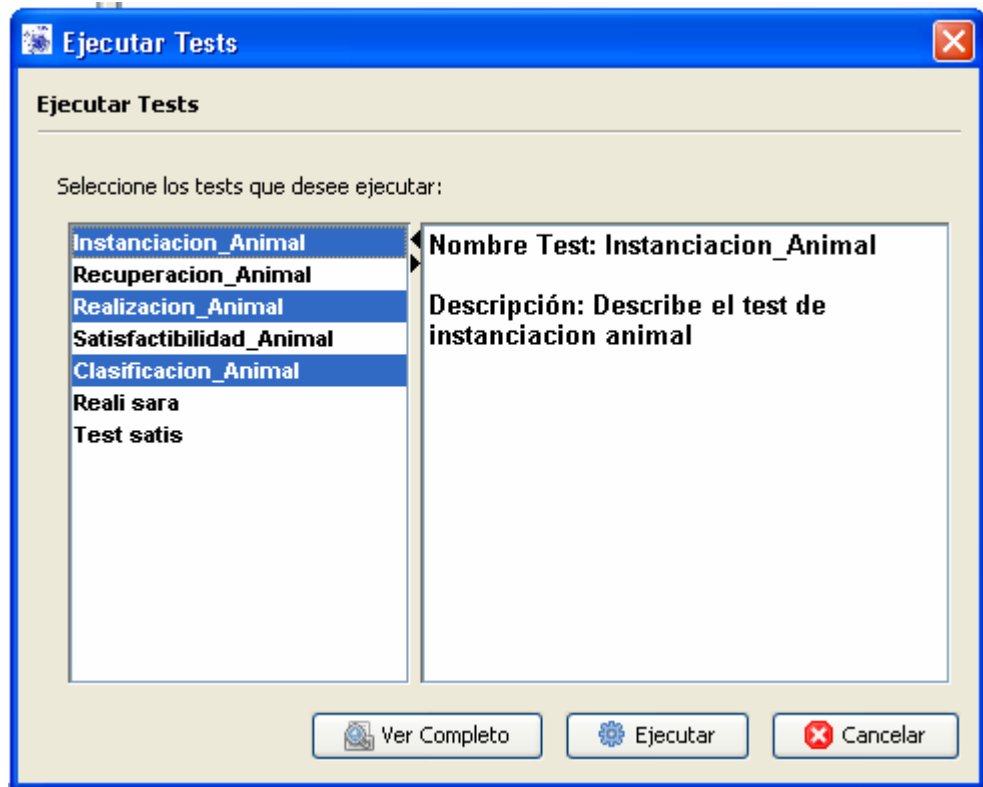
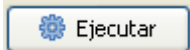
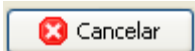


Figura 97 Selección de los tests a ejecutar

Podremos seleccionar uno o varios tests para su ejecución. A la derecha se nos muestra una breve descripción del test.

Con el botón  podemos ver el test completo.

Con el botón  ejecutamos los tests seleccionados.

Con el botón  cancelamos la acción.

Si por el contrario, en el menú principal, seleccionamos la opción “Todos los Tests”, se ejecutarán automáticamente todos los tests almacenados en la sesión de trabajo.

### 10.1.12.1 Resultado de la ejecución de los tests

Una vez ejecutado un test, se añadirán dos nuevas pestañas a la aplicación, una en la parte que contiene la lista de tests e instancias, y otra en la parte inferior de la aplicación, tal y como se ve en la Figura 98. Por cada ejecución que se realice una nueva pestaña contadora de los resultados de ese test será añadida.

