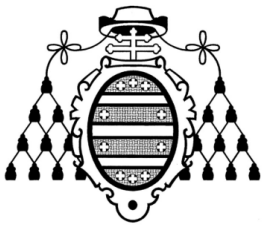


UNIVERSIDAD DE OVIEDO

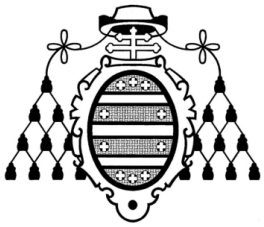
Máster en Ingeniería Web

TRABAJO FIN DE MÁSTER

TRIOO, estudio e implementación de modelos de datos RDF en lenguajes orientados a objetos

Sergio Fernández López

UNIVERSIDAD DE OVIEDO



MÁSTER EN INGENIERÍA WEB

TRABAJO FIN DE MÁSTER

TRIOO, estudio e implementación de modelos de datos RDF en lenguajes orientados a objetos

VºBº del Director

DIRECTORES: **José Emilio Labra Gayo**
Diego Berrueta Muñoz

AUTOR: **Sergio Fernández López**

RESUMEN

El proyecto trioo tiene como objetivo desarrollar una tecnología que permita de una manera lo más sencilla posible utilizar datos RDF directamente desde lenguajes orientados a objetos, permitiendo que el origen y la forma de los datos no tenga consecuencias para los diseños orientados a objetos, a la vez que la semántica de los datos se refleje de la manera más fiel posible. l objetivo no es sólo utilzar y persistir datos en bases de datos RDF, sino tambiñen consumir datos disponibles en la Web y exponer los modelos de negocio de las aplicaciones como Linked Data. La industria del software ya ofrece desde hace varios año de soluciones que abordan el mismo problema sobre otro tipo de modelos de datos, por ejemplo el relacional. Pero todavía no hay una solución equivalente para RDF, debido a que muchas de las actuales alternativas están demasiado inspiradas o basadas en sus homologas para las bases de datos relacionales. Y el modelo de RDF tiene muchas particularidades semánticas que deben tratarse como tal. Por tanto, desde nuestro punto de vista, es un reto muy atractivo tratar de proveer una tecnología para manejar adecuadamente datos RDF de una forma orientada a objetos.

PALABRAS CLAVE

Semántica, RDF, SPARQL, Linked Data, lenguajes orientados a objetos, persistencia.

Más detalles en http://trioo.wikier.org/

ABSTRACT

The trioo project aims to develop a technology able to easily manage RDF data directly from object-oriented programming languages, allowing the usage of that data without negative influences in object-oriented designs and trying to keep the semantics of data as accurate as possible. The objective is not only to persist that data on RDF stores, but also to consume data available on the Web and to expose the business model of applications as Linked Data. The software industry already offers some solutions since some years ago for relational databases. But it is clear that there is not any satisfactory solution to work with RDF. Because many of the current alternatives are strongly rely on inherited design patterns too close to the entity-relational model. And the RDF model has many special features and semantics that should be treated as such. Therefore, form our point of view, provide a technology capable for managing RDF data in a object-oriented way is a appealing challenge.

KEYWORDS

Semantics, RDF, SPARQL, Linked Data, Object-oriented languages, persistence.

Further details at http://trioo.wikier.org/

"The important thing is not to stop questioning.
Curiosity has its own reason for existing."

Albert Einstein

AGRADECIMIENTOS

Aunque suene a tópico, este trabajo no hubiera sido posible sin el apoyo de mucha gente. Empezando por mis directores Diego y Labra, por haber confiado en mi de nuevo. A toda la gente que me ha hechado una mano, tanto técnica como científicamente. Mis compañeros de trabajo por apoyarme siempre. Mis amigos por comprenderme estos dos duros años. Y a mi familia, que siempre ha confiado en mi.

De verdad, gracias a todos.

LICENCIA

1 http://creativecommons.org/licenses/by/3.0/es/

HISTORIAL DEL DOCUMENTO

Fecha	**Versión**	**Comentarios**
Octubre 2009	0.1	Primer esqueleto de la memoria
Febrero 2010	0.2	Añadido el estado del arte
Marzo 2010	0.3	Primera versión del análisis de los modelos computacionales
Mayo 2010	0.4	Revisión de los primeros capítulos
Junio 2010	0.5	Borrador
Julio 2010	1.0	Versión definitiva

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

LISTA DE ACRÓNIMOS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CURIE	Compact URI
CRUD	Create, Read, Update, Delete
CWA	Closed World Assumption
FOAF	Friend of a Friend
HTTP	Hypertext Transfer Protocol
ICSOFT	International Conference on Software and Data Technologies
JDO	Java Data Objects
JPA	Java Persistence API
JSP	JavaServer Pages
JVM	Java Virtual Machine
N3	Notation 3
MVC	Model-View-Controller
NAF	Negation as Failure
ORM	Object-Relational Mapper
OWA	Open World Assumption
OWL	Web Ontology Language
PHP	PHP Hypertext Preprocessor
POJO	Plain Old Java Object
RoR	Ruby on Rails
RDF	Resource Description Framework
RDFa	RDF in attributes
RDFS	RDF Schema
SQL	Simple Query Language
SPARQL	SPARQL Protocol and RDF Query Language
SPARUL	SPARQL Update Language
URI	Uniform Resource Identifier

W3C World Wide Web Consortium

WebDAV Web-based Distributed Authoring and Versioning

XML Extensible Markup Language

1 INTRODUCCIÓN

ÍNDICE

A lo largo de este capítulo se introduce la problemática que ha motivado la presente investigación. Se propone una tesis y se plantean los objetivos principales que debe abordarse para comprobar su validez. Posteriormente se presenta brevemente la organización general de la memoria.

1.1 INTRODUCCIÓN

La persistencia, en software, es un atributo de los datos que asegura su disponibilidad incluso más allá de la vida de una aplicación. Se trata por tanto de un problema común en cualquier desarrollo software. Aunque existen diferentes aproximaciones para resolver este problema, en los últimos años la tendencia de la industria ha ido hacia la linea de dotar a los diferentes lenguajes orientados a objetos de mecanismos para facilitar enormemente la persistencia sobre las bases de datos más habituales (relacionales, XML[1], orientadas a objetos, etc). En este sentido RDF[2] ofrece un potente marco sobre el que representar e intercambiar descripciones de recursos en la Web. La mayoría de lenguajes de programación disponen de bibliotecas para el acceso y manipulación de RDF, pero dado que se tratan de bibliotecas construidas alrededor del modelo de datos RDF, su integración con los lenguajes del programación orientados a objetos rompe drásticamente la filosofía de estos. A pesar de que en los últimos años han aparecido algunas propuestas que han intentado solucionar este problema, aún existe un gran vacío tecnológico. Este trabajo tratará de desarrollar una tecnología que, atendiendo a las particularidades de cada implementación del modelo orientado a objetos, resuelva la persistencia a RDF de una manera eficaz y estándar.

1.2 PROBLEMA

Los datos son fundamentales en cualquier aplicación software. Lo que convierte la persistencia de esos datos en algo clave en cualquier desarrollo. Hasta ahora, cuando el esquema de datos estaba perfectamente definido, se encontra-

1 Extensible Markup Language
2 Resource Description Framework

ba en las bases de datos relacionales un excelente soporte para realizar esta persistencia. Pero a pesar su avanzado desarrollo y su amplia adopción por prácticamente la totalidad de la industria, los almacenes de datos relacionales tradicionales presentan una serie de inconvenientes cuando el esquema de datos ya no está tan definido, o sufre de constantes evoluciones. Es precisamente por eso que en los últimos años la industria del software ha tendido hacia soluciones flexibles en cuanto al esquema de información a almacenar. Por un lado la persistencia a XML soluciona alguno de estos problemas de la flexibilidad de esquema, con el uso de un modelo de datos bien conocido y asimilado por la industria, aunque con algunas limitaciones que hacen que no alcance a ser una solución del todo satisfactoria en muchos casos. Pero la actividad en este frente esta siendo frenética en los últimos años, motivada en buena medida por los nuevos requerimientos, sobre todo, de aplicaciones Web. Estas nuevas aplicaciones Web tiene que dar cabida a unos requerimientos de disponibilidad y carga de trabajo jamás imaginables en entornos tradicionales. Basta pararse a buscar las cifras del volúmenes de datos que manejan algunas de estas aplicaciones Web, como por ejemplo Facebook o GMail, para darse cuenta que los conceptos que tradicionalmente se han venido usando no son válidos en tales escenario. En este sentido la iniciativa NoSQL ("Not only SQL" [3]), se la que se hablará con mucho más detalle en el estado del arte, concentra buena parte de la nueva generación de bases de datos que no dan soporte al modelo relacional, permitiendo una mayor escalabilidad horizontal en entornos distribuidos.

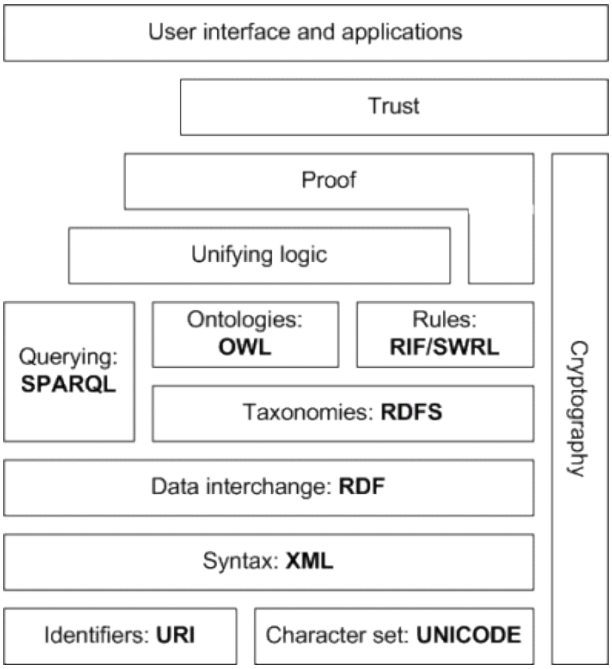


Figura 1: Pila de tecnologías de la Web Semántica.

Alternativamente el Consorcio W3C[4][5] propone otra aproximación, a menudo conocida como la Web Semántica [15]. La Web Semántica no pretende proveer

3 http://nosql-database.org/
4 World Wide Web Consortium
5 http://www.w3.org/

uno o varios productos concretos que solucionen este u otros problemas concretos, no, sino que tiene una aproximación más generalista. La Web Semántica aspira a proveer una serie de tecnologías (véase la Figura 1) que faciliten las implementación de soluciones a este, y otros, problemas, basándose en una arquitectura descentralizada, la Web, la Web de datos ('*Web of Data*" originalmente en inglés). Obviamente esta aproximación obliga a tener en cuenta otro tipo de cosas que no se tienen en cuenta en aproximaciones tradicionales, pues la información que puede encontrarse en la Web puede ser incompleta o incorrecta [82], pero este tipo de cuestiones quedarían fuera del ámbito de este trabajo de investigación.

Recapitulando, la Web Semántica pretende construir sobre la Web actual un ecosistema en el que distintos agente software puedan publicar y consumir datos. Utiliza por tanto HTTP[6] [42] como protocolo básico sobre el que transferir información, y XML [23] como formato básico para la codificación de documentos, todo ello enriquecido con la pila tecnológica de RDF. RDF [59] (*Resource Description Framework*) provee un modelo de datos extensible y extremadamente flexible. Basado en el concepto básico de "tripleta" `(sujeto, predicado, objeto)`, RDF simplifica drásticamente la interoperabilidad, intercambio e integración de datos desde fuentes y aplicaciones heterogéneas. RDF extiende el concepto de enlace de la Web tradicional, con el uso de URIs[7] para dar nombre a las relaciones entre cosas en lugar de entre documentos [9, 94]. Sobre este marco básico que provee RDF, la llamada *Web of Data* ofrece un nuevo paradigma para consumir datos disponibles y enlazados en la Web [11] (Linked Data). Linked Data está creciendo muy rápido, y ya existe una ingente cantidad de datos RDF disponibles y listos para ser usados por nuevas aplicaciones. Por tanto se abre un nuevo horizonte para la integración de la Wold Wide Web con los datos y aplicaciones de la empresa. Por estas características RDF puede ser una tecnología clave a medio plazo para realizar la persistencia en las aplicaciones software, convirtiéndose en un back-end que pueda unificar el acceso a datos para las aplicaciones. Esta arquitectura se ve complementada con lenguajes para el modelado del conocimiento (RDFS[8] [25] y OWL[9] [97]), y lenguajes para la consulta de este modelo de datos (SPARQL[10] [86]), entre otros. Precisamente en los momentos que se redacta esta memoria, el W3C acaba de lanzar un nuevo grupo de trabajo para desarrollar la nueva versión del lenguaje[11] que mejora ciertas lagunas de la versión actual e incluya nuevas características[12] , como por ejemplo SPARUL[13] [96] o los lenguajes de rutas [98].

Pero a pesar de los beneficios que este paradigma ofrece, el tener que familiarizarse con tecnologías como HTTP [42], URI [9], tripleta, SPARQL, etc..., no facilita para nada su adopción por la comunidad de desarrolladores. Por tanto es necesario ofrecer una tecnología que permita usar de una manera flexible los datos almacenados en RDF, pero sin por ello romper el diseño ni la dinámica orientada a objetos de las aplicaciones actuales. Después de todo "son objetos, no tripletas"[14]. Por tanto parece obvio que decir que el modelo de datos RDF podría tener buen encaje con los lenguajes orientados a objetos. Aunque

6 Hypertext Transfer Protocol
7 Uniform Resource Identifiers
8 RDF Schema
9 Web Ontology Language
10 SPARQL Protocol and RDF Query Language
11 http://www.w3.org/2001/sw/DataAccess/
12 http://www.w3.org/2009/sparql/wiki/Category:Features
13 SPARQL Update Language
14 http://harth.org/andreas/blog/2009/03/27/its-objects-not-triples/

en esta fase tan temprana del proyecto una afirmación así se antoja demasiado aventurada, por lo que será necesario esperar a los resultados del análisis de ambos modelos computacionales para conocer su posibilidades reales de integración. Aunque el ideal sería llegar a dar un soporte nativo a RDF desde los propios lenguajes de programación[15], eso ya dependerá de la implementaciones concreta que cada lenguaje de programación ofrezca del modelo orientado a objetos.

1.3 TESIS

La tesis detrás de trioo se enuncia como:

> *Es posible integrar modelos de datos basados en RDF con el modelo de aplicaciones software desarrolladas bajo el paradigma de la orientación a objetos, manteniendo la semántica del código cercana a la semántica de los datos y realizando la persistencia de forma nativa en RDF con un bajo acoplamiento a un fabricante concreto mediante el uso de estándares.*

1.4 OBJETIVOS

Por tanto el proyecto tiene el objetivo principal enunciado en la tesis:

- Integrar modelo de datos en RDF en lenguajes orientados a objetos.

Un objetivo de alto nivel quizá demasiado abstracto, pero que en realidad se materializa en una serie de objetivos mucho más concretos:

- Comprender las diferencias semánticas que hay tras ambos modelos computacionales.
- Aislar las diferencial irreconciliables entre ambos modelos, de manera que se pueda acotar el alcance del proyecto.
- Integrar mecanismos basados en estándares para el acceso a datos RDF en los mecanismos convencionales que ya se utilizan en los distintos lenguajes de programación.
- Proveer de implementaciones de referencia en algunos de los paradigmas de programación orientados a objetos más utilizados.

Lo que puede servir para ayudar a los organismos estandarizadores a extender e integrar la familia de tecnologías RDF con otras tecnologías más industriales y extendidas, proveyendo de nuevos casos de uso como ya se esta haciendo con otras tecnologías [85]. Y, si los resultados del proyecto son lo suficientemente usables, publicar las herramientas desarrolladas, aunque el foco del proyecto no esta en producir aplicaciones finales, sino utilizar estas implementaciones como medio para validar la investigación y encontrar futuros caminos por lo que pudiera continuar esta linea investigadora.

15 http://lists.w3.org/Archives/Public/public-lod/2010Jan/0118.html

1.5 ORGANIZACIÓN DEL DOCUMENTO

La presente memoria se organiza de la siguiente forma: primeramente el Capítulo 2 hace un profundo repaso sobre la tecnología y técnicas relevantes para el propósito del proyecto. A continuación en el Capítulo 3 se realiza un análisis de los modelos computacionales, por un lado el orientado a objetos y por otro el de RDF, a fin de detectar convergencias y divergencias entre ambos modelos computacionales que será ponderadas en el Capítulo 4. Estas conclusiones sirven como punto de partida para el diseño e implementación de los dos prototipos descritos a lo largo del Capítulo 5. Por último, en el Capítulo 6 se describen las conclusiones a las que se ha llegado con la realización de este trabajo de investigación, planteando una serie de preguntas que pudieran guiar los siguientes pasos a tomar para continuar la linea iniciada.

Adicionalmente, en los anexos se recoge el resto de información relevante en esta memoria, como las publicaciones científicas realizadas y las referencias al trabajo relacionado que se han consultado para la elaboración del mismo.

2 ESTADO DEL ARTE

ÍNDICE

Dado que la persistencia de datos es algo necesario en la mayoría de las aplicaciones software, a lo largo de los años la ingeniería ha desarrollado diferentes soluciones software para realizar esta tarea de sobre diferentes tipos de almacenes (*back-ends*). Dada la popularización en los últimos veinte años de los lenguajes de programación orientados a objetos, se ha realizado un esfuerzo importante para dotar al programador de tecnologías más cercanas a ese paradigma, de manera que la tarea resulte más liviana e incluso transparente en algunos casos.

El trabajo descrito en esta memoria se encuentra íntimamente relacionado con el trabajo hecho estos últimos años en el mapeo de objetos a bases de datos relacionales, pero también con el trabajo para el acceso a datos publicados en la Web (Web Semántica o Web de Datos). Para alcanzar los objetivos de este capítulo, primeramente se introducirán una serie de tecnologías y técnicas relevantes y necesarias para la comprensión del mismo.

2.1 PATRONES

Lo primero antes de comenzar a analizar el estado del arte, es conocer cómo funciona y se resuelve el problema de la persistencia en las aplicaciones software modernas. Por tanto es necesario conocer y comprender los patrones de diseño que hay tras la mayoría de estas herramientas. Y quizá los más relevantes sean dos: *Active Record* y *Data Mapper*. Aunque probablemente estos dos patrones de diseño sean los más relevantes y usados para el mapeo de objetos a datos (y viceversa), evidentemente hay muchos más, como por ejemplo "*Table Data Gateway*" o "*Row Data Gateway*", entre otros. Incluso alguno más alejados de la persistencia, pero con problemas muy comunes, como por ejemplo puede ser el "*Data Transfer Object*". Sin embargo, dada la extensa bibliografía que ya existe acerca de estos temas [46, 3], en este estado del arte no se pretende elaborar un detallado informe acerca de todos estos patrones.

El propósito es más introducir al lector en las soluciones comunes al problema, de manera que sea más sencilla la comprensión y asimilación de las características de las herramientas que se analizarán más adelante. Para ello sólo nos detendremos con algo de calma en los dos patrones más relevantes.

2.1.1 Active Record

Active Record es un patrón de diseño muy utilizado para trabajar con bases de datos que utiliza la aproximación más obvia: *coloca* la lógica de acceso a datos directamente en el objeto de dominio. Con Active Record un objeto es representado como una fila de una tabla en una base de datos relacional [46]. Por tanto Active Record es una aproximación de acceso a bases de datos: cada tabla (o vista) se *envuelve* como una clase, y cada objeto instancia se liga a una fila concreta de la tabla. La asunción de "*un objeto, una fila*" simplifica enormemente la programación de funciones CRUD[1] en bases de datos relacionales. El patrón estrictamente sólo define como mapear clases a tablas y objetos a filas, pero no resuelve como mapear las relaciones entre objetos; eso es algo que se resuelve de diferentes formas en cada implementación. Y cabe destacar que Active Record es el patrón detrás de la inmensa mayoría de las implementaciones actuales de ORM[2], como veremos más adelante en este capítulo.

2.1.2 Data Mapper

Los objetos y las bases de datos relacionales tienen diferentes mecanismos para estructurar los datos. Muchas partes de un objeto, como las colecciones o las relaciones de herencia, no están presentes en las bases de datos de manera nativa. Cuando se realiza un diseño orientado a objetos complejo, es muy recomendable el uso de este tipo de mecanismos para organizar mejor tanto los propios datos como su ciclo de vida. Cuanto más estrictamente se aplique el diseño orientado a objetos, más tienden a diferenciarse el esquema de objetos del esquema relacional. Aún así se necesitan transferir datos entre ambos esquemas, lo cual puede llegar a ser complejo; pero si los objetos en memoria son conscientes de la estructura de la base de datos, cambios en uno de los dos esquemas tiene un reflejo en el otro.

El patrón de diseño *Data Mapper* [46] (una especialización del patrón *Mapper*)provee una capa software para separar la representación de los objetos en memoria de la base de datos. Su responsabilidad es transferir los datos entre ambos esquemas, aislándolos entre ellos. Con este patrón los objetos no necesitan ni siquiera conocer que existe una base de datos; y por tanto no necesitan ni código ni conocer detalles de sobre su estructura para hacer las consultas SQL[3].

2.2 TECNOLOGÍAS

Aunque el ámbito del proyecto trioo se restringe únicamente al acceso a datos RDF, uno de los objetivos de este estado del arte es conseguir extraer el mayor número posible de conclusiones, tanto negativas como positivas, de

1 Create, Read, Update, Delete
2 Object-Relational Mapper
3 Simple Query Language

Figura 2: Patrón Data Mapper

cualquiera de las soluciones actuales. Por tanto, este estado del arte no se restringirá únicamente a analizar las soluciones para el acceso a datos RDF, sino que cubrirá un abanico mayor de soluciones y aproximaciones, aprendiendo de otras tecnologías más maduras como son las relacionales, a fin de alcanzar la mejor solución posible en trioo . Por tanto, será necesario darle una repaso a algunas de las tecnologías más relevantes para el desarrollo del proyecto: el modelo entidad-relación [30], SQL [45], XML [23], RDF [59], SPARQL [86], etc... Para ello se realizará un análisis detallados de las herramientas que se han considerado relevantes, probando de manera real la mayoría de ellas. Se confía que de esta manera se disponga ya de una visión más completa del problema y de las distintas aproximaciones que pueden existir para darle solución.

2.3 HERRAMIENTAS

El concepto de persistencia de objetos directamente a diferentes formas (especialmente a bases de datos relacionales) ha sido implementado por diferentes productos software. En esta sección se tratará de hacer un repaso de los más relevantes, tanto por su popularidad como por el posible impacto que pudieran tener en los resultados de este trabajo. Cada uno será analizado con detalle, tratando de sacar a la luz sus virtudes y defectos, y aprendiendo de todos ellos para el desarrollo de la tecnología de trioo .

2.3.1 Persistencia sobre almacenes relacionales

Las bases de datos relacionales son quizás la tecnología más desarrollada, fiable y rápida que actualmente se dispone para guardar datos. Dadas las limitaciones de modelo relacional, guardar objetos en bases de datos relacionales requiere renunciar a la libertad de relaciones y esquema, reduciendo las habilidades del desarrollador a la hora de diseñar y escribir código orientado a objetos. Pero dada el apabullante dominio de este tipo de bases de datos, la industria del software ha realizado un esfuerzo por tratar de minimizar estas deficiencias, tratando de reducir estas limitaciones para los programadores. Por tanto existen un gran número de productos y tecnologías susceptibles de analizar en este área concreta de la persistencia de objetos.

ActiveRecord

Active Record es el patrón en que se ha basado el desarrollo del ORM sobre el que se ha construido el popular framework de desarrollo Web *Ruby on Rails*[4] (RoR[5]). Esta implementación se ajusta perfectamente a la filosofía de

4 http://rubyonrails.org/
5 Ruby on Rails

RoR de obtener mucha funcionalidad con poco código, y la idea ha sido implementada en muchos otros lenguajes y frameworks. De manera poco original los desarrolladores de RoR decidieron llamar también ActiveRecord a su implementación[6]; por tanto es fácil que se produzcan más conflictos con el nombre de otras herramientas, como por ejemplo sucede con Castle ActiveRecord[7] para .NET. La Figura 3 ilustra con un pequeño ejemplo el uso de ActiveRecord en RoR[8].

```
require 'active_record'

ActiveRecord::Base.logger = Logger.new(STDERR)
ActiveRecord::Base.colorize_logging = false

ActiveRecord::Base.establish_connection(
    :adapter => "sqlite3",
    :dbfile  => ":memory:"
)

ActiveRecord::Schema.define do
    create_table :persons do |table|
        table.column :name, :string
        table.column :nick, :string
    end
end

class Person < ActiveRecord::Base
    has_many :friends, :class_name => 'Person'
end

person = Person.create(:name => 'Sergio', :nick => 'wikier')
person.friends.create(:name => 'Diego', :nick => 'berrueta')
person.friends.create(:name => 'Jose', :nick => 'labra')
person.save

puts Person.find(1).friends.length
puts Person.find_by_name('Diego').nick
```

Figura 3: Sencillo ejemplo del ActiveRecord de RoR.

Django Models

De igual forma que su homologo RoR, Django[9] provee los modelos (Models[10]), un componente para facilitar la persistencia de los objetos de aplicaciones Django a bases de datos relacionales. En Django un modelo es la fuente de datos última y única del modelo de datos de una aplicación, dado que contiene los campos y el comportamiento necesarios. Por normal general cada modelo en Django de mapea a una tabla de una base de datos relacional. Aunque también se pueden usar en otras capas de la aplicación, como por ejemplo la vista. A continuación se muestra brevemente el uso de este componente para realizar la persistencia de un objeto en Python (Figura 4).

6 http://rubyforge.org/projects/activerecord/
7 http://www.castleproject.org/activerecord
8 Señalar que RoR utiliza configuración por convención, por tanto la tabla a la que se mapea una clase se compone mediante el plural del nombre de la clase en Ruby.
9 http://docs.djangoproject.com/
10 http://docs.djangoproject.com/en/dev/ref/models/instances/

```
>>> from django.db.models import Model, CharField

>>> class Person(Model):
>>>     name = CharField()

>>> person = Person.objects.create(name="Sergio")
>>> person.save()
```

Figura 4: Ejemplo del uso de los modelos de Django.

Django trata de ser extremadamente sencillo, tratando de dar una solución simple para la mayor parte de los problemas. Pero es precisamente esa sencillez lo que a menudo se le achaca para desaconsejar su uso en determinados problemas en los que se requiere algo más de flexibilidad[11].

GORM

GORM[12] es el ORM utilizado en el framework de desarrollo Grails[13], de ahí precisamente su nombre: Grails' Object Relational Mapping. En realidad no es una implementación totalmente nueva, sino que se trata de un envoltorio desarrollado en Groovy [91, 61] que por debajo en realidad utiliza Hibernate, A pesar de que el capricho de los nombres ha querido que por orden alfabético aparezca primero en este estado del arte, quizá sea recomendable realizar antes la lectura sobre Hibernate para conocer en más detalle Hibernate. Pero dada la naturaleza dinámica de Groovy, permite trabajar tanto con tipado dinámico con estático, y mediante el nombrado por convención hacer cosas como las mostradas en la Figura 5 de una forma mucho más ágil que haciéndolas por configuración.

```
>>> def book = Book.findByTitle("Groovy in Action")

>>> book
...   .addToAuthors(name:"Dierk Koenig")
...   .addToAuthors(name:"Guillaume LaForge")
...   .save()
```

Figura 5: Uso de GORM en Grails.

Hibernate

Aunque Hibernate[14] originalmente era una implementación del patrón Data Mapper para el lenguaje Java, actualmente es mucho más, es un completo framework para el mapeo de de modelos de datos orientados a objetos a bases de datos relacionales [6]. Como se puede ver en la Figura 6, dentro de este framework de acceso a datos para Java, Hibernate provee un stack completo de tecnologías, entre ellas un lenguaje, conocido como Hibernate Query Language (HQL), que permite realizar consultas sobre el modelo de datos orientado a objetos con una sintaxis similar al popular SQL, y que con

11 http://adam.gomaa.us/blog/2007/aug/26/the-django-orm-problem/
12 http://www.grails.org/GORM
13 http://www.grails.org/
14 http://www.hibernate.org/

la estandarización de JPA[15] ha derivado hacia JPQL, Java Persistence Query Language.

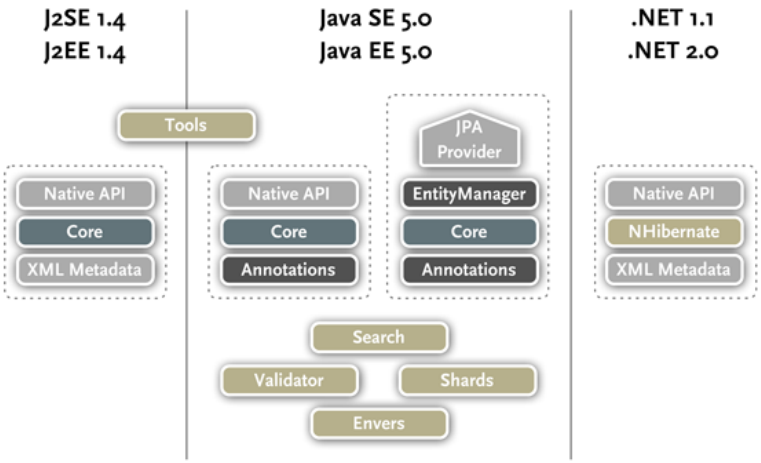


Figura 6: Pila de tecnologías de Hibernate.

De hecho, el proyecto Hibernate ha sido uno de los gérmenes de la especificación JPA (JSR220 [37] y JSR317 [36]), junto con otros fabricantes con productos similares, como Oracle con TopLink (liberado parcialmente como EclipseLink). Una especificación no sin polémica, pues aunque aparentemente intenta facilitar la migración a diferentes implementaciones, siempre y cuando se utilice sólo ese denominador común amparado por la especificación; en realidad cada producto quiere seguir manteniendo su ventaja competitiva ofreciendo determinadas extensiones a que complican el cambio. Aún así es innegable la popularidad que ha tomado JPA en los últimos aún, pero los nuevos productos no han movido a Hibernate de esa posición de liderazgo que ocupa desde hace años dentro del mundo Java. Incluso el proyecto ha realizado un *port* de algunos componentes al framework de Microsoft .NET, NHibernate[16].

MyBatis

MyBatis[17], el nombre bajo el que ahora se conoce al proyecto iBATIS[18] es una implementación en Java, .NET y Ruby del patrón de diseño Active Record, proveyendo un framework de persistencia de datos que automatiza el mapeo entre objetos y bases de datos relacionales. En este caso no se sigue la semántica de un ORM, sino que se ha optado por una aproximación diametralmente opuesta a la utilizada por otras herramientas, por ejemplo Hibernate: iBatis automatiza la creación de POJOs (Plain Old Java Objects) desde esquema relacionales ya existentes. Por tanto se adapta mejor al desarrollo de aplicaciones sobre bases de datos heredadas, que a la persistencia relacional de nuevos diseños orientados a objetos.

15 Java Persistence API
16 https://www.hibernate.org/343.html
17 http://www.mybatis.org/
18 http://ibatis.apache.org/

```
<mapper namespace="org.example.PersonMapper">
  <select id="getPerson"
          resultType="org.example.Person"
          parameterType="org.example.Person"
  >
    select * from people where id=#{id} and name=#{name}
  </select>
</mapper>
```

Figura 7: Ejemplo de mapeo de consultas con MyBatis

Para su uso se requiere *mapear* los POJOs a consultas SQL, tal y como se muestra en la Figura 7. Una vez realizado este *mapeo* ya se pueden ejecutar operaciones. La Figura 8 ilustra un ejemplo muy sencillo para obtener el mapper y ejecutar una sentencia con el API[19] de MyBatis.

```
SqlSession session = sessionFactory.openSession();
PersonMapper personMapper = session.getMapper(PersonMapper.
    class);
Person person = personMapper.getPerson();
```

Figura 8: Ejemplo básico de uso de MyBatis.

SQLAlchemy

SQLAlchemy[20] es un toolkit SQL para Python que provee un ORM para facilitar la persistencia de objetos a los desarrolladores de aplicaciones. Se trata de una biblioteca extremadamente flexible que da respuesta a un gran abanico de problemas.

Históricamente, como se muestra en la Figura 9, SQLAlchemy requería de la definición de las tablas y las clase por separado, realizando el mapeo entre ambas definiciones del modelo de datos. Esta separación tiene la clara ventaja de poder afinar detalles de aspectos concretos de cada representación del modelo de datos: por un lado la definición de los objetos que forman parte del modelo de dominio, y por otro la definición de las tablas sobre las que se va a realizar la persistencia. Aunque en muchos casos puede no requerirse un grado de detalle tan fino. Y como viola el principio DRY[21], el desarrollador debe ser consciente de la existencia de ambas definiciones a la hora de realizar una modificación en el modelo de datos. Como se puede ver en la Figura 10, en las últimas versiones de SQLAlchemy ya se ha incluido una forma alternativa para esta tarea, que el proyecto denomina *declarativa*.

2.3.2 Persistencia sobre almacenes orientados objetos

En este estado del arte no debiera dejarse sin cubrir los almacenes de datos orientadas a objetos. Lamentablemente, a pesar de la pujanza de esta tecnología en sus orígenes, este tipo de almacenes no gozan de una gran cuota en el mercado actual. Además, la mayoría de productos que dicen ser bases de datos de objetos no son más que meros subterfugios que finalmente almacenan los

19 Application Programming Interface
20 http://www.sqlalchemy.org/
21 Don't Repeat Yourself, no te repitas

```
>>> from sqlalchemy import create_engine

>>> engine = create_engine("sqlite:///:memory:")

>>> from sqlalchemy import Table, Column,String, MetaData
>>> metadata = MetaData()
>>> people_table = Table("people", metadata, Column("name",
    String))

>>> metadata.create_all(engine)

>>> class Person(object):
...     def __init__(self, name):
...         self.name = name

>>> from sqlalchemy.orm import mapper
>>> mapper(Person, people_table)

>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=engine)

>>> person = Person("Sergio")
>>> session.add(person)
```

Figura 9: Ejemplo del uso clásico de SQLAlchemy.

```
>>> from sqlalchemy import Column,String
>>> from sqlalchemy.ext.declarative import declarative_base

>>> Base = declarative_base()
>>> class Person(Base):
...
...     __tablename__ = "people"
...     name = Column(String)
...
...     def __init__(self, name):
...         self.name = name

>>> people_table = Person.__table__
>>> metadata = Base.metadata
```

Figura 10: Ejemplo del uso declarativo de SQLAlchemy.

objetos en bases de datos relacionales o similares, con los problemas y limitaciones que esto conlleva, tal y como se ha visto anteriormente. Por tanto en este apartado del estado del arte sólo se cubrirán algunas soluciones de persistencia nativa de objetos.

Apache JDO

JDO [92, 93] es una especificación para persistir objetos Java. Aunque existe cierto solapamiento con JPA (visto en la Sección 2.3.1), JDO[22] pretende ser una tecnología agnóstica respecto a la forma final que tiene la base de datos en la que se persisten los objetos. Por otro lado JDO requiere siempre de un contenedor de EJBs para funcionar.

22 Java Data Objects

Aunque existen varias implementaciones de JDO , Apache JDO[23] es la implementación de referencia de JDO 2.0. En la Figura 11 se ve cómo se utilizaría el API, y en la Figura 12 la configuración de los metadatos necesarios para persistir un POJO[24] con JDO (la configuración general iría aparte).

```
Person person = new Person();
...

PersistenceManagerFactory factory =
JDOHelper.getPersistenceManagerFactory("name");
PersistenceManager manager = factory.getPersistenceManager();
Transaction tx = manager.currentTransaction();
tx.begin();
manager.makePersistent(person);
tx.commit();
manager.close();
```

Figura 11: Uso de Apache JDO.

El estándar sigue evolucionando, y la versión 2.3 ya se encuentra en borrador, de la cual AccessPlatform (que se tratará más adelante) es ahora la implementación de referencia.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="example.jdo">
    <class name="Person" identity-type="application"
        objectidclass="PersonKey">
      <field name="name" primary-key="true" />
      <field name="nick"/>Java Persistence API (
    </class>
  </package>
</jdo>
```

Figura 12: Metadata para JDO.

Dobbin

Dobbin[25] se trata de una base de datos transaccional orientada a objetos para Python, que permite persistir objetos a disco con una muy reducida sobrecarga, incluso en escenarios con varios hilos accediendo a los objetos de la aplicación. La motivación del proyecto es proveer una alternativa a ZODB (que se verá a continuación) pura en Python, y que sea más ligera y sencilla de integrar en los diseños orientados a objetos. Su uso se ilustra brevemente en la Figura 13.

Durus

Durus[26] [16] es un sistema de persistencia de objetos para aplicaciones escrito en Python. Durus ofrece una forma sencilla de usar, mantener y persistir colecciones de objetos con soporte transaccional. Además de su formato de

23 http://db.apache.org/jdo/
24 Plain Old Java Object
25 http://pypi.python.org/pypi/dobbin
26 http://www.mems-exchange.org/software/durus/

```
>>> from dobbin.persistent import Persistent
>>> person = Persistent()
>>> person.name = "Sergio"

>>> from dobbin.database import Database
>>> import transaction
>>> db = Database(database_path)
>>> db.elect(person)
>>> transaction.commit()
```

Figura 13: Sencillo ejemplo de uso de Dobbin.

persistencia nativo, existe la posibilidad de utilizar otros formatos, como por ejemplo bases de datos relacionales[27]. Su uso puede verse en la Figura 14.

```
>>> from durus.file_storage import FileStorage
>>> from durus.connection import Connection

>>> connection = Connection(FileStorage("db.durus"))
>>> root = connection.get_root()
>>> person = Person()
>>> person.name = "Sergio"
>>> root["person"] = person
>>> connection.commit()
```

Figura 14: Ejemplo de uso del API de Durus.

ZODB

ZoDB[28] es una base de datos de objetos nativa para objetos Python. Originalmente desarrollada para dar soporte a la persistencia al servidor de aplicación Zope, ZODB permite almacenar los objetos sin importar el paradigma de programación que se utilice en Python. En la Figura 15 se ilustra con un pequeño ejemplo el su uso.

```
>>> from persistent import Persistent

>>> class Person(Persistent):
...     (...)
...
>>> from ZODB.FileStorage import FileStorage
>>> from ZODB.DB import DB

>>> storage = FileStorage("data.fs")
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()
>>> root["person-1"] = Person()
```

Figura 15: Ejemplo básico de uso de ZODB.

27 http://www.jcea.es/programacion/durus-berkeleydbstorage.htm
28 http://www.zodb.org/

2.3.3 Persistencia sobre almacenes libres de esquema

Dentro de esta clasificación podrían encajarse almacenes basados en XML o RDF, pero dada su relevancia para este trabajo de investigación, ambos se tratarán con mayor profundidad en siguientes secciones.

Respecto a la persistencia libre de esquema, es necesario destacar la iniciativa NoSQL. NoSQL, abreviatura de "Not only SQL"[29], concentra buena parte de la nueva generación de bases de datos que no dan sólo soporte al modelo relacional, con el fin de permitir una mayor escalabilidad horizontal en entornos distribuidos. Se parte del teorema de CAP [24], en el cual Eric Brewer enuncia que es imposible que un servicio Web asegure las tres propiedades que forman su acrónimo en inglés: ***C**onsistency, **A**vailability, **P**artition tolerance*. Por tanto estas bases de datos se mueven del tradicional concepto de ACID [89] (acrónimo en inglés de ***A**tomicity, **C**onsistency, **I**solation, **D**urability*) predominante en los motores relacionales, a un concepto diametralmente opuesto: BASE [83] (acrónimo de ***B**asically **A**vailable, **S**oft state, **E**ventually consistent*). El cambio fundamental está en la mentalidad: mientras ACID es intrínsecamente pesimista y fuerza a chequear la consistencia después de realizar cada operación, BASE es de una naturaleza mucho más optimista, asumiendo que el nivel de consistencia de los datos puede fluctuar en el tiempo. Aunque esto parezca descabellado, es algo perfectamente admisible, y permite escalar hasta umbrales jamás alcanzables con ACID. Y toda esta filosofía ha sido acuñada bajo el término NoSQL. Bajo el cual se agrupan tipos muy distintos de bases de datos: de columnas variables (Google BigTable [28], Pregel [66], BitVault [110], Hadoop HBase[30], Cassandra[31], Hypertable[32]), de documentos (CouchDB[33], MongoDB[34]), clave-valor (Amazon SimpleDB[35], Amazon Dynamo[36], MemcacheDB[37]), basadas en grafos (Neo4J[38], AllegroGraph[39], BigData[40], OpenLink Virtuoso[41]), etcétera. Cabe destacar la evidente intersección que existe entre las tecnologías RDF y algunos de los productos bajo esta iniciativa, particularmente las bases de datos basadas en grafos, muchas de las cuales proveen soporte nativo para SPARQL.

2.3.4 Persistencia sobre almacenes XML

En los últimos años XML [23] se ha convertido en el formato estándar de transferencia de datos. Mucho más allá de su mero uso en la Web, XML trataba de convertirse en el framework básico sobre el que se han definido diversos formatos de intercambios de datos para diferentes dominios (legales, económicos, etc). El desarrollo de tecnologías como JAXB [57] permite mapear estructuras XML con objetos del lenguaje de programación (el extracto de XML Schema de la Figura 16 con la clase de la Figura 17 por ejemplo).

29 http://nosql-database.org/
30 http://hadoop.apache.org/
31 http://incubator.apache.org/cassandra/
32 http://hypertable.org/
33 http://couchdb.apache.org/
34 http://www.mongodb.org/
35 http://aws.amazon.com/simpledb/
36 http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
37 http://memcachedb.org/
38 http://www.neo4j.org/
39 http://www.franz.com/agraph/
40 http://www.systap.com/bigdata.htm
41 http://www.openlinksw.com/

```
<xsd:simpleType name="GroupType">
  <xsd:restriction base="xsd:int">
    <xsd:minInclusive value="1" />
    <xsd:maxInclusive value="255" />
  </xsd:restriction>
</xsd:simpleType>
```

Figura 16: Fragmento de un XML Schema.