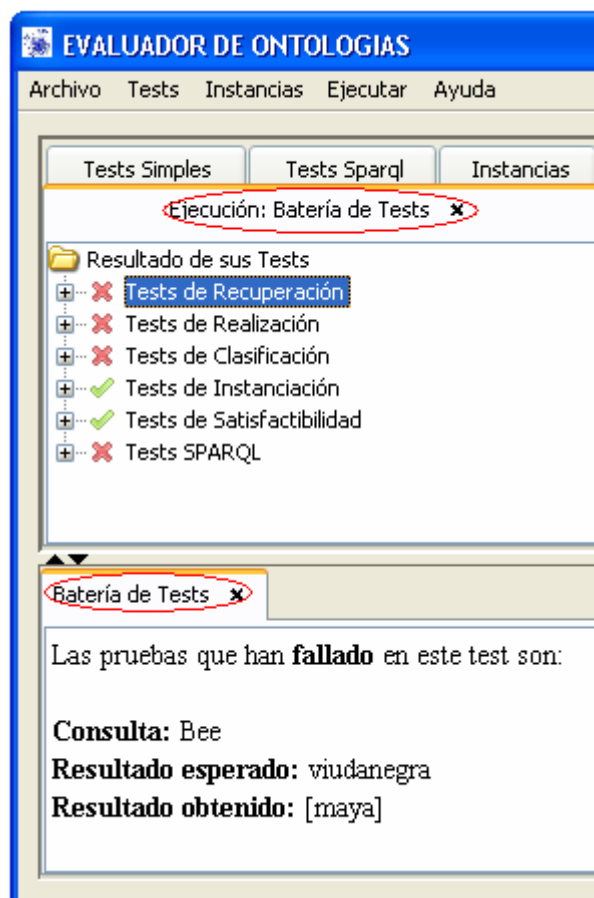
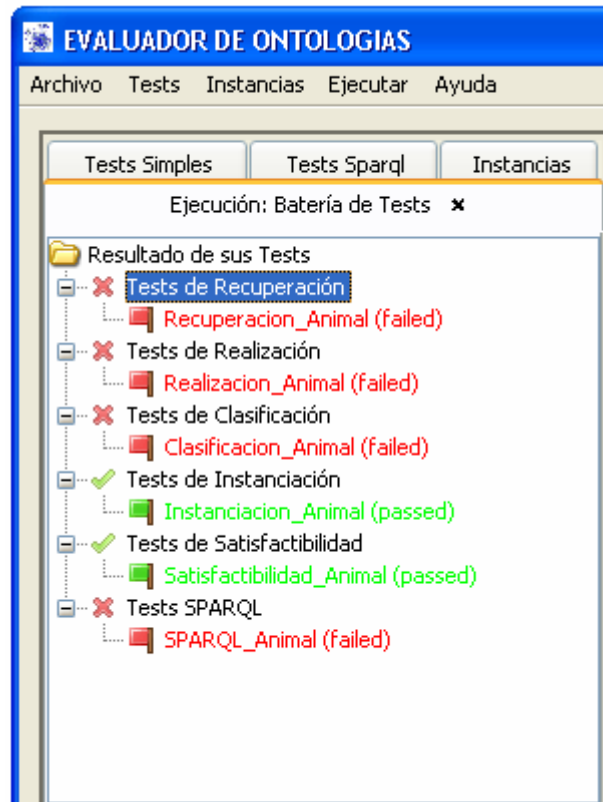


Figura 98 Vista de la aplicación tras la ejecución de los tests

En la parte superior aparecerán, en forma de árbol, todos los tests que han sido ejecutados. En color rojo se mostrarán aquellos que han fallado y en verde los que han sido correctos (ver Figura 99).



**Figura 99** Árbol resultado de la ejecución de los tests

Seleccionando un test del árbol, en la parte inferior de la aplicación se mostrarán los resultados del mismo, indicando qué consultas han fallado (indicando el resultado esperado y el obtenido) y cuáles han sido correctas (ver Figura 100).