```
public class ElementType {

    @XmlAttribute(name="Group", required=true)
    protected int group;

    // ...

    public int getGroup() {
        return group;
    }

    public void setGroup(int value) {
        this.group = value;
    }

    // ...
}
```

Figura 17: Clase Java con anotaciones de JAXB.

Pero a pesar de la existencia y madurez en el mercado de este tipo de tecnologías, las bases de datos nativas XML no han terminando de despegar. Y no será por ausencia de opciones donde escoger: eXist[42], Tamino[43], Apache Xindice[44], BaseX[45], MarkLogic Server[46], entre otras; incluso bases de datos relacionales como Oracle[47] o SQLServer[48] han incluido soporte para XML. En paralelo W3C ha producido un lenguaje de consulta para modelos de datos XML, XQuery [19], que dispone de implementaciones en diferentes lenguajes (como por ejemplo para Java con la JSR225 [76], o Zorba[49] para C++, con bindings Python). Aún así estas bases de datos no han terminando de salto como alternativa de mercado para el almacenamiento de datos.

42 http://exist.sourceforge.net/
43 http://www.softwareag.com/Corporate/products/wm/tamino/default.asp
44 http://xml.apache.org/xindice/
45 http://basex.org/
46 http://www.marklogic.com/product/marklogic-server.html
47 http://www.oracle.com/technology/tech/xml/xmldb/index.html
48 http://msdn.microsoft.com/en-us/library/ms345117%28SQL.90%29.aspx
49 http://www.zorba-xquery.com/

2.3.5 Persistencia sobre almacenes RDF

ActiveRDF

ActiveRDF[50] (Eyal Oren et al. [75]) propone una novedosa forma de trabajar sobre RDF con orientación a objetos, basándose en el ORM de RoR ActiveRecord, y por tanto basado en el patrón del mismo nombre ActiveRecord [46] y fuertemente influenciado por el trabajo en el desarrollo Web dirigido por modelos realizado en los últimos años. Pero el desarrollo de ActiveRDF está quizás demasiado focalizado en su desarrollo para un lenguaje concreto (en este caso Ruby), además de tener que utilizar interfaces propietarias de cada back-end para realizar la persistencia, dado que cuando fue desarrollado SPARQL no tenía características de modificación.

```
1 require 'active_rdf'
2
3 ConnectionPool.add_data_source(:type => :sparql, :results =>
      :sparql_xml,
4  :url => "http://example.org/sparql")
5 Namespace.register :foaf, 'http://xmlns.com/foaf/0.1/'
6
7 person = RDFS::Resource.new 'http://www.wikier.org/foaf#
      wikier'
8 puts person.foaf::name
9
10 ObjectManager.construct_classes
11 otherperson = FOAF::Person.new 'http://example.org/person'
```

Figura 18: Ejemplo de uso de ActiveRDF

Empire

Empire[51] se trata de una implementación basada en JPA para RDF. Para ello extiende el framework de anotaciones de JPA con nuevas anotaciones para dar soporte a datos RDF, no dando soporte a determinadas características que no tienen ninguna relación con RDF. Provee acceso a diferentes almacenes de datos RDF (actualmente sólo 4Store, Sesame y Jena) usando SPARQL, SeRQL[27] y las APIs propietarias de los almacenes soportados. La Figura 21 ilustra como anotar una clase con Empire.

En definitiva, la aproximación de Empire es muy similar a la que trioo pretende seguir en lenguajes basados en clases, como por ejemplo es el caso de Java. Lamentablemente Empire no existía cuando comenzó este proyecto, y ha sido asimilado en el estado del arte en fases más avanzadas del desarrollo del mismo. Aún así, como elemento diferenciador, cabe destacar que Empire ha optado por utilizar APIs propietarias para la persistencia de datos RDF.

Hercules

Hercules[52] es una pequeña biblioteca que, con una sintaxis muy compacta como se puede ver en la Figura 20, permite la consulta de endpoints SPARQL y el mapeo de los datos a Javascript.

50 http://www.activerdf.org/
51 http://github.com/clarkparsia/Empire
52 http://www.arielworks.net/works/codeyard/hercules

```
@Entity
@Namespaces({"foaf", "http://xmlns.com/foaf/0.1/"})
@RdfsClass("foaf:Person")
public class Book implements SupportsRdfId {

    @RdfProperty("foaf:name")
    private String name;

    @RdfProperty("foaf:friend")
    @OneToMany(fetch = FetchType.LAZY,
               cascade = {CascadeType.PERSIST, CascadeType.
                   MERGE})
    private Collection<Person> friends = new HashSet<Person
        >();

}
```

Figura 19: Ejemplo de uso de Empire.

```
var e = new Hercules("http://uri/of/sparql/server");
e.registerPrefix("foaf", "http://xmlns.com/foaf/0.1/");

var sergio = e.find("http://example.org/sergio");
alert(sergio.getObject("foaf:name"));

var people = e.foaf$Person.findAll();
for (person in people) {
    alert(person.getObject("foaf:name"));
}
```

Figura 20: Ejemplo de código con Hercules.

Al respecto de lo ilustrado con esta herramienta, es impresionante la flexibilidad que ofrece un lenguaje como Javacript para realizar determinando tipo de cosas.

JenaBean

JenaBean[53] es un biblioteca que persiste (y lee) POJOs a modelos de Jena[54] en Java. Aunque parcialmente ha movido parte de su API a la forma de anotaciones Java, requiere que los objetos de modelo de dominio hagan herencia por implementación de una clase (RdfBean) que sirve como plantilla a las operaciones básicas, lo cual implica una restricción muy importante a la hora de integrar esta herramienta en cualquier diseño orientado a objetos mínimamente complejo.

Moriarty

Moriarty[55] es una biblioteca para acceder al API[56] de la plataforma de Talis[57] desde el lenguaje PHP. Entre sus componentes cuenta con uno llama-

53 http://jenabean.googlecode.com/
54 http://jena.sourceforge.net/javadoc/com/hp/hpl/jena/rdf/model/Model.html
55 http://moriarty.googlecode.com/
56 http://n2.talis.com/wiki/Platform_API
57 http://www.talis.com/platform/

```
@Namespace("http://xmlns.com/foaf/0.1/")
public class Person extends RdfBean<Person> {

    private String name;

    (...)

}

(...)\ac{

OntModel m = ModelFactory.createOntologyModel();
Jenabean.instance().bind(m);

Person person = new Person();
person.setName("Sergio");
person.save();
```

Figura 21: Sencillo ejemplo de cómo anotar una clase Java con JenaBeans.

do DataTable[58]. DataTable es una implementación del patrón Active Record (véase la Sección 2.3.1) que provee acceso mediante SPARQL (por ahora[59] en modo sólo lectura) a los datos RDF de la plataforma de Talis.

```
$dt = new DataTable('http://api.talis.com/stores/mystore');
$dt->map('http://xmlns.com/foaf/0.1/name', 'name');
$dt->map('http://xmlns.com/foaf/0.1/nick', 'nick');

$dt->select('name, nick')->limit(10);
$res = $dt->get();

foreach ($res->result() as $row) {
   echo $row->name;
   echo $row->nick;
}
```

Figura 22: Ejemplo de uso de los DataTable de Moriarty

OntoBroker

OntoBroker no es una simple herramienta de acceso a datos RDF. OntoBroker nación como un conjunto de lenguajes y herramientas que, utilizando ontologías como mecanismo base para la representación de conocimiento, en conjunto provee un servicio de acceso y consulta de información sobre la World Wide Web [40]. Hoy en día OntoBroker[60] se ha convertido en un producto comercial consolidado que provee un eficiente y escalable middleware de razonamiento. Soporta muchas de las tecnologías de la Web Semántica, entre ellas RDF, RDFS, OWL y SPARQL, además de F-logic [58].

Aunque OntoBroker no pueda considerarse una herramienta de acceso a datos RDF convencional como el resto de herramientas estudiadas, ya que su aproximación se encuentra un escalón un poco más alto en cuanto al nivel de abstracción, determinados aspectos de OntoBroker lo hacen de relevancia para este estado del arte.

58 http://code.google.com/p/moriarty/wiki/DataTables
59 A 18 de Diciembre de 2009.
60 http://www.ontoprise.de/en/home/products/ontobroker/

Oort

Oort[61] es una biblioteca que permite el acceso a grafos RDF desde objetos planos en Python (Figura 23) y su publicación como HTML. Lamentablemente es un proyecto discontinuado y sin actividad durante los últimos tres años.

```
>>> from oort.rdfview import *
>>> from oort.util.queries import Annotated

>>> SIOC = Namespace("http://rdfs.org/sioc/ns#")

>>> class Sioc(Annotated):
...     _rdfbase_ = SIOC
...     name = localized()
...     description = localized()
...     seeAlso = each(RDFS) >> Annotated

>>> class Site(Sioc):
...     main_item = one_where_self_is(SIOC.has_host) >> 'Item'

>>> class Container(Sioc):
...     children = each_where_self_is(SIOC.has_parent) >> Sioc

>>> class Item(Sioc):
...     host = one(SIOC.has_host) >> Site
...     parts = each(SIOC.has_part) >> Annotated

# you need an rdflib graph instance and a get_lang callable here
>>> context = QueryContext(graph, get_lang, queries=[Site, Container, Item])

>>> site = context.view_for(URIRef("http://oort.to"))
>>> for item in site.main_item.children:
...     print item.uri, item.name, [p.label for p in item.parts]
```

Figura 23: Uso de Oort.

Penry

Penry[62] es una reimplementación en Java de la biblioteca Moriarty vista anteriormente en la Sección 2.3.5. Una biblioteca que internamente Talis venia utilizando en sus productos para acceder a su plataforma[63], y que liberó a finales de 2008, pero sin un gran esfuerzo por cubrir necesidades más allá de las ya soportadas internamente.

Ripple

Ripple tiene una aproximación muy diferente a lo visto hasta ahora en este análisis. Ripple es un lenguaje relacional de flujo de datos basado en pila para la Web Semántica [100]. Se trata por tanto no de una herramienta en si, sino de un framework muy versátil para la ejecución de algoritmos transversales en

61 http://oort.to/
62 http://code.google.com/p/penry/
63 http://www.talis.com/platform/

redes semánticas. La implementación libre[64] está escrita en Java, y funciona únicamente sobre el API nativa de Sesame. Incluye un intérprete en linea así como una API sobre la que realizar consultas. Dado el amplio abanico de problemas que potencialmente puede cubrir esta herramienta, es difícil poner un ejemplo que ilustre su uso; en la Figura 24 se muestra cómo utilizarlo para recoger y agregar datos como si de un robot RDF se tratase.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

@define foafStep:
    (   id
        owl:sameAs
        foaf:knows
    )/each/i
    /unique.

<http://www.w3.org/People/Berners-Lee/card#i>
    :foafStep 2/times /foaf:name.
```

Figura 24: Sencillo ejemplo de crawler RDF con Ripple.

RDF DataObjects

RDF.rf[65] dispone de un plugin para persistir datos RDF a bases de datos relacionales. Para ello utiliza aproximación muy sencilla y un tanto ingenua, que consiste en almacenar los datos en una tabla de cuatro columnas (sujeto, predicado, objeto, contexto), por lo que no es necesario realizar de ningún mapping más sofisticado. Su uso por tanto es bastante sencillo, tal y como se puede ver en la Figura 25, y no sigue el paradigma orientado a objetos.

```
require 'rdf'
require 'data_objects'
require 'do_sqlite3'

repo = RDF::DataObjects::Repository.new('sqlite3:test.db')
repo.load('http://www.wikier.org//foaf.rdft')

repo.count

any_subject = repo.first.subject
any_subject_statements = repo.query(:subject => any_subject)
repo.delete *any_subject_statements
repo.count
```

Figura 25: Ejemplo de uso de RDF::DataObjects.

RDFobject

RDFobject[66] es un conjunto de clases para facilitar el uso de objetos desde la Web. En realidad es un *wrapper* a RDFlib[67], que facilita una serie de tareas

64 http://ripple.googlecode.com/
65 http://rdf.rubyforge.org/
66 http://pypi.python.org/pypi/RDFobject
67 http://www.rdflib.net/

habituales. Aunque, como se puede ver en el ejemplo de uso de RDFobject de la Figura 26, no comulga demasiado con los principios de Linked Data [11], además de la poca usabilidad del submódulo para manejar los espacios de nombres.

```
>>> from rdfobject import RDFobject
>>> sergio = RDFobject("http://www.wikier.org/foaf#wikier")
>>> len(sergio.triples)
0
>>> sergio.from_url("http://www.wikier.org/foaf")
(...)
    raise NotANamespaceException()
rdfobject.urihelper.NotANamespaceException
>>> sergio.add_namespace("owl", "http://www.w3.org/2002/07/
    owl#")
>>> (...)
>>> sergio.from_url("http://www.wikier.org/foaf")
>>> len(sergio.triples)
2
>>> sergio.from_url("http://www.wikier.org/foaf#wikier")
>>> len(sergio.triples)
136
>>> sergio.list_objects("foaf:name")
[rdflib.Literal(u'Sergio Fern\xe1ndez', lang=u'es')]
```

Figura 26: Ejemplo de utilización de RDFobject.

Pero adicionalmente tiene una característica peculiar: en lugar de utilizar formatos estándar de la Web Semántica para volver a almacenar los objetos recuperados de la Web, RDFobject soporta guardar los objetos a disco utilizando tanto Pairtrees [62] como el modelo de datos propuesto por Fedora [63]; ambas opciones parece interesantes desde el punto de vista de la integración de la Web de Datos con las aplicaciones de escritorio, aunque habría que estudiar con más calma sin son modelos lo suficientemente flexibles como almacenar un modelo de datos RDF.

SemWeb4J

El Semantic Web 4 Java Toolkit[68] se compone de una serie de herramientas para trabajar con la Web Semántica desde Java. En primer lugar RDF2Go[69] es una abstracción de triple (o quad) stores, que permite a los desarrolladores abstraerse de una implementación concreta (por debajo da soporte a diversas APIs propietarias). Por encima se coloca RDFReactor[70], una herramienta que permite realizar vistas de modelos de datos RDF a través de orientación a objetos en Java [108], traduciendo automáticamente una ontología a sus correspondientes clases en Java (Figura 27).

68 http://semweb4j.org/
69 http://rdf2go.semweb4j.org/
70 http://rdfreactor.semweb4j.org/

```
1  package org.example.myproject;
2
3  import org.ontoware.rdf2go.Reasoning;
4  import org.ontoware.rdfreactor.generator.CodeGenerator;
5
6  public class GenerateSomething throws Exception {
7
8     public static void main(String[] args) {
9         CodeGenerator.generate("path/to/data/myschema.n3",
10                               "src/gen",
11                               "org.example.myproject",
12                               Reasoning.rdfs, true, true);
13    }
14 }
```

Figura 27: Generación de clases Java a partir de una ontología con RDFReactor.

Sommer

Sommer[71] es una herramienta para mapear RDF a objetos Java, al más puro estilo JPA o JDO. Aunque como se puede ver en la Figura 28[72] para ello no utiliza anotaciones, sino requiere empotrar determinada lógica de la persistencia en los beans.

71 https://sommer.dev.java.net/

72 El ejemplo completo puede consultarse en https://sommer.dev.java.net/sommer/howto.html

```
1  public class Person implements RdfSerialisable, SommerMapable
       {
2
3      public static String foaf = "http://xmlns.com/foaf/0.1/";
4      protected HashSet sommerNullFieldHash;
5      protected RewriteMapper sommerRewriteMapper;
6
7      public Person() {}
8
9      public void setFirstName(String firstName) {
10         this.javassistSetPerson_firstName(firstName);
11     }
12
13     void javassistSetPerson_firstName(String s) {
14         if (sommerRewriteMapper == null) {
15             firstName = s;
16         } else {
17             try {
18                 Field field = Person.getDeclaredField(
19                 "firstName");
20                 firstName = (String)
21                      sommerRewriteMapper.setField(
22                        java/lang/String, this,
23                        (rdf)field.getAnnotation(
24                           net/java/rdf/annotations/rdf),
25                        s);
26                 if (firstName == null)
27                     sommerNullFieldHash.add(field);
28                 else
29                     sommerNullFieldHash.remove(field);
30             } catch(NoSuchFieldException e) {
31                 e.printStackTrace();
32             }
33         }
34     }
35
36     //...
37
38     public void rdfSerialise(RdfSerialiser rdfserialiser) {
39         Field afield[] = Person.getDeclaredFields();
40         for(int i = 0; i < afield.length) {
41             Field field = afield[i];
42             try {
43                 if(!rdfserialiser.isInterestingField(field))
44                     continue;
45                 Object obj = rdfserialiser.processField(
46                                 this, field,
47                                 field.get(this));
48                 if(obj != null)
49                     field.set(this, obj);
50                 continue;
51             }
52             catch(Exception e) {
53                 Logger.getLogger(Person.getName()).log(Level.
                       SEVERE,
54                          "Illegal Argument: should never happen
                               ", e);
55             } finally {
56                 i++;
57             }
58         }
59
60     }
61
62 }
```

Figura 28: Utilización de Sommer.

El mismo autor propone[73] que se puede reutilizar Jersey[74] (la implementación de referencia de JAX-RS [52]) para serializar objetos Java a RDF.

Spira

Spira[75] se trata, aunque los autores lo denominan ORM Object RDF Mapper), de un DataMapper para RDF desarrollado en Ruby. Al contrario que ActiveRDF, se basa en RDF.rb[76] debido a algunos problemas que los autores ha detectado en los bindings de Redland para Ruby[77]. Su diseño se ha basado en el concepto al que los autores denominan "*open model*", entendido que Spira no espera que una clase sea la definición autorizada, exclusiva y completa del modelo de datos. Ello aleja a Spira de la herencia que otras muchas otras herramientas tienen hacia las bases de datos relacionales, acercando su semántica mucho más a RDF. En la Figura 29 se ilustra de manera muy sencilla su uso.

```
1  require 'spira'
2
3  repo = "http://www.wikier.org//foaf.rdf"
4  Spira.add_repository(:default, RDF::Repository.load(repo))
5
6  class Person
7    include Spira::Resource
8
9    property :name,  :predicate => FOAF.name
10   property :nick,  :predicate => FOAF.nick
11 end
12
13 sergio = RDF::URI("http://www.wikier.org//foaf#wikier").as(
       Person)
14 sergio.name #=> "Sergio Fernandez"
15 sergio.nick #=> "wikier"
16 sergio.save!
```

Figura 29: Ejemplo de uso de Spira

Dada su juventud[78], lógicamente aún presenta muchas limitaciones y errores. Pero aún así se trata de una iniciativa prometedora.

SuRF

SuRF [5][79] se trata de otra implementación del patrón Active Record, proveyendo un mecanismo orientado a objetos para el acceso a tripletas RDF. Al igual que ActiveRDF, proyecto que claramente ha servido de inspiración a SuRF, puede trabar tanto sobre ficheros en disco como base de datos RDF remotas, para lo que utiliza SPARQL para el proceso de lectura de datos, pero implementando las interfaces propietarias para realizar la escritura y/o modificación de información para las base de datos que soporta. En la Figura 30 se recupera un ejemplo sobre el uso de SuRF directamente extraído de su página web.

73 http://blogs.sun.com/bblfish/entry/serialising_java_objects_to_rdf
74 https://jersey.dev.java.net/
75 http://github.com/datagraph/spira
76 http://rdf.rubyforge.org/
77 http://librdf.org/docs/ruby.html
78 la primera versión ha sido publicada en mayo de 2010: http://blog.datagraph.org/2010/05/spira
79 http://surfrdf.googlecode.com/

```
>>> from surf import *

>>> store = Store(  reader='rdflib',
...                 writer='rdflib',
...                 rdflib_store = 'IOMemory')
...
>>> session = Session(store)

>>> print 'Load \ac{RDF} data'
>>> store.load_triples(source='http://www.w3.org/People/
    Berners-Lee/card.rdf')

>>> Person = session.get_class(ns.FOAF['Person'])

>>> all_persons = Person.all()

>>> print 'Tim Berners-Lee knows %d people' % (len(
    all_persons))
Tim Berners-Lee knows 61 people
>>> for person in all_persons:
...     print person.foaf_name.first
(...)

>>> somebody = Person()
>>> somebody_else = Person()
>>> somebody.foaf_knows = somebody_else
```

Figura 30: Breve ejemplo del uso de SuRF

Telescope

Claramente influenciado por SQLAlchemy Telescope[80] realiza un mapeo de recursos contenidos en un grafo RDF a objetos Python. Esta compuesto tanto de un generador programático de consultas SPARQL, como de mecanismos para mapear recursos RDF a objetos del dominio en Python mediante una sintaxis declarativa. Lamentablemente la documentación del proyecto es más bien escasa, y leyendo el código fuente parece que sólo el generador de SPARQL está en un estado usable, del cual se puede ver un ejemplo en la Figura 31.

```
>>> from telescope import *

>>> FOAF = Namespace("http://xmlns.com/foaf/0.1/")
>>> query = Select([v.name]).where((v.x, FOAF.name, v.name))
>>> query = query.filter(v.name == "Sergio")
>>> query = query.filter(op.bound(v.name), ~op.isBlank(v.x))
>>> print query.compile({FOAF:"foaf"})
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
  ?x foaf:name ?name .
  FILTER (?name = "Sergio") .
  FILTER (bound(?name) && !isBlank(?x))
}
```

Figura 31: Generador programático de consulta SPARQL de Telescope.

80 http://bitbucket.org/exogen/telescope

Se trata por tanto de un proyecto extremadamente interesante que habrá que seguir de cerca, y del cual seguro se pueden aprender muchas cosas.

2.3.6 Persistencia sobre otro tipo de almacenes de datos

AccessPlatform

DataNucleus AccessPlatform[81] provee la persistencia de objetos Java a diferentes almacenes de datos: bases de datos relacionales, bases de datos orientadas a objetos (db4o, NeoDatis), Documentos (XML, Excel, ODF, OOXML), Web (JSON, Amazon S3, GoogleStorage), Map (HBase, BigTable, Cassandra, MongoDB) y otros (LDAP). Para ello implementa diferentes APIs estándares, como por ejemplo JDO [92, 93] y JPA [37, 36]. Cabe destacar que es la tecnología de acceso a datos para aplicaciones Java sobre la Google App Engine[82].

LINQ

LINQ es lenguaje de consulta de propósito general propuesto por Microsoft para dotar de forma nativa de capacidades de consulta de datos a los lenguajes orientados a objetos de su plataforma .NET [68].

```
1 NorthwindDataContext db = new NorthwindDataContext();
2
3 var people = from person in db.People
4              select person;
```

Figura 32: Ejemplo de uso de LINQ para el acceso a bases de datos relacionales

Aunque lógicamente su uso más frecuente es para acceder a bases de datos relacionales, como muestra la Figura 32, LINQ aplicaría a cualquier fuente de información semi-estructurada, mediante la adición de proveedores de acceso apara diferentes fuentes. Por ejemplo, en la Figura 33 puede verse un ejemplo de como se puede utilizar LINQ para acceder a documentos XML. Dada la flexibilidad del diseño de LINQ, comienzan a aparecer proveedores para los más variopintos almacenes de datos, incluso para RDF: LinqToRdf[83].

```
1 XDocument xmlDoc = XDocument.Load("people.xml");
2
3 var people = from person in xmlDoc.Descendants("People")
4              select new Person {
5                   Name = person.Element("Name").Value,
6                   Nick = tutorial.Element("Nick").Value
7              };
```

Figura 33: Ejemplo de uso de LINQ para la consulta de fuentes de datos XML

81 http://www.datanucleus.org/products/accessplatform/
82 http://appengine.google.com/
83 http://linqtordf.googlecode.com/

2.4 RESUMEN Y CONCLUSIONES DEL ESTADO DEL ARTE

Este estado del arte ha tratado de dar una perspectiva lo más amplia posible de las tecnologías y herramientas más relevantes para el acceso a datos en los sistemas software modernos. Aunque el tópico de este proyecto se centre únicamente en como los lenguajes orientados a objetos pueden utilizar modelos de datos basados en RDF, abriendo el horizonte a otro tipos de modelos de datos quizá se haya conseguido una visión mucho más amplia y rica que al final redunde en la calidad del proyecto trioo . De hecho, a pesar de las diferencias entre el modelo orientado a objetos y el relacional, las herramientas de acceso a bases de datos relacionales han sido la referencia más relevante en cuanto a madurez y utilización. Aunque quizá esta situación se encuentre motivada más por la indiscutible posición dominante en el mercado de las bases de datos relacionales, que por el rendimiento y lo fiel que sea el mapeo a objetos. Mención especial merecen Hibernate y JPA (véase la Sección 2.3.1) por su impacto en la industria del software Java, y ActiveRecord (Sección 2.3.1) por su parte de *culpa* en el gran impacto que ha tenido en el desarrollo Web. Y aunque el modelo entidad-relación no se encuentre tan próximo a modelos de datos RDF, hay un gran número de cosas que aprender de la experiencia adquirida con este tipo de herramientas. En ese sentido JDO igual pudiera ser una linea más interesante a seguir.

Volviendo al tópico principal, cuando surgió la idea que motivó este proyecto (allá por verano de 2009) el panorama de las herramientas software para el acceso a modelos de datos RDF no era demasiado alentador: o estaban demasiado cercanas a RDF, lo cual dificultaba enormemente su asimilación por desarrolladores software sin amplia experiencia en este modelo de datos, o estaban demasiado influenciadas por su homólogas en el paradigma relacional. Por tanto la reflexión parecía obvia: dado el incipiente uso que puede tener la familia de tecnologías RDF, su fácil integración con los lenguajes y herramientas orientadas a objetos actuales se antoja algo estratégico, de manera que se facilite en la medida de lo posible su adopción por la industria del software más clásica. Tampoco es válida la aproximación seguida por algunos proyectos[84]. de equiparar una clase RDFS/OWL a una clase de un lenguaje de programación, ambas abstracciones tienen diferente semántica. En el siguiente capítulo se analizarán con mayor detalle, aunque dadas las diferencias que existe entre ambos modelos computacionales (orientados a objetos vs. RDF), dar una solución satisfactoria exige tener muy en cuenta todas esas peculiaridades. En este aspecto es extremadamente interesante la reflexión que hace el proyecto Spira (véase la Sección 2.3.5) acerca de que no se puede (debe) esperar que una clase sea la definición autorizada, exclusiva y completa del modelo de datos RDF. Aunque lógicamente, para marcar unas sólidas bases en las que fundamentar la investigación, este ha sido el primer capítulo que se ha abordado en la realización de este trabajo. Aunque esto no significa que se hubiera cerrado en esas primeras fases del proyecto. Los distintos mecanismos de vigilancia tecnológica se han mantenido a lo largo de toda la ejecución. Por tanto este estado del arte ha sido sometido a continuas revisiones y actualizaciones, tratando a su vez de llevar estos cambios a otras fases del proyecto de manera

84 Como por ejemplo Soprano (http://soprano.sourceforge.net/apidox/trunk/soprano_devel_tools.html#onto2vocabularyclass) o Protégé (http://protegewiki.stanford.edu/index.php/JSave)

que el resultado final las contemple. Y es en estas continuas iteraciones sobre este estado del arte donde ha aparecido Empire (Sección 2.3.5). Empire es una herramienta, desarrollada por Clark&Parsia[85], que si bien está más cercana a JPA de lo que está trioo , ha sabido entender la problemática de una forma muy similar a nosotros y le ha dado una solución eficaz y funcional en un lenguaje orientado a clases como Java. Pero que empiecen a surgir competidores no resta ni un ápice la motivación del proyecto, al contrario, demuestra que el problema esta ahí y aún hay que darle una buena solución. En el próximo capítulo se analizarán las necesidades básicas que se deben cubrir, tratando de aprender de la experiencia de todas las herramientas y tecnologías recogidas en este estado del arte.

85 http://clarkparsia.com/

3 ANÁLISIS DE LOS MODELOS

ÍNDICE

Para alcanzar una solución satisfactoria para dar soporte a modelos de datos basados en RDF dentro de lenguajes orientados a objetos, primero es necesario comprender la naturaleza de ambos modelos computacionales, extrayendo tanto los puntos en común como las diferencias irreconciliables que puede haber entre la semántica de ambos.

3.1 MODELOS COMPUTACIONALES

El modelo computacional, el modelo matemático o formal que hay tras ambas tecnologías, merecería ser estudiado y analizado con mayor detalle. Pero en esta sección se tratará de focalizar en los aspectos más importantes y relevantes para esta investigación concreta, tratando de no perderse en un excesivo detalle que consiga al lector perder el foco del problema.

3.1.1 Modelo computacional orientado a objetos

El paradigma de programación orientado a objetos se basa en la intuitiva correspondencia entre el software que simula un objeto físico y el objeto físico como tal en el mundo real [1]. De ahí precisamente la analogía de "objeto". Su origen[1] se remonta al lenguaje SIMULA [35], un lenguaje de programación diseñado para mejorar la forma en que se analizaban los sistemas mediante el uso de conceptos unificados, objetos. Aunque más adelante Smalltalk [50]

1 Esta breve introducción no pretende ser un recorrido detallado de la historia de los lenguajes de programación orientados a objetos, para eso hay otras fuentes (por ejemplo http://en.wikipedia.org/wiki/Timeline_of_programming_languages), sino que simplemente pretende servir como introducción a los que se consideran más relevantes para este trabajo.

redefiniría parte de estos fundamentos. Después vendría Self [105] introduciendo muchos de los conceptos que todavía hoy en día lenguajes de programación *modernos* empiezan a adoptar. Desde entonces el paradigma orientado a objetos a ido ganando popularidad, y hoy en día está plenamente asentado en el mercado como la opción mayormente escogida para el desarrollo software. Esto es debido en buena medida a tres de sus grandes fortalezas: intuitiva analogía con los sistemas reales a simular, elasticidad y flexibilidad en el modelado, y reutilización de componentes software. Y, a pesar que se siguen rescatando conceptos enterrados de los orígenes de esta filosofía de desarrollo, la programación orientada a objetos sigue estando en boga[2], aunque siempre surge la duda de si se ha seguido el camino incorrecto estos últimos año[3].

Formalmente el modelo orientado a objetos se puede ver como un conjunto de objetos O tal que $O = \{0_1, 0_2, .., 0_n\}$. Cada objeto $o_x \in O$ se define tal que $o_x = (A_x, M_x)$, donde A_x son los atributos de ese objeto y M_x sus métodos. Un atributo $a_x \in A_x$ es un relación unidireccional entre el objeto o_x y cualquier otro objeto $o_y \in O$. Un método $m_x \in M_x$ se trata de una función especial que dispone de acceso a todos los $a_x \in A_x$ del objeto. Aunque tómese esta formalización como simplista incompleta, aunque suficiente para los objetivos que ahora se plantea. Labra realiza un análisis mucho más profundo y detallado de esta estructura y su semántica [48], mientras que Abadi y Cardelli profundizan mucho más en la teoría de objetos [1].

Pero esta formalización es un ideal. En realidad la orientación a objetos no se compone de un único modelo computacional [64], sino que existen varias versiones (*interpretaciones*) del mismo, y además diferentes implementaciones con diferentes matices de algunas de estas versiones. Tradicionalmente se dividen los lenguajes orientados a objetos en dos grandes familias [39]: lenguajes orientados a objetos basados en clases y lenguajes orientados a objetos basados en prototipos. Por tanto es necesario atender a las peculiaridades de ambas aproximaciones en trioo .

Lenguajes orientados a objetos basados en clases

Los lenguajes orientados a objetos basados en clases utilizan un modelo computacional en el cual los objetos se agrupan bajo conjuntos con idéntica estructura y comportamiento, instancias de una misma clase [20]. Las clases son la descripción de los objetos, definen tanto su estructura como su comportamiento. De manera formal una clase C se define como $C = (A, M)$, donde A es el conjunto de atributos de esa clase, y M el conjunto de métodos. Un objeto se genera mediante la instanciación de una clase, usando lo que abstractamente se conoce el operador *new*, aunque luego sintácticamente no sea utilizado en todos los lenguajes. En la instanciación de ese objeto este se inicializa con un contexto, conocido como “self“ o ”this“ dependiendo del lenguaje de programación concreto, dando un valor concreto a los atributos de su clase. Por tanto un objeto O podría definirse como $O = (C, CTX)$, con C es la clase que instancia este objeto, y CTX el valor de sus atributos en un momento determinado.

Por tanto, existe un enlace muy fuerte entre un objeto y la clase que instancia, y no es posible crear o usar objetos sin haber definido previamente su comportamiento y su estructura estática como una clase, y por supuesto el objeto jamás puede modificar o extender la estructura y comportamiento definidos originalmente por la clase.

2 http://www.infoq.com/interviews/johnson-armstrong-oop
3 http://www.infoq.com/news/2010/07/objects-smalltalk-erlang

```
1  class Person {
2
3      private String name;
4
5      public Person() {
6
7      }
8
9      public Person(String name) {
10         this.name = name;
11     }
12
13     public String getName() {
14         return this.name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21 }
```

Figura 34: Ejemplo de una clase en Java.

La Figura 34 muestra una sencilla clase de ejemplo con un atributo y los métodos para manejar ese atributo. Una vez definida esta clase, se podrán instanciar objetos de la misma, tal y como se muestra en la Figura 35. A tenor de ese ejemplo, pudiera parecer que los métodos se encuentran embebidos en los objetos, pero eso no es más que una "ilusión" ya que en realidad los métodos se van a buscar a la clase que instancia el objeto.

```
1 diego = Person("Diego");
2 System.out.println("Me llamo " + diego.getName());
3
4 sergio = Person();
5 sergio.setName("Sergio");
6 System.out.println("Y yo me llamo " + sergio.getName());
```

Figura 35: Ejemplo de una clase en Java.

Adicionalmente aparece el concepto de herencia, por el que una clase (subclase) puede extender de otra, heredando toda la estructura y comportamiento de la superclase. Aunque en realidad el comportamiento puede modificarse en los lenguajes que soportan polimorfismo [71]. En definitiva, estas subclases son especializaciones de las superclases. Aunque tampoco merece la pena profundizar mucho más en este tipo de aspectos.

Lenguajes orientados a objetos basados en prototipo

Por contra, los lenguajes orientados a objetos basados en prototipos[4] pretenden seguir de una manera más fiel la definición original de la orientación a objetos, donde sólo es necesario la existencia de objetos, sin la necesidad de tener la abstracción adicional de las clases [18, 74, 21]. En un lenguaje basado en prototipos los "*trait objects*" [104] (se suele usar la palabra reservada

4 Alguna bibliografía, por ejemplo Abadi y Cardelli [1], se refieren a ellos como lenguajes basados en objetos (object-based languages).

`class`), aunque realmente no son clases) actúan como colecciones de comportamiento (métodos) y estructuras que son iguales para todas las instancias, mientras que las instancias llevan los datos de los objetos. Y la relación de instanciación realmente es una relación de herencia del prototipo. Por tanto la distinción del papel se basa así sobre todo en una distinción entre la estructura y el comportamiento en un lado, y el estado en el otro. El original (y el más canónico) ejemplo de lenguaje prototipado es el lenguaje Self [105], desarrollado por David Ungar y Randall Smith, aunque existen otros muchos lenguajes que gozan de gran popularidad: Python, Ruby o JavaScript, entre otros.

Figura 36: Tripleta RDF (fuente: RDF Primer).

3.1.2 Modelo RDF

RDF [59, 67] (*Resource Description Framework*) provee un modelo de datos extensible y extremadamente flexible. Basado en el concepto básico de "tripleta" `(sujeto, predicado, objeto)`, donde el sujeto representa el recurso que describe la tripleta, el predicado es la propiedad que establece la relación con otro recurso, representado por el objeto, o con un dato simple, denominado literal. Gráficamente se puede ver en la Figura 37. Formalmente una tripleta RDF se define como una tupla (s, p, o) tal que $(s, p, o) \epsilon (U \cup B) \times U \times (U \cup L \cup B)$, donde U es el universo de todos los posibles recursos nombrados , identificados por una URI [9], B es el conjunto infinito de nodos en blanco, y L es el conjunto de todos los literales RDF. Un conjunto de tripletas se interpreta estructuralmente como un grafo $G = (V, E)$, conde V es un conjunto de vértices tal que $V \subset (U \cup B \cup L)$, y E es un conjunto de arcos nombrados tal que $E \subset V \times U \times V$. Por cada tripleta (s, p, o) del modelo de datos RDF hay un arco dirigido, etiquetado p, entre los vértices s y o. Es decir, los recursos (tanto sujetos como objetos) son los nodos de un grafo dirigido, y los predicados los arcos. Por tanto las estructuras de datos en RDF son conjuntos de tripletas que forma un grafo RDF.

La flexibilidad que ofrece este modelo datos simplifica drásticamente la interoperabilidad, intercambio e integración de datos desde fuentes y aplicaciones heterogéneas. Pero a semántica de RDF es arduo complicada. RDF es un lenguaje asertacional para expresar proposiciones usando vocabularios formales [53]. La semántica del lenguaje está especificado usando teoría de modelos, una técnica en la cual se asume que el lenguaje se refiere a un "*mundo*", y describe las condiciones mínimas que ese *mundo* debe satisfacer para ser una interpretación que haga válida una expresión en ese lenguaje. La utilidad de una teoría semántica formal no es otra que proveer un mecanismo que sirva para determinar procesos de inferencia válidos; en otras palabras, cuando la verdad se preserva. A través de las nociones de satisfacibilidad, vinculación y validez, la semántica de RDF da una definición rigurosa del significado de un grafo RDF. Aunque realmente, la especificación de la semántica de RDF define cuatro interpretaciones normativas para los grafos RDF: Simple, RDF, RDFS, y Datatype.

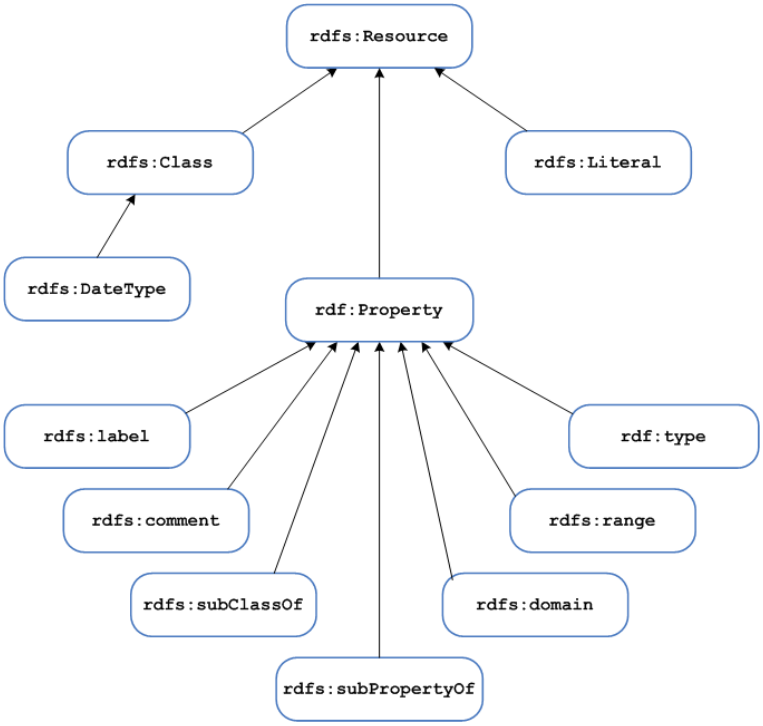


Figura 37: Extracto de la jerarquía de RDFS.