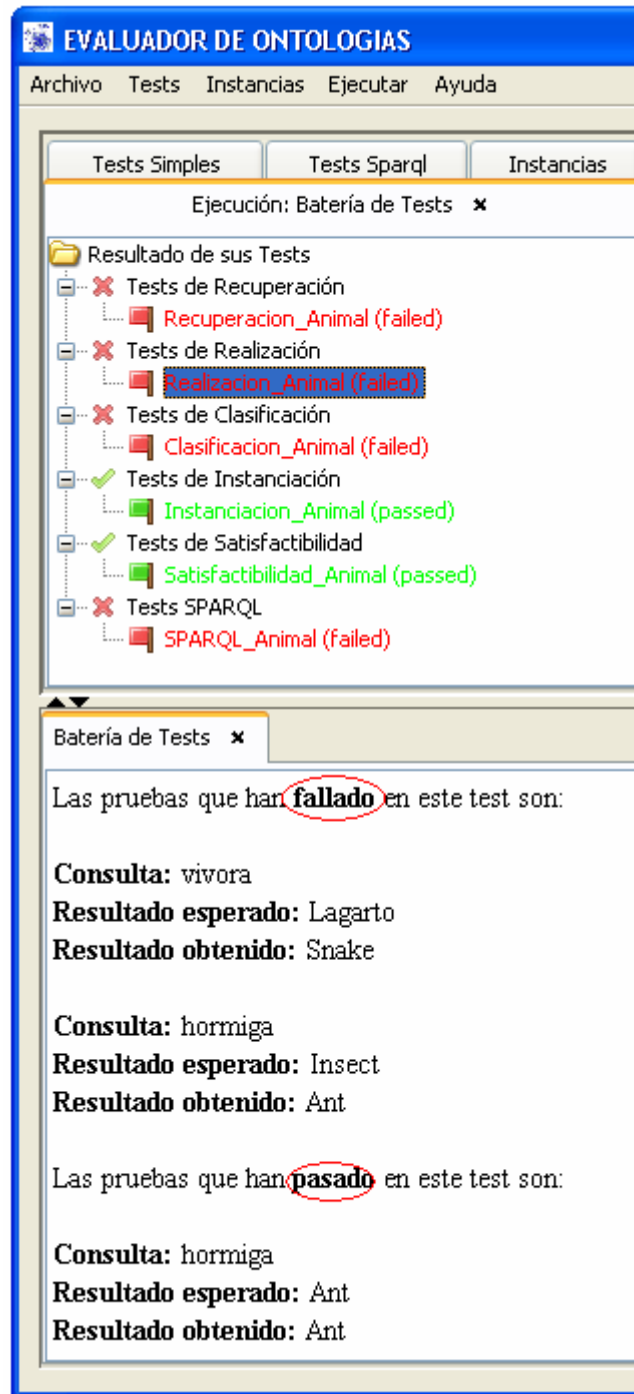


Figura 100 Resultado de la ejecución para un test



### 10.1.13 Ontología

Desde el menú principal, podemos ver la descripción de la ontología con la que estamos trabajando mediante Ontología→Ver. (Ver Figura 101).



Figura 101 Ver Ontología

### 10.1.14 Idiomas

Desde el menú principal, podemos cambiar el idioma de la aplicación, seleccionando el Español, el Inglés (Gran Bretaña) o el Inglés (Estados Unidos), a través de Idiomas. (Ver Figura 102).

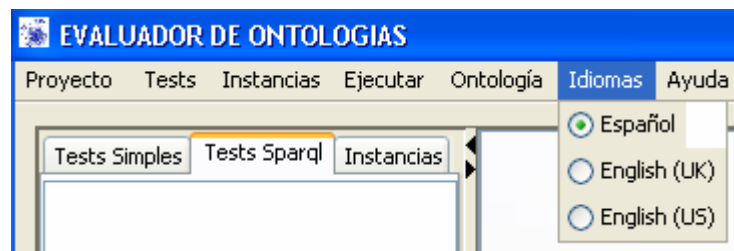


Figura 102 Seleccionar Idioma

## 10.2 Guardar el Proyecto

Desde el menú principal de la aplicación, tenemos acceso a dos formas distintas para guardar el proyecto en local: Archivo→Guardar y Archivo→Guardar como (ver Figura 103).



**Figura 103 Guardar Proyecto**

Si seleccionamos la primera opción, Guardar, el proyecto se guardará en el directorio que el usuario indicó a la hora de abrir o crear el proyecto, con el nombre que le dio al proyecto y en formato .xml.

Si seleccionamos la segunda opción, Guardar como, aparecerá un cuadro de diálogo que nos pedirá que seleccionemos la ubicación y el archivo con extensión .xml en la que queremos guardar el proyecto. Este caso aparece representado en la Figura 104, donde el proyecto se guardará en el directorio “Proyecto Ontologías” con el nombre *ProyectoOntologiasPrueba.xml*.