SIMPLE Simple-entailment captura la semántica de un grafo RDF cuando no es necesario atender a el significado de los nombres de los arcos y nodos del grafo. Esta es precisamente la semántica soportada por la versión actual de SPARQL, SPARQL 1.0. Si se quisiera obtener el significado completo de un grafo usando un vocabulario particular, entonces sería necesario añadir más condiciones semánticas que permitieran interpretar elementos en un grafo.

RDF Por tanto, RDF-entailment añade un soporte básico para la interpretación básica de entidades en un grafo. Utilizando el vocabulario de RDF se

le añade al grafo elementos básicos como son el tipado, los literales, estructuras básicas como las listas, etcétera.

D-ENTAILMENT Con Datatype se añaden nuevos implicaciones que permite utilizar condiciones extras para el tratamiento de nuevos tipos de datos identificados por una URI.

RDFS Y finalmente RDFS (RDF Schema [25]) extiende RDF introduciendo un nuevo vocabulario con reglas y restricciones con una semántica más compleja, tales como clases, herencia de clases, dominios y rangos para las propiedades, etc.

Los grafos RDF, serializados utilizando RDF/XML [7], son también la representación normativa de ontologías en OWL para propósitos de intercambio. Puesto que la semántica de OWL introduciría demasiada complejidad, añadiendo la necesidad de interpretar en vocabulario de OWL y sus axiomas mediante soporte para razonamiendo con lógica descriptiva [4], en un principio no se le planea darle soporte en el proyecto trioo a OWL, aunque esto será discutido más adelante a lo largo de este capítulo.

3.2 COMPARACIÓN

La breve introducción realizada hasta ahora ya arroja algunos de los interrogantes acerca del posible encaje que pueden tener estos diferentes modelos computacionales entre si. A continuación se procederá a realizar una comparación individualizada y pormenorizada de cada uno de estos aspectos.

3.2.1 Estado y comportamiento

En primer lugar cabe destacar que ambos modelos computacionales tienen dos aproximaciones totalmente diferentes. RDF es básicamente un lenguaje para la representación del conocimiento, cuyo propósito es proporcionar un modelo de datos común para la descripción de recurso en la Web. Mientras que el modelo computacional orientado a objetos esta pensando para ser implementado en lenguajes de programación, y por tanto a ser un modelo que de forma autónoma puede cambiar tanto su estructura como contexto mediante la ejecución de las instrucciones de las aplicaciones. Resumiendo, los lenguajes de programación no tienen las mismas características que los lenguajes de representación como RDF, por lo que sólo serán comparables las estructuras de datos en ambos lenguajes. Por tanto, aunque estructuralmente ambos modelos computacionales compartan determinadas semejanzas estructurales, hay diferencias dinámicas irreconciliables.

3.2.2 Identidad

La identidad de un objeto en un lenguaje de programación es algo complejo. Si bien las clases suelen poder identificarse con un nombre canónico para evitar conflictos en el nombrado, para los objetos depende del nivel de abstracción al que se quiera realizar esta identificación. A bajo nivel un objeto se identifica por la dirección de memoria que ocupa, o la referencia en la heap de la maquina virtual en lenguajes que se ejecutan sobre una máquina

virtual. Esos punteros y referencias puede utilizarse para identificar un objeto concreto en tiempo de ejecución. Aunque dado que pueden convivir en memoria en un mismo instante dos objetos distintos que en realidad representan el mismo objeto del mundo real, esa referencia no puede utilizarse para identificar un objeto dentro de la lógica de negocio de un programa. Para ello los lenguajes de programación ofrecen protocolos concretos acerca de como realizar estas comparaciones, permitiendo al desarrollador implementar esa lógica sobreescribiendo determinados métodos (`equals()` en Java, `__eq__()` en Python, etcétera). Por tanto es posible determinar si dos objetos distintos son en realidad el mismo objeto, pero no es posible dotar a los objetos de una identidad global por la que ser identificados.

Por el contrario el concepto de URI [9] está íntimamente relacionado a RDF. Las URIs son identificadores universales que no cambian [10]. Como hemos visto anteriormente, el sujeto de una tripleta puede ser tanto una URI como un nodo en blanco (*blank node*)[5]. Los nodos en blanco suelen utilizarse sólo para describir recursos al los que no es necesario dotarles de una identidad, dado que no tienen mayor relevancia fuera del un contexto determinando en el grafo RDF. Por tanto los recursos que son relevantes en RDF se les *nombra* con una **UR!**[6], con lo que puede referirse a un recurso incluso desde otros grafos. Si encima se utilizan URIs HTTP, esa identidad se lleva a un ámbito global en la Web.

Aunque existan servicios[7] e implementaciones para algunos lenguajes[8], integrar la semántica de la identidad en ambos modelos computacionales exigiría, al menos, establecer una preferencia en los criterios que la deciden. En otras palabras, si el programador decide que dos objetos con distinta OWL son los mismos, ¿eso te legitima para general una tripleta `owl:sameAs` entre ambos recursos?

3.2.3 Tipado

Comparar el tipado es un interesante reto. Para empezar, en ambas familias de lenguajes orientados a objetos (basados en clases y basados en prototipos) el tipado se implementa de una forma diferente. En lenguajes de programación basados en clases el tipado se construye encima de un modelo jerárquico de tipos definidos por clases. Por tanto la herencia se define a nivel de clase, construyendo una jerarquía de clases. Sin embargo, en los lenguajes de programación basados en prototipos se provee una implementación de los tipos mucho más fiel a la filosofía original del modelo orientado a objetos. En los lenguajes basados en prototipos la relación de instanciación entre dos objetos es en realidad una relación de derivación/herencia, por lo que ya no es necesaria la relación de instanciación como en los lenguajes basados en clases. Además, idealmente la herencia podría ser desde más de un objeto a la vez, habitualmente la instanciación múltiple es algo que por razones de usabilidad pocos lenguajes basados en prototipos proveen. En RDF el tipado funciona de

5 En el reciente workshop organizado por W3C para actualizar RDF (`http://www.w3.org/2009/12/rdf-ws/`) se ha discutido también la posibilidad de incluir a los literales como elementos susceptibles de ser sujeto de una tripleta RDF (algo a lo que tampoco se cierran en otras especificaciones del stack tecnológico de RDF, como por ejemplo SPARQL), aunque finalmente parece que hay un interés por realizar un cambio de reste tipo en el modelo de RDF (`http://www.w3.org/2010/06/rdf-work-items/table`).

6 **UR!**

7 `http://sameas.org/`

8 `http://code.google.com/p/asmx-sameas4j/`

una manera muy similar a como lo haría un lenguaje basado en prototipos puro: cada objeto RDF puede tener simultáneamente varios tipos, sin ningún tipo de restricción, al menos si no *salimos* de RDF-entailment. En RDF no hay chequeo de tipos; es más, puede haber recursos sin tipo. En la Figura 38 se ilustra gráficamente algunas de estas diferencias acerca del tipado.

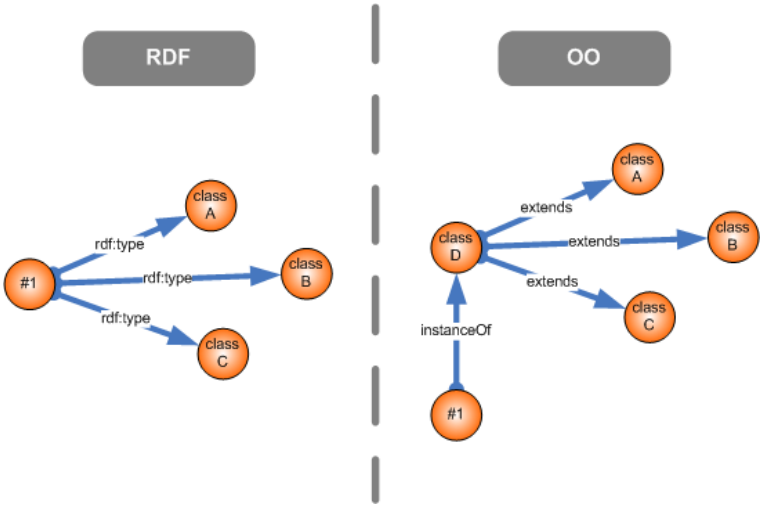


Figura 38: Diferencias entre el tipado en RDF y en orientación a objetos.

La inferencia de tipos se describe a continuación, como complemento al tipado.

3.2.4 Inferencia de tipos

La inferencia de tipos se refiera a una característica de los lenguajes de programación para automáticamente deducir el tipo de un valor, tanto en tiempo de compilación como en tiempo de ejecución dependiendo del lenguaje concreto. Se trata de una característica muy común en lenguajes dinámicamente tipados, aunque también aparece esporádicamente en algunos lenguajes fuertemente tipados, como puede ser el caso de C#. En programación esta inferencia de tipos afecta al tipo de las variables, o atributos en caso de lenguajes orientados a objetos.

Por el contrario, en RDF las relaciones son declaradas utilizando propiedades. Las propiedades son declaradas de manera global por cada vocabulario RDF, por tanto la misma propiedad (mediante su URI puede ser utilizada para relacionar diferentes objetos. En RDFS (y en OWL) estas propiedades pueden declarar tanto un dominio como un rango: el tipo que el sujeto y el valor podrán tomar respectivamente. Por tanto, dada un propiedad P que declara una restricción de dominio D y una restricción de rango D, con una tripleta tal que (a, P, b) se inferirá que a es tipo D y b es de tipo R. Sobra decir que este tipo de inferencia de tipos es muy complicado de soportar en lenguajes estáticamente tipados, pero se antoja realizable en lenguajes dinámicos.

3.2.5 Navegabilidad

Dado que ambos modelos computacionales al final pueden ser vistos como un grafo dirigido, la navegabilidad es un aspecto muy importante a estudiar.

En la mayoría de lenguajes de programación se ha adoptado el operador punto (“`.`”) para navegar por los atributos de los objetos, pudiendo acceder mediante expresiones tipo `objeto.atributo` a los atributos de un objeto. Por ejemplo, con un objeto de la Figura 34 se podría acceder a su atributo nombre utilizando `objeto.name` si el atributo fuera público. Pero eso ya depende de las políticas de visibilidad que implemente cada lenguaje concreto. Por ejemplo, en Python siempre se puede acceder a los atributos de esa manera. Pero dado que la visibilidad de los atributos es una cuestión importante del propio lenguaje, mucho ofrecen subterfugios alternativos para facilitar este acceso a los atributos sin violas restricciones de seguridad. Por ejemplo Java dispone del Expression Language (EL), un lenguaje de scripting que, siguiendo la convención de nombrado para *getters* y *setters* estándar del lenguaje, permite el acceso a objetos Java mediante el uso de expresiones tipo `objeto.atributo` en las JSP[9] [31]. Por otro lado en el lenguaje C# se puede encapsular el acceso a los atributos de una forma muy ingeniosa, utilizando lo que denominan Properties[10]. Teniendo en cuenta la clase definida en la Figura 39, `objeto.name` no sería un acceso válido debido a las restricciones de seguridad declaradas, pero sin embargo `objeto.Name` sí lo sería, tanto para lectura como para escritura.

```
1  public class Person {
2
3      private string name;
4
5      public string Name {
6          get { return name; }
7          set { name = value; }
8      }
9
10 }
```

Figura 39: Acceso a atributos utilizando las properties de C#.

Por el contrario, la navegabilidad de grafos RDF ha sido siempre un tema delicado y aún sin una solución uniforme y completa, como ya existe hace muchos años para, por ejemplo, XML [32], dados los problemas que aparecen al navegar grafos [38]. Concretamente en grafos RDF, el primer intento por avanzar en este aspecto se realizo en la sintaxis N3[11] [14], proponiendo un sencillo lenguaje de rutas para resolver algunos sencillos problemas. Pero esta claro que este problema debe resolverse desde un lenguaje que permita la selección, un lenguaje de consulta, en este caso SPARQL [86]. Si nos fijamos en la consulta de la Figura 40 podremos ver que con SPARQL no es cómodo realizar consultar sobre propiedades multivaluadas, y habrá que realizar un postprocesado de los resultados para volver a agruparlos, lo cual es una limitación bastante importante. En estos años ha habido algunas propuestas bastante in-

9 JavaServer Pages
10 http://msdn.microsoft.com/en-us/library/x9fsa0sw(28VS.80).aspx
11 Notation 3

teresantes [60, 87][12] [13], incluso algunas implementaciones[14]. Pero nunca nada dirigido hasta que el grupo de trabajo de W3C incluyó esta características dentro de su hoja de ruta[15], dotando al fin a SPARQL 1.1 de un lenguaje de rutas [98], profundamente inspirado en la propuesta de Pérez et al. [87]. La consulta de la Figura 41 reemplazaría la anterior, aunque obviamente se trata de un ejemplo muy sencillo y el lenguaje de rutas permite cosas mucho más potentes.

```
1 SELECT ?person ?name
2 WHERE {
3   ?person a foaf:Person .
4   ?person foaf:friend ?friend .
5   ?friend foaf:name ?name .
6 }
```

Figura 40: Consulta para obtener todos los amigos de una persona con SPARQL.

```
1 SELECT ?person ?name
2 WHERE {
3   ?person a foaf:Person .
4   ?person foaf:knows/foaf:name ?name .
5 }
```

Figura 41: Consulta para obtener todos los amigos de una persona con SPARQL.

3.3 OTROS ASPECTOS

Pero además hay otra serie de aspectos sobre los que es necesario evaluar el impacto que pueden tener al integrar ambos modelos computacionales.

3.3.1 Expresividad

Como ya se ha comentado anteriormente, a priori trioo no dará soporte a OWL [78], por diferentes motivos, aunque principalmente por un compromiso entre expresividad y decidibilidad. Sin embargo, Horst ha probado que RDFS entailment es decidible, NP-completo, y dentro de P[16], si el grafo RDF sobre el que se trabaja no contiene nodos en blanco [103]. Por tanto pudiera ser mucho más realista utilizar la semántica de RDFS. Además, este nivel de expresividad pudiera ser alcanzado sin la necesidad de aplicar un proceso completo de razonamiento sobre el conjunto de datos, porque el subconjunto de operadores necesario puede ser implementado directamente sobre el modelo de objetos de una forma un tanto *basta*; de hecho ya existe una implementación en Python de tal técnica[17].

12 http://esw.w3.org/RdfPath
13 http://exmo.inrialpes.fr/software/psparql/
14 http://jena.sourceforge.net/ARQ/property_paths.html
15 http://www.w3.org/2009/sparql/wiki/Feature:PropertyPaths
16 Con una complejidad en tiempo polimórfico.
17 http://www.ivan-herman.net/Misc/PythonStuff/RDFSClosure/

3.3.2 Restricciones de integridad

Las restricciones de integridad son utilizadas para asegurar la precisión y la consistencia de los datos. Estas restricciones está directamente aseguradas por los motores de persistencia en otras familias de almacenes de datos, como por ejemplo en los modelos entidad-relación. Sin embargo, detectar este tipo de violaciones de la integridad de los datos esta totalmente fuera de las responsabilidad de RDFS y OWL, dado que se trabaja sobre el supuesto de mundo abierto (OWA[18] [54]). Bajo mundo abierto, no puede inferirse que una sentencia es falsa sólo porqué no se pueda probar; por tanto si no hay evidencias, es imposible decir si es cierto o es falso. Una consecuencia de trabajar en mundo abierto es que todo puede ser verdadero mientras no haya evidencias de lo contrario. En mundo abierto se asume que la información puede ser incompleta, algo que realmente útil para la gestión del conocimiento en contexto abiertos como la Web, dado que las deducciones de limitan a realizarse sólo cuando se conoce que algo es cierto.

```
1 foaf:name  rdfs:domain  foaf:Person .
2 :sergio    foaf:name    "Sergio" .
```

Figura 42: Grafo serializado en N3 con información sobre una persona.

Sin embargo, volviendo a las restricciones de integridad, RDFS y OWL sólo son capaces de detectar inconsistencias lógicas en los datos, pero la falta de información no es suficiente para representar una violación de las restricciones de integridad. Por ejemplo, considérese el grafo RDF serializado en N3 de la Figura 42. Bajo mundo abierto las restricción de dominio de la propiedad `foaf:name` implicaría que `:Sergio` es una `foaf:Person`, pero no existe ninguna violación del dominio si esto no se indica. La falta de información no es causa de inconsistencias, aunque esa información puede ser inferida mediante la aplicación de las reglas de RDFS, y puede ser de mucha utilizar en escenarios en los que se requiera validación de los datos. Para chequear este tipo de inconsistencias se deberá explorar como introducir parcialmente mundo cerrado (CWA[19]) como alternativa al mundo abierto. Con la interpretación de mundo cerrado se considera falso si simplemente no hay evidencias, verdaderas o falsas, a considerar. Esta semántica está comúnmente soportada por los sistemas de programación lógicos tradicionales mediante la implementación de la negación como fallo (NAF[20] [33]). Existen varias formas de atender a las restricciones de integridad para los axiomas RDFS o OWL de una ontología [102, 73]. Una de las aproximaciones más interesantes se basa en la traducción de axiomas OWL a consultas SPARQL mediante la interpretación definida por Sirin et al. [101], cambiando de paradigma de modelado seguido por la Web Semántica [79]. Dado que SPARQL es equivalente a datalog no recursivo [29, 81], y NAF podría ser codificado utilizando patrones OPTIONAL/FILTER/!BOUND en SPARQL. De hecho Pellet Integrity Constraint Validator[21] implementa esta aproximación, por lo que una ontología puede ser utilizada para validar la integridad de datos RDF. La restricción de dominio sobre la propiedad `foaf:name` definida anteriormente en la Figura 42 podría

18 Open World Assumption
19 Closed World Assumption
20 Negation as Failure
21 http://clarkparsia.com/pellet/icv

ser escrita como una restricción de integridad con SPARQL similar a la que se ilustra en la Figura 43 a modo de reglas en SPARQL.

```
ASK WHERE {
  ?person  foaf:name  ?name .
  OPTIONAL { ?person2  rdf:type foaf:Person }
  FILTER ( bound(?person2) )
  FILTER ( ?person = ?person2 )
}
```

Figura 43: Restricción de integridad escrita como consulta SPARQL.

3.3.3 Versionado

Potencialmente trioo podría encargarse de la ardua tarea de manejar el versionado de los datos. Por ejemplo algunas bases de datos relacionales ofrecen mecanismos propietarios para realizar nativamente esta tarea[22], aunque para RDF no aplican algunas cuestiones como el versionado del esquema [90]. En el fondo del versionado aparece una cuestión extremadamente interesante y a la que aún no se le ha encontrado una solución satisfactoria: cómo extraer las diferencias existentes entre dos grafos RDF [13]. Las propuestas han sido varias: utilizar una ontología para gestionar el versionado [107], usar Deltas RDF [109], o incluso aplicar el control sobre subgrafos de manera distribuida [95]. Ciertamente es un aspecto que queda fuera del alcance actual, pero que sería extremadamente interesante retomar a futuro.

3.4 CONCLUSIONES EXTRAÍDAS

Como se preveía, aunque de primeras pudiera parecerse, ambos modelos computacionales tienen muchas diferencias semánticas. Por lo visto en este análisis, que ni es un análisis concienzudo ni final, los lenguajes orientados a objetos basados en prototipos podrían estar más cercanos a RDF, dado que el concepto de clase en lenguajes basados en clases *choca* un poco con la flexibilidad de RDF.

En el próximo capítulo se evaluarán con más detalles las posibilidades desde un plano más real, poniendo en práctica ya sobre lenguajes reales concretos los conceptos aquí estudiados.

22 http://download.oracle.com/docs/cd/B28359_01/readmes.111/b28280/toc.htm#BABGIGDC

4 VIABILIDAD TÉCNICA

ÍNDICE

A tenor de los resultados del análisis realizados en el Capítulo anterior (3), ambos modelos computacionales, a un lado el orientado a objetos y al otro RDF, se encuentra mucho más alejados de lo que inicialmente se esperaba. Como se ha visto hasta ahora, ambos presentan diferencias semánticas irreconciliables. Será por tanto necesario evaluar si en el estado actual de la tecnología es viable enfrentarse a esta integración de forma realista.

4.1 REQUISITOS TECNOLÓGICOS

Hay determinados aspectos que, a tenor de lo estudiado hasta ahora, sería interesante analizar con mayor detalle a ver si es viable darle cabida en las diferentes implementaciones que se vayan a realizar.

4.1.1 Persistencia

Respecto a cómo realizar la persistencia, ese es quizá uno de los elementos claves que motivó este proyecto. La actual versión del lenguaje de consulta, SPARQL 1.0 [86], no dispone de capacidades de modificación ni inserción de nuevos datos, está limitado a consultas de acceso. Esta circunstancia ha motivado que para cubrir tal carencia la mayoría de las soluciones analizadas en el Capítulo 2 se vean abocadas a implementar las APIs propietarias que cada base de datos RDF ofrece. Tal circunstancia limita en gran medida la portabilidad de aplicaciones, dado que para los desarrolladores de las bibliotecas es imposible cubrir toda la familia de RDF stores. W3C se ha dado cuenta de que tal circunstancia esta limitando la adopción de las tecnologías de la Web Semántica, por lo que a principios de 2010 ha arrancado un grupo de trabajo para actualizar SPARQL[1]. Pero SPARQL no es la única linea de trabajo. Tim Berners-Lee también propone utilizar WebDAV[2] [49] para modificar directamente los datos

1 http://www.w3.org/2009/05/sparql-phase-II-charter
2 Web-based Distributed Authoring and Versioning

publicados como Linked Data [12]. Aunque no se descarta este último mecanismo, a un corto plazo se prefiere apostar por SPARQL 1.1 como mecanismo principal mediante el cual persistir los datos.

Pero apostar tan claramente por SPARQL tiene ciertos riesgos que es necesario tener claros. Dado que la sintaxis para la modificación (anteriormente conocida como SPARUL [96] como propuesta de Hewlett-Packard), el estado de las implementaciones es aún bastante limitado. Por ejemplo, en las versiones actuales de Jena sólo está soportada la anterior sintaxis[3]. ARC (PHP[4]) soporta una sintaxis un tanto personal denominada SPARQL+[5]. Las distintas bibliotecas para Python aún no han tenido en cuenta esta nueva circunstancia. Y así con el resto de bibliotecas... Por tanto parece claro que para no desviarse de los objetivos del proyecto habrá que tratar de confiar en la implementación que ofrezcan los RDF stores. Y si de paso tenemos la posibilidad que soporten consultas federadas [88, 84], pues mucho mejor.

A este respecto, en algún momento será necesario formalizar el álgebra utilizada para persistir el modelo orientado a objetos a RDF, al igual que ya se. ha realizado con el modelo relacional[34]. En este sentido puede ser de utilidad el trabajo que el grupo incubadora del W3C[6] viene realizando para estandarizar el mapeo entre ambos modelos[7] .

```
1  public class Something {
2
3      private String name;
4      private Other other;
5
6  }
7
8  private class Other {
9
10     private String otherstuff;
11
12 }
```

Figura 44: Ejemplo para ilustrar el problema de las diferencias estructurales.

4.1.2 Publicación

Pero además de permitir persistir y utilizar datos RDF, debería permitir exponer modelos de datos orientados como Linked Data [11] de una forma sencilla. Se trataría de proveer la mayor flexibilidad posible para que la gente publique sus datos. Focalizando en ofrece al menos serializaciones estándar de los datos (RDF/XML [**?**]), y atendiendo a detalles que siempre son complicados de implementar como la negociación de contenidos [42]. Tampoco sería descabellado ofrecer una aproximación basada en RDFa[8] [2], como por ejemplo ya se ofrece en Grail[9], aunque esto no sería prioritario. Lo que sí es primordial

3 http://jena.sourceforge.net/ARQ/update.html
4 PHP Hypertext Preprocessor
5 http://arc.semsol.org/docs/v2/sparql+
6 http://www.w3.org/2005/Incubator/rdb2rdf/
7 http://www.semanticuniverse.com/blogs-relational-database-and-semantic-web.html
8 RDF in attributes
9 http://www.grails.org/plugin/rdfa

es centrar esfuerzos en hacer esta característica lo más sencilla posible de utilizar, integrándola con las distintas soluciones que ofrece la industria para el desarrollo de aplicaciones Web (frameworks MVC[10], etc). Aunque eso ya será un detalle particular de cada implementación.

4.1.3 Legibilidad

Las URIs están íntimamente ligadas a RDF, todo se nombra con URIs. Pero dado que el uso de URIs ensuciaría notablemente la legibilidad del código, desde un primer momento se querido optar por el uso de CURIEs[11] [17]. Una CURIE es una sintaxis compacta para expresar URIs. Por ejemplo, la URI de la clase Person de FOAF[12] [26] se puede representar como la CURIE `foaf:Person`. Esta elección tiene un aspecto positivo muy claro en términos de legibilidad y brevedad al referirse a una URI, pero también tiene uno negativo, ya que cada implementación requerirá registrar previamente los mapeos entre una espacio de nombres y su etiqueta (prefijo). Para ello opcionalmente se debe permitir utilizar servicios como prefix.cc[13].

4.1.4 Diferencias estructurales

Durante el análisis del estado del arte se ha visto que pocas herramientas atienden la necesidad de manejar diferencias estructurales en ambos modelos. Por ejemplo, en muchos casos puede que el diseño orientado a objetos se componga de una serie de objetos (como lo representado en la Figura 44) que necesariamente no quieran describir como RDF (algo como por ejemplo lo que se muestra en la Figura 45).

```
1 <> a ex:Something ;
2   ex:name "name" ;
3   ex:other "otherstuff" .
```

Figura 45: Ejemplo en N3 para ilustrar el problema de las diferencias estructurales.

Por tanto será interesante estudiar la mejor forma de implementar este comportamiento. Quizás más que por el mero hecho de persistir a RDF estas estructuras complejas, por acomodar datos RDF ya existentes a este tipo de diseños.

4.1.5 Mapeos dinámicos

Las relaciones entre objetos no son estáticas, sino que pueden tener una dinamicidad. Considérese por ejemplo la variabilidad en la relación entre las personas: puedes simplemente expresar que conoces a alguien, o puedes ser más concreto diciendo que es un familiar o un compañero de clase. En definitiva, grafos generalizados [103]. Soportar únicamente mapeos estáticos limitaría demasiado el uso de la herramienta, y no cubriría muchos casos de uso de los

10 Model-View-Controller
11 Compact URIs
12 Friend of a Friend
13 http://prefix.cc/

potenciales programadores. Por ejemplo es una necesidad que ha aparecido tratando de utilizar trioo con TeRRAS[14] [72].

Lógicamente esto es mucho más sencillo de conseguir en lenguajes basados en prototipos. Pero en lenguajes basados en clases debe encontrarse un mecanismos que permita que los mapeos no sean únicamente definidos de manera estática en tiempo de compilación, sino que puedan ser modificados en tiempo de ejecución. Dado lo dependiente que es esta característica de cada lenguaje de programación concreto, este aspecto se tratará con más detalle para cada una de las implementaciones.

4.1.6 Configuración

Aunque no sea un aspecto tan particular de este proyecto, cuando se abordó se quería explorar formas de configuración alternativas a los clásicos ficheros XML. Por tanto será necesario evaluar otras alternativas, tanto basadas en fichero (como por ejemplo YAML [8]), como basadas en convenciones de nombrado. Lo primero podrías aplicarse a cualquier tipo de lenguaje, aunque lo segundo quizá sólo sería aplicable a lenguajes basados en prototipos.

4.2 LENGUAJES CANDIDATOS

Con análisis del estado del arte del Capítulo 2 se ha visto que muchas veces la elección de una u otra tecnología atiende muchas veces más a motivos irracionales que a motivos con una justificación racional. Por ello, para la elección de los lenguajes en los que validar trioo se han tenido en cuenta principalmente dos motivos: su potencial impacto en el mercado y sus posibilidades tecnológicas. Desde el principio uno de los aspectos clave era la transversalidad del proyecto respecto del lenguaje de programación: realizando el análisis de la manera más independiente posible de un lenguaje de programación concreto, las diferentes decisiones de implementación de uno u otro lenguaje no tendrían tanto impacto el estudio y no alterarían los resultados.

Pero un proyecto de investigación no puede quedarse sólo en la mera investigación básica. Es necesario llevar estos resultados a escenarios reales para realizar una validación de la investigación. Y desde un primer momento se ha tenido claro que, dentro de las limitaciones temporales del proyecto, habría que implementarlo en diferentes lenguajes de programación. Y del éxito de esta última fase dependerá en buena medida el impacto que el proyecto pueda alcanzar. Por tanto la elección del marco (o marcos) en donde implementar el proyecto se antoja una decisión extremadamente importante. Para tomar una decisión adecuada hay que valorar, por un lado la potencial aplicabilidad de este estudio en el lenguaje concreto, y por otro lado el impacto que pueda tener en la industria. Respecto a lo primero, a tenor de lo visto anteriormente, parece claro decir que un lenguaje basado en prototipos se integra mejor con el modelo de datos RDF. Los lenguajes basados en clases dependen tan fuertemente de la clase como esa entidad que contiene la definición última de una entidad, que es complicado se adapten a la flexibilidad de RDF. Pero el impacto en la industria puede ser un aspecto que equilibre la balanza un poco más. Es difícil hablar con exactitud, por tanto nos basaremos en el estudio que mensualmente realiza la comunidad TIOBE para determinar una aproximación de la evolu-

14 http://terras.sourceforge.net/

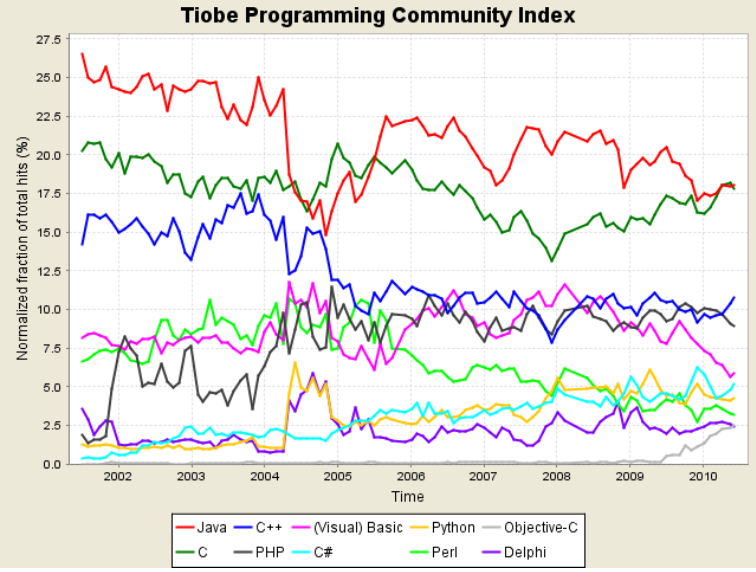


Figura 46: Índice TIOBE.

ción en el uso de los diferentes lenguajes de programación[15]. La Figura 46 muestra la evolución a lo largo de los últimos 9 años de los lenguajes de programación más utilizados. Cuando se realizó este estudio[16], el lenguaje Java aún disfrutaba de una privilegiada posición. Es gracioso destacar que al poco de haber realizado esta toma de datos, dicho ranking daba un pequeño vuelco: en abril el lenguaje C volvía a la primera posición, aunque sólo sería de manera temporal, pues en junio la cosa volvería a darse la vuelta, dejando esto en una mera anécdota sin mayores consecuencias para este estudio. La Figura 47 muestra la distribución actual[17].

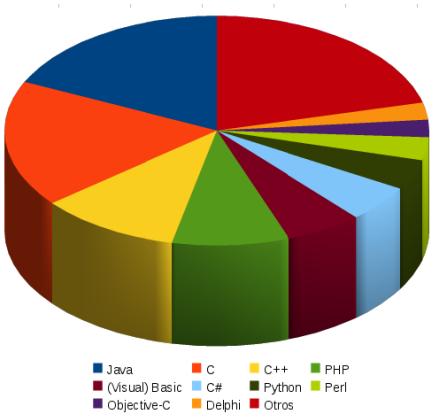


Figura 47: Distribución del uso actual de los lenguajes de programación (fuente: TIOBE).

15 Disponible online en http://www.tiobe.com/index.php/tiobe_index
16 En febrero de 2010.
17 Actualizado a junio de 2010.

Lenguaje	Porcentaje de Uso
Java	18.033 %
C	17.809 %
C++	10.757 %
PHP	8.934 %
(Visual) Basic	5.868 %
C#	5.196 %
Python	4.266 %
Perl	3.200 %
Objective-C	2.469 %
Delphi	2.394 %
Otros	21.07 %

Tabla 1: Distribución actual del uso de los lenguajes de programación.

Por tanto, teniendo en cuenta ambos aspectos (características y popularidad), eses necesario realizar una evaluación más detallada de alguno de los lenguajes más relevante, a fin de tener suficiente criterios que quién la elección de los lenguajes sobre los que realizar las implementaciones.

4.2.1 Candidatos al detalle

Java

Como se acaba de ver, desde 2001 Java es el lenguaje más utilizado. Java[18] es un lenguaje de propósito general, orientado a objetos y basado en clases [51]. Su modelo de datos es bastante simple y carece de algunas características de bajo nivel, como por ejemplo el acceso directo a memoria. Su modelo de clases no soporta herencia múltiple, por lo que para tal propósito recurre al uso de interfaces (tipos de referencia que carecen de implementación). Las aplicaciones Java se compilan a un lenguaje intermedio, conocido como bytecode, que es interpretado por la máquina virtual (JVM[19]), con ciertas características de reflectividad. Adicionalmente a todas estas características, cabe la pena destacar Java el lenguaje de programación con mayor y mejor soporte para las tecnologías de la Web Semántica, con un amplio abanico de herramientas: Jena[20], OWLAPI[21], Pellet[22], etc. Por tanto, todas estas razones convierten a Java en el primer candidato para realizar una implementación de trioo.

Python

Sin embargo, se ha visto que los lenguajes dinámicos se encuentran más cercanos al modelo de datos de RDF, dado que realmente implementan un modelo basado en prototipos. Descartando Self [105] por puras razones de usabilidad, aunque quizá sea el más puro y completo lenguaje candidato, Python es el candidato más realista. Python[23] es un lenguaje de propósito general orientado

18 http://www.java.com/
19 Java Virtual Machine
20 http://openjena.org/
21 http://owlapi.sourceforge.net/
22 http://clarkparsia.com/pellet/
23 http://www.python.org/

a prototipos, dinámicamente tipado [106]. Python[24] fue diseñado por Guido van Rossum[25] bajo la filosofía de enfatizar la legibilidad del código. Ofrece un modelo computacional muy potente y versátil, que soporta diversos tipos de paradigmas de programación orientada a objetos. Además posee de una gran número de interesantes características: introspección, reflectividad computacional (a menudo la unión de estas dos se le llama meta-programación), herencia múltiple, entre otras, convirtiendo su modelo computacional en uno de los más usables y versátiles de entre los lenguajes de programación modernos. Si a todo ello le sumamos la experiencia con que dentro del equipo se cuenta sobre este lenguaje, Python es otro candidato perfecto.

C#

C# es un lenguaje de programación basado en clases desarrollado originalmente por Microsoft para su plataforma .NET, y posteriormente estandarizado por el ECMA [56]. De una sintaxis muy parecida a C++, pero con una aproximación más parecida a Java, C# ha cobrado una especial relevancia dada a la rápida incorporación de nuevas versiones y características, como por ejemplo el soporte para el tipado dinámico y capacidades provenientes de lenguajes funcionales. Por tanto estas nuevas características hace de C# de un candidato muy interesante sobre el que probar los conceptos aquí estudiados.

Javascript

JavaScript [43] es un lenguaje de scripting basado en prototipos con tipado dinámico, con funciones de primer orden y con determinadas características de programación funcional. En realidad es una implementación del lenguaje estándar ECMA262 [55], un lenguaje muy comúnmente utilizado para dotar de comportamiento dinámico a las páginas Web. En los últimos años ha cobrado una especial relevancia dado el uso masivo de AJAX[26]. Por tanto, dadas sus interesantes características, lo convierten en otro candidato que será interesante estudiar. Incluso se podría tratar de dar soporte a RDFa [2] de una forma más limpia y dinámica en las interfaces Web.

4.3 EXPERIMENTACIÓN

Una vez se han extraído los requisitos tecnológicos y seleccionados los lenguajes candidatos, puede decirse que por fin se está en condiciones de llevar a la práctica los conceptos analizados a lo largo de estos tres últimos capítulos. En el siguiente (Capítulo 5) se llevará por tanto lo analizado a un plano mucho más práctico, tratando de implementar las conclusiones de forma que sirva como validación de la investigación más básica realizada.

24 Su nombre es un homenaje a los humoristas británicos Monty Python.
25 http://www.python.org/~guido/
26 Asynchronous JavaScript and XML

5 EXPERIMENTACIÓN PRÁCTICA

ÍNDICE

Pero este proyecto de investigación no se quería dejar únicamente en un plano teórico. De hecho este proyecto nació porque en nuestro trabajo diario con las tecnologías de la Web Semántica detectamos una necesidad tímidamente cubierta y a la que era necesario prestarle atención. Por tanto se puede decir que trioo es un proyecto de investigación "*para desarrolladores, por desarrolladores*". A lo largo de los teóricas extraídas en los Capítulos 3 y 4 nos hemos preocupado de extraer unas conclusiones sólidas que posibilitaran la creación de soluciones software que simplifiquen el código necesario para acceder a la Web de Datos. Tomando como punto de partida las elecciones tomadas en la Sección 4.2, este capítulo describe las implementaciones realizadas a modo de experimentación software. Para tal objetivo, el nivel de detalle acerca de las implementaciones se ha ajustado para tratar de describir de la forma más concisa posible. Por tanto no lugar para dar aspectos como el código fuente. Ese tipo de detalles ya se difundirán por otros foros, liberando el código en la página web del proyecto.

Todas las implementaciones, tanto actuales como las que se desarrollen en un futuro, se encontrarán disponibles de forma libre en la página Web del proyecto[1].

5.1 JTRIOO

Jtrioo[2] es la implementación de trioo en Java. Hay determinados detalles de implementación que son interesantes discutir.

1 http://trioo.wikier.org/
2 http://jtrioo.googlecode.com/

5.1.1 Anotaciones

Las anotaciones fueron introducidas en la versión 5.0 del lenguaje Java [22]. Se trata de una sintaxis especial para añadir meta-información al código fuente Java, que puede ser consultada reflectivamente en tiempo de ejecución. Por tanto parece un mecanismo idóneo para añadir la meta-información que se necesita en jtrioo. La Figura 48 ilustra el uso de estas anotaciones en una sencilla clase Java.

```
1 @RdfResource(rdftype="foaf:Person")
2 public class Person {
3
4     @RdfProperty(type="foaf:name")
5     private String name;
6
7 }
```

Figura 48: Ejemplo sencillo de clase Java anotada con jtrioo

Pero el acceso a las anotaciones en Java es muy limitado, dado que sólo soporta tres niveles de retención[3][4]: source (visibles en los fuentes, pero descartadas por el compilador), class (accesibles por el compilador al compilar la clase, pero no se mantienen en tiempo de ejecución), runtime (accesibles para lectura en tiempo de ejecución). Por tanto por ahora en Java no se contempla que de manera reflectiva se modifique el estado de las anotaciones. Pero atendiendo al requisito descrito en la Sección 4.1.5, hay que buscar un mecanismo que permita *saltarse* esta limitación y permitir que el mapeo con anotaciones en Java permita cierta dinamicidad. Para ello se creo una anotación adicional, `DynamicRdfProperty`, que permite cierta dinamicidad en la definición de los mapeos, tal y como se puede ver en la Figura 49. Dado que los requisitos que por ahora se han extraído para tal funcionalidad se restringen al tipado de la relación, por ahora sólo se da soporte a ese necesidad concreta.

```
1 @RdfResource(rdftype="foaf:Person")
2 public class Person {
3
4     //...
5
6     @DynamicRdfProperty(typeAt="reltype")
7     private Person rel;
8
9     private String reltype;
10
11 }
```

Figura 49: Use de mapeos dinámicos mediante anotaciones en Java.

El catalogo de anotaciones se compone de:

- `RdfResource`[5]: es la anotación principal que identifica que un objeto Java se puede mapear a un recurso RDF. Dado que el tipado RDF se ha

3 http://download.oracle.com/docs/cd/E17476_01/javase/1.5.0/docs/api/java/lang/annotation/Retention.html
4 http://download.oracle.com/docs/cd/E17476_01/javase/1.5.0/docs/api/java/lang/annotation/RetentionPolicy.html
5 org.fundacionctic.jtrioo.annotations.RdfResource

sacado fuera del sistema de tipos de Java, esta anotación permite mapear una clase Java con n clases RDFS/OWL.