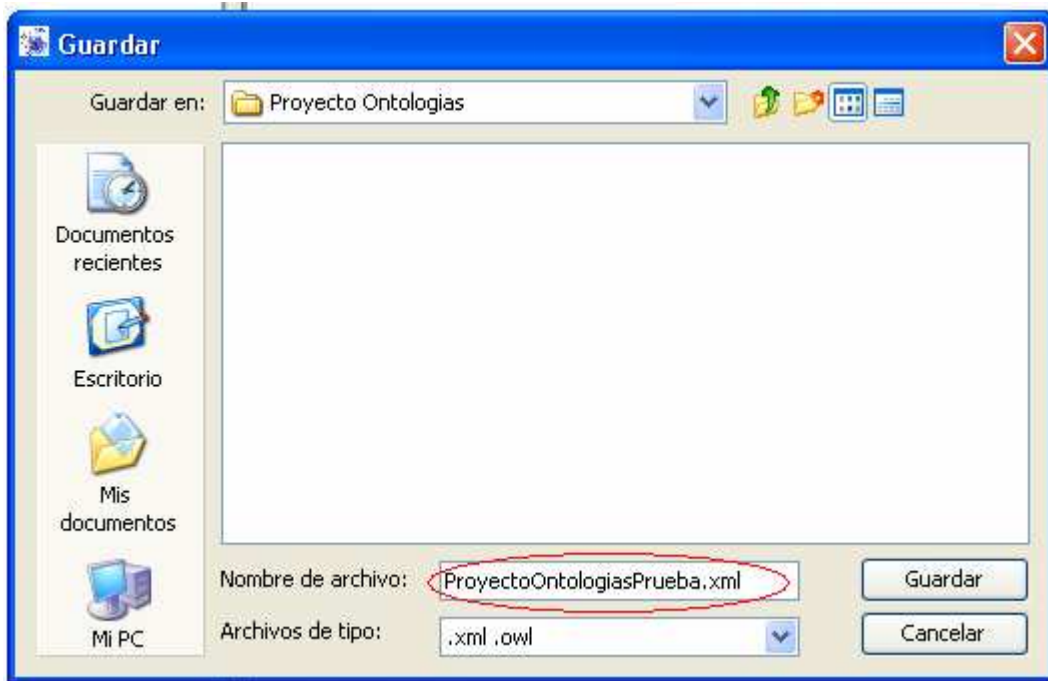


Figura 104 Guardar un proyecto en un directorio específico

### 10.3 Eliminar Proyecto

Desde el menú principal de la aplicación, tenemos acceso a la posibilidad de eliminar el proyecto abierto: Archivo→Eliminar. Con esta opción eliminaremos el archivo xml que representa el proyecto así como la carpeta que lo contiene.

### 10.4 Cerrar Proyecto

Desde el menú principal de la aplicación, tenemos acceso a la posibilidad de cerrar el proyecto abierto: Archivo→Cerrar. Con esta opción cerraremos el proyecto.

## 11 Bibliografía

- [Beck04]: K. Beck y C. Andres. Extreme programming explained. Addison-Wesley Professional segunda edición (26 de noviembre, 2004). ISBN: 978-0321278654.
- [Cockburn04]: A. Cockburn. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley Professional (29 de octubre, 2004). ISBN: 978-0201699470.
- [code.google.com]: <http://code.google.com/p/ontologytesting/>
- [Cohen06]: J. Cohen. Best Kept Secrets of Peer Code Review (Modern Approach. Practical Advice.). Smartbearsoftware.com (2006). ISBN: 1599160676
- [Corch et al.]: ODEVAL: A tool for evaluating rdf(s), daml+oil, and owl concept taxonomies.
- [DSDM03]: DSDM Consortium y J. Stapleton. DSDM: Business Focused Development. Pearson Education segunda edición (13 de enero, 2003). ISBN: 978-0321112248.
- [Edna Ruckhaus] Lógicas Descriptivas y Ontologías  
[<http://www.cs.man.ac.uk/~horrocks/Slides/>]
- [Elliott02]: J. Elliott, R. Eckstein, M. Loy, D. Wood y B. Cole. Java *Swing*. O'Reilly Media, Inc. segunda edición (1 de noviembre, 2002). ISBN: 978-0596004088
- [Hart et al. 05]: Methods for ontology evaluation  
[<http://www.starlab.vub.ac.be/research/projects/knowledgeweb/KWeb-Del-1.2.3-Revised-v1.3.1.pdf>]
- [kaon2]: <http://kaon2.semanticweb.org>
- [Kroll03]: P. Kroll y P. Kruchten. The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP. Addison-Wesley Professional (April 18, 2003). ISBN: 978-0321166098.
- [netbeans.org]: <http://www.Netbeans.org/kb/articles/matisse.html>
- [one-jar]: <http://one-jar.sourceforge.net/>

[O’Cooner07]: J. O’Cooner. Improve Application Performance With *SwingWorker* in Java SE 6. Sun Microsystems (enero, 2007).

<http://java.sun.com/developer/technicalArticles/javase/Swingworker/>

[Palmer02]: S. R. Palmer y J. M. Felsing. A Practical Guide to Feature-Driven Development. Prentice Hall PTR (21 de febrero, 2002). ISBN: 978-0130676153.

[Parsia,Sirin] Bijan Parsia and Evren Sirin. Pellet: An OWL DL reasoner

[pellet]: <http://clarkparsia.com/pellet>

[Ruckhaus05]: Edna Ruckhaus Lógicas Descriptivas y Ontologías. Universidad Simón Bolívar.

[Schwaber04]: K. Schwaber. Agile Project Management with Scrum. Microsoft Press (10 de marzo, 2004). ISBN: 978-0735619937.

[sun.com c]: <http://java.sun.com>

[sun.com d]: <http://java.sun.com/javase/6/docs/api/>

[sun.com e]: <http://java.sun.com/docs/books/tutorial/reallybigindex.html>

[w3.org]: <http://www.w3.org/2007/09/OWL-Overview-es.html#s1.3>