- `RdfProperty`[6]: es la anotación que permite indicar que un atributo sustenta también una relación en un grafo RDF. Al contrario que otras herramientas analizadas en el Capítulo 2 (por ejemplo JenaBeans), este mapeo be hacerse de forma explícita, dando mayor flexibilidad en el mapeo.
- `Literal`[7]: anota una atributo para un literal RDF, un tipo de dato simples en definitiva (primitivos, fechas, etc).
- `DynamicRdfProperty`[8]: provee una forma de dar soporte a los descrito en Sección 4.1.5 acerca de la dinamicidad en el mapeo.
- `URI`[9]: provee una forma de declarar como se compone las URIs de los recursos en función del valor de determinados atributos en los objetos, que como es lógico previamente son codificados para formar una URI válida.

Tanto `Literal` como `DynamicRdfProperty` son en realizad especializaciones de `RdfProperty`. Aunque dado que las anotaciones en Java no soportan herencia, esta relación no se encuentra explícita en el lenguaje, con los problemas de implementación que ello conlleva.

5.1.2 Registro de clases

El paquete `java.lang.reflection` [69] provee al lenguaje Java de una potente API de reflectividad [65]. De esta manera se puede interrogar en tiempo de ejecución a los objetos sobre su estructura y estado, y, por ejemplo, obtener las anotaciones anteriormente descritas. Lamentablemente acceder reflectivamente tiene un alto coste[10], por lo que la primera ver que una clase que representa a un recurso RDF es utilizada, se registra un `RDFMetaResource`[11] que contiene una meta-descripción que evita parte de estos accesos. Hay que resalta que este registro de meta-clases no sólo se hizo por temas de rendimiento, algo que no entra dentro de las necesidades del proyecto, sino que también se realizo por comodidad a la hora de implementar otros componentes de la biblioteca y evitar repetir tener que utilizar el API de reflectividad de forma disgregada por diferentes partes del código. Este paquete incluye determinados componentes para describir abstractamente los recursos RDF en Java de una forma lo más flexible posible.

5.1.3 API

Por tanto, una vez ya introducidos algunos aspectos preliminares, ya estamos en disposición de describir brevemente el API. A semejanza de otras APIs

6 `org.fundacionctic.jtrioo.annotations.RdfProperty`
7 `org.fundacionctic.jtrioo.annotations.Literal`
8 `org.fundacionctic.jtrioo.annotations.DynamicRdfProperty`
9 `org.fundacionctic.jtrioo.annotations.URI`
10 Aunque en las últimas versiones de Java (>=5.0) el rendimiento de este paquete ha mejorado enormemente. Tanto que igual habría que replantearse este mecanismo. No se han encontrado razones contrastadas para justificar tal mejoría, pero las sospechas mayoritariamente apuntan hacia una mejora de la compilación JIT dentro del HotSpot de Java.
11 `org.fundacionctic.jtrioo.rdf.meta.RDFMetaResource`

```
@RdfResource(rdftype="foaf:Person")
@URI(base="http://example.org/person/", parts="name")
public class Person {

    @RdfProperty(type="foaf:title", optional=true)
    private String title;

    @Literal(type="foaf:name", lang="es")
    private String name;

    @RdfProperty(type="foaf:knows", fetch=FetchType.LAZY)
    private List<Person> knows;

    //...

}
```

Figura 50: Ejemplo de clase anotada con jtrioo.

de persistencia en Java, la interacción se centra principalmente en un gestor, el `ResourceManager`[12], que es el encargado de lidiar con el grueso de tareas. Considérese la clase de la Figura 50, un ejemplo de cómo utilizar jtrioo sería similar a lo recogido en la Figura 51.

```
ResourceManager rm = ResourceManagerFactory.
    getResourceManager();

Person sergio = (Person)rm.find("http://www.wikier.org/foaf#
    wikier");

System.out.println(sergio.getName());
for (Person friend : sergio.getFriends()) {
    System.out.println(friend.getName());
}
```

Figura 51: Uso de jtrioo.

5.1.4 Interceptación de métodos

Teniendo en cuenta una de las clasificaciones de la reflectividad computacional [77], a priori, y teniendo en cuenta lo que se refleja, Java sólo dispondría de capacidad para poder inspeccionar u observar, pero no modificar los objetos; en definitiva introspección [44], el nivel más bajo de reflectividad. Pero posteriormente, mediante el uso patrón Proxy [47][13], este paquete ha añadido cierta reflectividad computacional, mediante lo que en Java se conoce como "Dynamic Proxy Classes" [70], permitiendo por tanto modificar la propia semántica del software. Y es el mecanismos utilizado en jtrioo para interceptar la invocación a métodos, y por ejemplo manejar eficientemente las colecciones según las políticas definidas. Concretamente se ha recurrido a la implementación proveída cglib[14], que permite realizar de una manera más sencillas algunas tareas ciertamente complejas.

12 `org.fundacionctic.jtrioo.ResourceManager`
13 `java.lang.reflection.Proxy`
14 `http://cglib.sourceforge.net/`

5.1.5 Generación de consultas

La biblioteca utiliza únicamente SPARQL para el acceso a datos, tanto para lectura como para escritura. Las consultas son generadas dinámicamente en tiempo de ejecución utilizando reflectividad sobre los RDFMetaResource y los objetos involucrados, según una serie de políticas definidas que pueden ser personalizadas por el usuario. Para ello se emplea como base una implementación reflectiva del patrón Visitor [47], que se especializa tanto para generar la sintaxis abstracta de la consulta, como para serializarla a SPARQL. Por ahora la biblioteca general SPARQL 1.0 para consultas `SELECT`, y SPARUL [99] para consultas `INSERT`, `MODIFY` y `DELETE`. Esta decisión se ha tomado dado en base a lo inestable que es la sintaxis de SPARQL 1.1 en el momento de realizar esta implementación, por lo que no hay todavía implementaciones del mismo. Pero cuando haya un borrador de trabajo más estable y las implementaciones comiencen a darle soporte, tan sólo sería necesario actualizar la parte relativa a generación del código de las consultas, el resto de componentes no se tendrían porqué ver afectados.

5.1.6 CURIEs

Para administra mapeo de CURIEs la biblioteca viene con una serie de prefijos comúnmente utilizados cargados por defecto (`xsd`, `rdf`, `rdfs`, `owl`, `sioc`, `foaf`, etc..), pudiendo lógicamente el usuario añadir los que necesitara a través de una clase de utilidad llamada `NamespacesManager`[15]. El uso de una CURIE que no se encuentre registrada produce el lanzamiento de una excepción[16].

5.1.7 Propiedades multivaluadas

Dado que el uso de colecciones en RDF [59] no se encuentra demasiado extendido[17], el comportamiento por defecto para serializar a RDF colecciones Java es mediante el uso de propiedades multivaluadas.

Adicionalmente, la biblioteca lanza una excepción[18] cuando más de un atributo de una clase se encuentra anotado con la misma CURIE. Se advierte al usuario que para tal caso debe hacer uso de alguna de las colecciones disponibles en la biblioteca estándar de Java.

5.2 PRYOO

Dadas las conclusiones de los capítulos anteriores, era necesario evaluar el concepto tras trioo en un lenguaje basado en prototipos. Y pudiera decirse que todos los indicadores (popularidad, características, usabilidad, experiencia, etc) apuntaba hacía el mismo candidato: el lenguaje de programación Python. Por tanto pryoo[19] se convertiría en la segunda implementación de trioo . Hay que reseñar que debido a las fuertes restricciones temporales que ha habido

15 `org.fundacionctic.jtrioo.rdf.NamespacesManager`
16 `org.fundacionctic.jtrioo.exceptions.MappingNotFoundException`
17 http://www.w3.org/2001/sw/wiki/RDF_Core_Work_Items#Data_Model_Issues
18 `org.fundacionctic.jtrioo.exceptions.DuplicatedPropertyException`
19 http://pryoo.googlecode.com/

en este trabajo, quizá se haya comenzado un poco tarde esta implementación como para extraer unas mejores conclusiones. De todos modos es interesante comentar lo avanzado hasta el momento.

Para comenzar hay determinados detalles de la implementación que merecen la pena ser comentados:

5.2.1 Versión del lenguaje

En el momento de comenzar con esta implementación, el lenguaje Python se encuentra inmerso en una transición del lenguaje, de las versiones 2.x a la versión 3.x, con determinados e interesantes cambios[20] que hacen la nueva versión incompatible hacia atrás. Por tanto la elección de la versión (más bien rama) a utilizar es algo a valorar detenidamente.

```
1 @RdfResource("foaf:Person")
2 class Person:
3
4     @RdfProperty("foaf:name")
5     name = None
6
7     (...)
```

Figura 52: Primer intento para definir el API de pryoo.

Para comenzar, con el fin de no confundir a los usuarios y facilitar que los desarrolladores *salten* de una implementación a otra con la curva de aprendizaje menos pronunciada posible, el API en Python debiera parecerse lo más posible a su equivalente en Java descrita anteriormente. Eso es algo como lo que se muestra en la Figura 52. Cabe señalar que al contrario de lo que ocurre en Java donde las anotaciones no son más que metadatos que pueden consultarse reflectivamente en tiempo de ejecución, las anotaciones en Python son azúcar sintáctico que realmente provee una implementación genérica del patrón Decorator [47], permitiendo actuar activamente sobre el elemento anotado. Lamentablemente las construcciones utilizadas en la Figura 52 no son válidas en Python: para empezar Python 2.x sólo permite utilizar anotaciones en funciones o métodos, no en atributos; sin embargo en Python 3.x es posible anotar una clase. Por tanto, utilizando Python 3.x habría que moverse a algo como lo que se muestra en la Figura 53, o incluso algo como la Figura 54.

```
1 @RdfResource("foaf:Person")
2 class Person:
3
4     @RdfProperty("foaf:name")
5     def get_name(self):
6         return self.name
7
8     (...)
```

Figura 53: Segundo intento para definir el API de pryoo.

Ambas aproximaciones (Figura 53 y Figura 54) si nos lanzásemos a Python 3,x. Pero es necesario valorar otros factores, quizás el más importante para

20 http://docs.python.org/dev/3.0/whatsnew/3.0.html

```
@RdfResource("foaf:Person")
class Person:

    name = RdfProperty(curie="foaf:name")

    (...)
```

Figura 54: Tercer intento para definir el API de pryoo.

agilizar el desarrollo sea elegir bien las bibliotecas que va a necesitarse utilizar. Lamentablemente, aunque en muchos casos debiera ser trivial[21] demasiadas dependencias (RDFLib[22], SPARQLWrapper[23], etc) aún no han sido portadas a la última versión del lenguaje. Por tanto, dado que en los plazos temporales que nos movemos será complicado tener portadas estas bibliotecas a Python 3.x[24], parece claro que habrá que decantarse por Python 2.x para los primeros pasos del desarrollo de la implementación de referencia de trioo en Python. La versión 2.7 está llamada a ser la última actualización de esta rama, ya que incorpora muchas novedades[25] de 3.0 y 3.1, pero dado el estado de desarrollo en el que aún se encuentra[26] [80]. Aunque lógicamente esto jamás implica perder de vista Python 3.x, pues una vez se pase la moratoria, será la versión de referencia en el lenguaje y habrá que actualizar la biblioteca en algún momento. Cuando llegue ese momento, se estudiará las consecuencias de una cambio del API.

5.2.2 API

```
class Person(RdfResource):

    name = RdfProperty(curie="foaf:name")
    knows = RdfProperty(Person, curie="foaf:knows")

    def __init__(self):
        RdfResource.__init__(self, "foaf:Person")

    (...)
```

Figura 55: Aspecto definitivo del API de pryoo.

Por tanto el aspecto definitivo del API de pryoo quedaría tal y como se muestra en la Figura 55. Para quienes estén familiarizados con otros frameworks en Python, su uso es muy similar por ejemplo al API de modelos de Django (ver Sección 2.3.1). Pero si nos fijamos con más detalle en el ejemplo de la Figura 55, notaremos que en el API no se hace distinción entre colecciones o elementos simples, como si es necesario hacer en otras APIs que trabajan sobre

21 http://diveintopython3.org/porting-code-to-python-3-with-2to3.html
22 http://www.rdflib.net/
23 http://sparql-wrapper.sourceforge.net/
24 Aunque intenciones hay: https://sourceforge.net/mailarchive/message.php?msg_name=AANLkTinyzAF7mMg7o6ILCOYGm8XpaKvmshivqZcQTQYa%40mail.gmail.com
25 http://docs.python.org/dev/whatsnew/2.7.html
26 La Release Candidate 2 todavía ha sido publicada el 20 de junio de 2010.

otros tipos de modelos. Esto es debido a que se aprovecha tanto la flexibilidad del modelo de datos como la del tipado dinámico del lenguaje para tratar de que el usuario (desarrollador) no tenga que tomar este tipo de decisiones si no es necesario. Para lograr este efecto se varía dinámicamente el tipo y el comportamiento del atributo según el estado del modelo de datos RDF, y dicho atributo acepta tanto los métodos de un tipo primitivo como los de una lista. El ejemplo de uso que se muestra en la Figura 56 muestra tres forma diferentes de utilizar pryoo: primeramente utilizando la definición de la clase de la Figura 55 para crear un nuevo recurso RDF, luego tratando de obtener todos los individuos que *encajen* con la definición de esa clase, y en tercer lugar construyendo un nuevo objeto desde cero de manera reflectiva a partir de la URI de un recurso RDF[27].

```
>>> sergio = Person("sergio")
>>> print sergio
Person: sergio
>>> sergio.save()

>>> for (person in  Person.find_all()):
...     print person
Person: xxx

>>> wikier = RdfResource.retrieve("http://www.wikier.org/foaf
    #wikier")
>>> print wikier
FOAFPerson: Sergio Fernandez
>>> print wikier.foafname
Sergio Fernandez
>>> print wikier.__class__
<class '__main__.FOAFPerson'>
>>> print wikier.rdftype
foaf:Person
```

Figura 56: Uso de pryoo.

5.2.3 Meta-programación

La meta programación ha sido un elemento que siempre ha estado latente en la implementación en Python dee trioo. Tal y como dice Tim Peters[28], "Las metaclasses son tan mágicas, que el 99 % de los usuarios de Python debieran preocuparse de ellas". Pero aunque esto sea verdad, desde mi humilde opinión no son intuitivos para los desarrolladores finales, de ahí que en el API propuesta en la Figura 55 no se *vea* ningún resquicio de meta programación. Por tanto la idea es que realice una herencia normal, y no modifique directamente el atributo `__metaclass__`[29]

27 Las URIs son automáticamente traducidas a una especie de CURIE utilizando el mapeo de prefijos proveído en la configuración y en el servicio `prefix.cc`, aunque sin separador para que sea nombres válidos para clases y atributos en Python.

28 http://uswaretech.com/blog/2009/11/the-magic-of-metaclasses-in-python/

29 Esto cambia en Python 3.x.

5.2.4 Asimetría __getattr__/__setattr__ en Python

No es un detalle de implementación particular de pryoo, pero dado que se usa intensivamente es relevante documentarlo brevemente[30]:

El API de reflectividad de Python es extremadamente potente y flexible. En Python cuando se le pide un atributo/método a un objeto, en realidad el método `__getattr__` es el encargado de buscarlo, bien en el propio objeto instancia, en su jerarquía de clases. Este método devuelve el atributo (o método) si lo encuentra, o lanza una excepción `AttributeError`. Para evitar llamadas recursivas y provocar errores en caso de no encontrarse, el modelo reflectivo de Python presenta una asimetría entre `__getattr__` y `__setattr__`: `__getattr__` no se llama si el atributo no se encuentra por las vías convencionales. Para tener control total de la operación, necesario recurrir al método `__getattribute__`[31] . Para evitar esos problemas de recursividad infinita, este método siempre debe invocar el método de la clase base.

```
1 self.__dict__['__setattr__'] = MethodType(set_attr, self,
      self.__class__)
2 self.__dict__['__getattribute__'] = MethodType(get_attr, self
      , self.__class__)
```

Figura 57: Tratamiento de la asimetría __getattr__/__setattr__ en pryoo.

Por tanto, como se puede ver en la Figura 57, ha sido necesario tener en cuenta tal asimetría en algunas de las operaciones reflectivas que realiza pryoo internamente.

5.3 CONCLUSIONES EXTRAÍDAS

Como era de esperar, la experimentación práctica hacer salir a la luz muchos detalles interesantes que sobre el papel pueden no llegar a verse. Lamentablemente en ese aspecto el tiempo a jugado en contra. La implementación en Java se encuentra mucho más madura, en parte gracias al soporte de herramientas de terceros. Mientras que los resultados de la implementación en Python son esperanzadores, y será la linea que habrá que motivar más en futuras acciones.

Aunque se ha intentado que las implementaciones fueran lo más usables posibles, ello no implica que el objetivo haya sido producir implementaciones maduras listas para ser utilizadas fuera de entornos de laboratorio. En ese sentido hubiera sido muy interesante cubrir otros lenguajes de programación (por ejemplo (C# o JavaScript) cuyas características los convierten en lo siguientes candidatos en un futuro a medio plazo.

Respecto a los resultados extraídos con las dos implementaciones realizadas, como es de suponer, poner en práctica los conceptos estudiados a lo largo de los capítulos anteriores te lleva a validarlos realmente. Y no sólo validarlos internamente, sino que te permite validar si el concepto tras trioo tiene buena acogida por otros desarrolladores. Lamentablemente por falta de tiempo estas pruebas no se han hecho de manera sistemática, sino más bien de una manera

30 Fuentes: http://pyref.infogami.com/__getattr__ y http://pyref.infogami.com/__setattr__

31 http://pyref.infogami.com/__getattribute__

un tanto informal con compañeros del trabajo; por tanto lamento decir que estas pruebas no han sido propiamente documentadas.

6 CONCLUSIONES

Ciertamente me resulta complicado extraer una conclusiones tangibles de forma global de este trabajo de investigación. Para conclusiones más particulares, consúltese las breves conclusiones que hay al final de cada capítulo. Pero globalmente es diferente. El ejercicio de asimilar el modelo de datos RDF en un lenguaje orientado a objetos han resultado algo mucho más complejo de lo que nunca pudiera imaginarme al principio. A medida que se iba profundizando en algunos aspectos, se han ido abriendo muchos nuevos frentes donde es necesario realizar una investigación más pausada y profunda. Este amplio abanico de aspectos ha sido algo complicado de gestionar, debido a las restricciones temporales de que se disponía. Pero se ha tratado de focalizar el trabajo en aquellos que se creía podían tener mejores resultados, y por tanto un mayor impacto.

Debido a esto se ha puesto especial énfasis en el estado del arte, en el cual hay muchas e interesantes propuestas. Si bien es cierto que quizá demasiadas de ellas excesivamente *viciadas* por otras tecnologías (principalmente el modelo entidad-relación). Probar todas y cada una de estas herramientas ha supuesto un importante inversión de recursos, pero que ha dado sus frutos, pues de de casi todas ellas se ha extraído algo positivo e interesante que se ha tratado de aplicar en las implementaciones que posteriormente se ha producido. Las cuales han dado buena cuenta de los problemas de integración que hay entre ambos modelos de datos. Si bien es verdad que ambas implementaciones no se encuentran en un avanzado estado de madurez, tampoco era el objetivo, ambas demuestran que es viable integrar parcialmente RDF en ambas familias de lenguajes orientados a objetos, cada una de ella con sus particularidades y limitaciones. En este sentido será interesante avanzar tanto en estabilizar las implementaciones actuales, como en llevar trioo a otros escenarios donde se puedan probar más aspectos.

La difusión científica es algo muy importante en cualquier ejercicio de este tipo. Para optimizar este aspecto, y dados los cortos plazos en que nos movíamos, en otoño de 2009 se planificó una estrategia a este respecto. No se quería ir a una conferencia pura de semántica, sino más bien se quería buscar un foro mixto de lenguajes y semántica. En esta línea la ICSOFT[1] 2010 (5th International Conference on Software and Data Technologies[2]) encajaba perfectamente: por tópicos, por fechas para un position paper (el dealine inicial era el 1 de febrero de 2010, aunque luego fue extendido en dos ovaciones), lugar (fuera de Europa sería complicado encontrar financiación para viajar), etcétera. Y parece que la elección fue acertada, pues el artículo fue aceptado con bastantes buenos comentarios por parte de los revisores. Para su consulta, el Anexo A reproduce la versión final del artículo que será publicado por las actas de la conferencia.

Recapitulando... A mi modo de ver este documento no marca un hito final en esta investigación, sino más bien todo lo contrario. Simplemente se han sentado las bases, unas bases sólidas en una linea de investigación que se comenzaba

1 International Conference on Software and Data Technologies
2 http://www.icsoft.org/

desde cero. Una linea que se ha desvelado como un campo realmente duro en el que todavía quedan muchísimas cosas por investigar en los próximos años. Aunque es algo que depende de muchos otros factores, personalmente mi intención es darle continuidad a esta linea y profundizar hacia la posibilidad de tener una integración de la semántica de ambos modelos computacionales.

A PUBLICACIONES REALIZADAS

Dentro de la metodologías de trabajo seguida en este trabajo de investigación, se ha tenido muy en cuenta desde un principio la necesidad de publicar los resultados alcanzados en el mismo. Para ello, en fases muy tempranas, se realizó una selección de foros relevantes, tanto conferencias como revistas. Dados los limitados recursos temporales de los que se disponía, finalmente tan sólo se ha realizado una contribución a una conferencia internacional, tal y como se describirá con mayor detalle a continuación. De todos modos, dado que el interés del autor es continuar la linea iniciada por este trabajo, se confía en poder llevar finalmente los resultados de esta investigación a foros de mayor impácto científico.

A.1 TRIOO, KEEPING THE SEMANTICS OF DATA SAFE AND SOUND INTO OBJECT-ORIENTED SOFTWARE

El artículo titulado "*TRIOO, Keeping the Semantics of Data Safe and Sound into Object-Oriented Software*" [41] ha sido aceptado para su publicación como resultado del proceso de revisión llevado a cabo por el comité científico del ICSOFT2010[1] (5th International Conference on Software and Data Technologies) dentro del área de *Data Management*. El artículo, que describe la posición e intenciones de este proyecto, ha contado con la colaboración, ademas de los directores de este proyecto de investigación José Emilio Labra Gayo y Diego Berrueta Muñoz, del doctorando de la Universidad de Oviedo Miguel García Rodríguez.

A continuación se reproduce la versión final del artículo publicado por las actas de la citada conferencia.

1 http://www.icsoft.org/

TRIOO
Keeping the Semantics of Data Safe and Sound into Object-Oriented Software

Sergio Fernández, Diego Berrueta
Fundación CTIC, C/ Ada Byron 39, Parque Científico y Tecnológico, Gijón, Asturias, Spain
sergio.fernandez@fundacionctic.org, diego.berrueta@fundacionctic.org

Miguel García Rodríguez, Jose E. Labra
Computer Science Department, Universidad de Oviedo, Campus Los Catalanes, Oviedo, Asturias, Spain
be37378@uniovi.es, labra@uniovi.es

Keywords: RDF, SPARQL, Web, data, persistence, object-oriented programming

Abstract: Data management is a key factor in any software effort. Traditional solutions, such as relational databases, are rapidly losing weight in the market towards more flexible approaches and data models due to the fact that data stores as monolithic components are not valid in many current scenarios. The World Wide Consortium proposes RDF as a suitable framework for modeling, describing and linking resources on the Web. Unfortunately the current methods to access to RDF data can be considered a kind of handcrafted work. Therefore the Trioo project aims to provide powerful and flexible methods to access RDF datasets from object-oriented programming languages, allowing the usage of this data without negative influences in object-oriented designs and trying to keep the semantics of data as accurate as possible.

1 INTRODUCTION

Software data management has experienced a notable evolution in the last few years. Relational databases are still one the most powerful and used solutions for data storage in many cases, but there are many reasons to discard it as a suitable technology: On the one hand, the emergence of new distributed architectures of software requires new approaches able to fulfil all these new requirements relying on availability and flexibility. Unfortunately federated databases do not yield the expected features and performance in some scenarios. On the other hand, it is impossible to normalise a relational schema when data schema is not clearly decided or condensed. Both issues have motivated a lot of research in the last few years. Concerning distributed storage, the software industry currently offers some quite interesting solutions: Google uses Bigtable or Pregelin many of their applications, Microsoft developed BitVault, among others. With respect to how to efficiently store schema-free data, there are other promising proposals facing this challenge: from proprietary data models, as solutions used in many of the distributed storage systems or in native object-oriented databases, to more open solutions, such as the XML-based databases.

The World Wide Consortium[1] adopted another approach, often known as the Semantic Web (Berners-Lee et al., 2001). Semantic Web aims to provide a set of technologies that would facilitate the implementation of solutions to this (and others) current problems, based on the decentralised Web architecture, the idea of a "*Web of Data*". Obviously it has some additional open issues, such as the trust and provenance of data, that do not apply to an intranet environment; but once they are solved, the "*Web of Data*" reveals as a much more powerful solution than a one. W3C's main proposal is build upon the current Web. So that means using HTTP (Fielding et al., 1999) as basic protocol for transfer information and XML as basic format for encoding documents, but enriched with the RDF technologies stack. RDF (Klyne and Carroll, 2004) (*Resource Description Framework*) provides a flexible and extensible data model, based on the concept of "triple" `(subject, predicate, object)`, that greatly simplifies data interoperability, interchange and integration across different heterogeneous sources and applications. RDF extends the linking structure of the Web to

[1] `http://www.w3.org/`

use URIs to name the relationship between things instead of documents (Berners-Lee et al., 2005; Sauermann and Cyganiak, 2008). This architecture is complemented with languages to model knowledge (RDFS (Brickley et al., 2004) and OWL (Schneider et al., 2009)) and languages to natively query RDF data such as SPARQL (Prud'hommeaux and Seaborne, 2008)/SPARUL (Schenk and Gearon, 2010).

Consequently RDF allows to relate data from different providers, interlinking resources from different datasources, as the "*Linked Data*" principles (Berners-Lee, 2006) dictate. The Linked Data initiative[2] is growing quite fast, and a large amount of data is already available on the Web as RDF, ready to be consumed by new applications capable of exploiting this innovative paradigm. This means a new way to integrate the Word Wide Web with business data and applications. Due to these features, RDF may be a key technology to store the data in the next generation of software applications, becoming a common layer where data can be stored and retrieved.

In this position paper we present the Trioo project (which name means something like "*triples object-oriented*"), an ongoing initiative that aims to add support to some of the most important object-oriented programming languages for managing RDF data based on those standard technologies in an efficient and flexible way. Because at the end "are objects, not triples"[3]. A key premise of the project is that software developers must not be forced to adapt their oo-design keeping the semantics of data safe and sound in the software.

This paper is structured as follows: Section 2 analyses the related work relevant to the objectives addressed by Trioo. Section 3 describes in detail these objectives and the design issues found, facing and comparing both computational models, RDF and object-oriented. Section 4 briefly depicts the ongoing implementations in the current early state of development. Finally Section 5 discusses the current conclusions, setting the basic guidelines for the work in the upcoming months.

2 RELATED WORK

The ongoing work described in this paper is extensively related to the work made these last years on object-relational mappings, but also to work in accessing (Semantic) Web data. On these fields, and for better understating of this analysis, it is necessary to introduce two relevant design patterns:

Active Record is a widely used design pattern to work with relational databases where each row is represented as a single object (Fowler, 2002). Therefore, ActiveRecord can be considered as an approach to access databases: each table (or view) is *wrapped* as a class, and each row as a instance object. The assumption "*one object, one row*" greatly simplifies the development of CRUD functions (Create, Read, Update and Delete) on relational databases. In fact, most of the current ORMs ("*Object-Relational Mapping*") implements this design pattern, or an adaptation of it.

Data Mapper is an adaptation of the Mapper design pattern (Fowler, 2002). Objects and relational databases have different mechanisms for structuring data, and many parts of an object, such as business logic, collections and inheritance, are not present in relational databases. The object schema and the relational schema do not need to match up. The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects do not need to know even that there is a database present; they do not need SQL interface code, and certainly no knowledge of the database schema; in fact the database schema is always ignorant of the objects that use it.

These two design patterns are probably the most relevant and used for object-data mapping, but there are many more, such as "*Table Data Gateway*" or "*Row Data Gateway*", among others. However this is not a full report about these patterns, just a brief introduction of the concepts, in order to illustrate how data access use to be implemented in modern software systems. Although the target of the Trioo project are just RDF-based stores, one of the aims of this position paper is to extract (positive and negative) conclusions from other current solutions. Therefore this state of the art will not be restricted to RDF stores, but it will cover a wider range of approaches in order to reach the best possible solution in Trioo.

2.1 Traditional stores

In this paper the term "*traditional stores*" actually concerns traditional relational databases. There are several software solutions that solve the object-relational impedance mismatch problems

[2] http://linkeddata.org/

[3] http://harth.org/andreas/blog/2009/03/27/its-objects-not-triples/

by replacing direct persistence-related database accesses with high-level object handling functions. Some of the most relevant are (by alphabetic order):

ActiveRecord[4] is the implementation of the pattern with the same name used in the popular Web framework Ruby on Rail, which main contribution to the pattern is to relieve the original of two stunting problems: lack of associations and inheritance. By adding a simple domain language-like set of macros to describe the former and integrating the Single Table Inheritance pattern for the latter, Active Record narrows the gap of functionality between the data mapper and active record approach. It follows the principle DRY philosophy[5]. There are many other implementations for many other languages that follow a quite similar approach, such as Django Models for Python, nHydrate and Castle for C# (this last one actually an implementation built on top of NHibernate), CakePHP for PHP, or GORM for Groovy.

Hibernate originally is an implementation of the Data Mapper design pattern for the Java language, but currently is much more, providing a framework for mapping an object-oriented domain model to a traditional relational database (Bauer and King, 2004). The current version is a certified implementation of the Java Persistence API 1.0 specification (JPA (DeMichiel and Keith, 2006)), but hopefully soon it will start to support the recent version 2.0 (JSR317 (DeMichiel, 2009)). Hibernate also provides a SQL inspired language, called Hibernate Query Language (HQL), which allows SQL-like queries to be written against Hibernate's data objects. Undoubtedly, Hibernate is the most extended data mapper for the Java world. A port for the .NET framework, known as NHibernate[6], has been even carried out. Nowadays JPA has become so popular that there are several implementations, although Hibernate is still the most popular one.

iBATIS is an implementation in Java, .NET and Ruby of the Active Record design pattern, providing a persistence framework which automates the mapping between relational databases and objects. It takes the reverse approach than, for instance, Hibernate: iBatis automates the creation of POJOs (Plain Old Java Objects) from an already existed relational database.

[4] http://ar.rubyonrails.org/

[5] Don't Repeat Yourself

[6] https://www.hibernate.org/343.html

All these solutions allow to develop persistent classes over relational databases following object-oriented idiom, including association, inheritance, polymorphism, composition, and collections management.

2.2 RDF-based stores

The critical problem of how to efficiently query RDF datasets has been there since the origins of the Semantic Web until today (Sintek and Decker, 2002). It is true that the current technological landscape has substantially improved: there is a standard query language (SPARQL), efficient RDF stores (Virtuoso[7], Sesame[8], 4Store[9], and many others), and libraries that support all this technology. However there is still a huge gap on how all this RDF-based data is actually managed on the software systems. As described above, a RDF class is not exactly the same that a class in object-orientation, even though there are some solutions based on this approach[10]. Anyway there are some interesting attempts to be analysed:

ActiveRDF proposed a novel way to work with RDF with object-orientation on Ruby (Oren et al., 2008), based on the ActiveRecord design pattern (Fowler, 2002), and clearly influenced by the work on model-driven Web development. When it was developed, SPARQL did not have data update capabilities, so it works over proprietary interfaces (based on a pluggable architecture that allows new implementations) for each store for persisting the RDF data the query language has evolved since then, but apparently there is not any plan to support its new features.

Empire[11] is an implementation of a large chunk of the core of JPA to provide an interface to RDF databases (currently only 4Store, Sesame, and Jena) using SPARQL, SeRQL (Broekstra and Kampman, 2003) and the supported stores' proprietary API, by a small annotation framework for tying Java beans to RDF.

[7] http://virtuoso.openlinksw.com/

[8] http://www.openrdf.org/

[9] http://4store.org/

[10] Such as the one use by Soprano (http://soprano.sourceforge.net/apidox/trunk/soprano_devel_tools.html#onto2vocabularyclass) or by Protégé (http://protegewiki.stanford.edu/index.php/JSave)

[11] http://github.com/clarkparsia/Empire

JenaBean[12] persists POJOs to Jena's models[13], and back. Although it is annotation-based, requires to extend a templated class (RdfBean) to gain RDF capabilities, which is a heavy design restriction on programming languages with single inheritance like Java.

OntoBroker originally consists of a number of languages and tools that enhance query access and inference service of the World Wide Web (Fensel et al., 1998). This is based on the use of ontologies to annotate Web documents and to provide query access and inference service that deal with the semantics of the presented information. Nowadays OntoBroker[14] has become a consolidated commercial product that provides a high performance and scalable Semantic Web reasoning middleware, supporting several standard technologies such as OWL, RDF, RDFS, SPARQL or F-logic.

Ripple is a relational, stack-based data-flow language for the Semantic Web (Shinavier, 2007), a versatile framework for traversal-based algorithms on semantic networks. The open source implementation[15] is written in Java and includes a command-line interpreter as well as a query API which only interoperates with the native API of Sesame RDF store.

SuRF[16] is another implementation of the Active Record pattern. It exposes the RDF triple sets as sets of resources and seamlessly integrates them into the object-oriented paradigm in Python, in a similar manner as ActiveRDF does for Ruby. In can work both over files or RDF stores, using SPARQL for reading where possible, but implementing proprietary interfaces for writing data to a couple of natively supported stores.

This analysis draws that many these products use the same approaches for RDF than their equivalents for other technologies. For instance, even though the *Active Record* pattern fits well for relational databases, that is not so for RDF: a RDF object is not a row, it is a set of triples. Thus RDF data model has many differences that should be taken into account for developing such a kind of tools. Therefore it would be necessary learn from other more developed technologies, but it must flee from negative influences. Many of these details will be described below in this paper.

[12] http://jenabean.googlecode.com/

[13] http://jena.sourceforge.net/javadoc/com/hp/hpl/jena/rdf/model/Model.html

[14] http://www.ontoprise.de/en/home/products/ontobroker/

[15] http://ripple.googlecode.com/

[16] http://surfrdf.googlecode.com/

2.3 Other families of stores

Data storage embraces many more technologies than those described above, such as object-oriented databases. Unfortunately object-oriented databases are still minor players with reduced market niches. And meanwhile some of the products described above have added object capabilities to relational databases. In a parallel way there are other alternatives that do not require fixed table schemes, and usually avoid join operations. It is worth highlighting the NoSQL initiative[17], that actually it is not a concrete product, but a portmanteau term for a loosely defined class of non-relational data stores that break with a long history of relational databases and ACID guarantees[18]. In addition, there are two products that are somehow interesting due their adaptability to many different stores:

EclipseLink[19] provides an extensible framework that allows Java developers to interact with various data services, including relational databases, web services, Object XML mapping (OXM), and Enterprise Information System (EIS). EclipseLink supports a number of persistence standards including Java Persistence API (JPA), Java API for XML Binding (JAXB), Java Connector Architecture (JCA) and Service Data Objects (SDO).

LINQ is a general-purpose query language proposed by Microsoft for adding native query capabilities to their .NET framework (Meijer et al., 2006). LINQ applies to all sources of information, not just relational or XML data, by implementing new data providers, even for RDF, such as the fledgling effort on LinqToRdf[20].

[17] http://nosql-database.org/

[18] Atomicity, Consistency, Isolation, and Durability

[19] http://www.eclipse.org/eclipselink

[20] http://linqtordf.googlecode.com/

3 BRINGING TRIOO TO THE ARENA

The Trioo project aims to develop a technology enabling to easily use RDF data directly in some object-oriented programming languages, not only to persist that data in RDF stores, but also to consume data available on the Web and to expose the business model of the application as *Linked Data*. It is true that some languages start to offer new generic mechanisms closer to the object-orientation that to the query technology, such as the above described Microsoft's LINQ; but they still fail to offer more natural ways to access the data, from an object-oriented perspective. This is not a new challenge in computer science, because the software industry already offers some solutions since some years ago, for instance Hibernate for relational databases. But going deeply into the problem, it is clear that there is not any satisfactory solution to work with RDF, because they all strongly rely on design patterns, such as Active Record, too close to the entity-relational model. And the RDF model has many special features and semantics that should be treated as such. Therefore providing a technology capable of managing RDF data in a object-oriented way is a appealing scientific challenge that, from our point of view, may potentially have a great impact on the usage of the Semantic Web to provide innovative software solutions to real world problems.

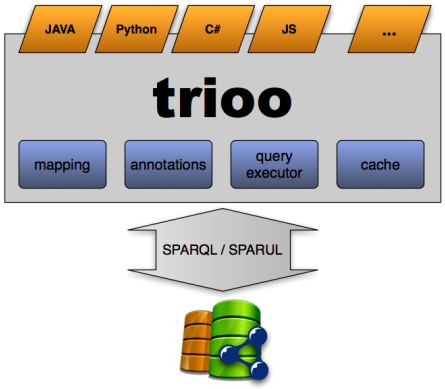


Figure 1: Trioo overview.

Ideally with Trioo developers will be not need to deepen in detail on the RDF data model, abstracting on how the query language internally works. However Trioo would not only take care about data mapping, but also about the typing inference where possible. And probably a cache mechanism would be also required, in any case such a kind of implementation details are not so relevant for the scope of this paper.

3.1 Computational models

Both computational models (RDF and OO) share many concepts. But at the same time they present many substantial differences on their semantics.

In the object-orientation there is no a unique computational (Logozzo, 2004), there are more than one with several implementations. Usually object-oriented languages are divided into two big families (Evins, 1994): class-based and prototype-based programming languages.

Class-based programming languages use a computational model where objects are grouped on sets with equal structure and behaviour as instances of the same class (Booch, 1993). Thus there is a heavy link between an object and its class on instantiation, and it is not possible to create and use an object without previously define its static structure and behaviour on a class.

Prototype-based programming languages aim to be more faithful to the object computational model, where it is only necessary to have objects, no additional abstractions such as classes are required (Blaschek, 1994; Noble et al., 1999; Borning, 1986). The common behaviour is defined by what is known as "*trait objects*" (Ungar et al., 1991). The shared structure is represented as *prototype objects*, which inherit their behaviour from the *trait objects*. Therefore the state is always defined always as objects. The creation of objects (actually *instance objects*) is done using cloning those prototype objects.

So both approaches of object-orientation must be taken into account in Trioo.

RDF semantics is further complicated. It is an assertional language intended to be used to express propositions using precise formal vocabularies (Hayes and McBride, 2004). The semantics of the language is specified using model-theory, a technique which assumes that the language refers to a "world", and describes the minimal conditions that a "world" must satisfy in order to be an interpretation which makes a expression in the language true. The utility of a formal semantic theory is to provide a mechanism to determine valid inference processes, i.e. when the truth is preserved. Through the notions of satisfaction, entailment and validity, RDF semantics gives a rigorous definition of the meaning of a RDF graph. Actually, the RDF semantics specification defines four normative

notions of entailment for RDF graphs: Simple, RDF, RDFS, and Datatype entailment. Simple-entailment captures the semantics of RDF graphs when no attention is paid to the particular meaning of any of the names in the graph (this is precisely the semantics supported by the current version of SPARQL). To obtain a complete meaning of an RDF graph written using a particular vocabulary, it is necessary to add further semantic conditions to interpret particular resources and typed literals in the graph. This way, RDF-entailment provides support for basic entity interpretation using the RDF vocabulary (typing, literals, basic structures, etc.). Furthermore, D-entailment imposes additional conditions on the treatment of datatypes (for instance, for the use of externally defined datatypes identified by a particular URI reference). And finally RDFS Vocabulary extends RDF to include a larger vocabulary with more complex semantic constraints, such as classes, subclasses, domains, ranges and so on (Brickley et al., 2004). RDF graphs (serialised using RDF/XML syntax) are also the normative representation of OWL ontologies for exchange purposes. However, Trioo is not intended to support OWL semantics for now. A full compatibility with OWL would introduce a lot of complexity, as it will require interpreting OWL vocabulary and axioms, and providing support for DL reasoning.

Hence there are many details on these computational models (object-oriented and RDF) that would need to be analysed in detail in order to archive a real integration:

3.1.1 Typing

Typing is an intriguing challenge, since both approaches implementing the object-oriented computational model do it differently. For class-based languages the typing is build upon a hierarchy model of types based on classes (inheritance is defined at that class level, building a hierarchy system of classes). Nevertheless prototype-based languages provide a faithfully implementation of the object-oriented model: the inheritance relationship between objects is actually a derivation relationship, consequently "*instance of*" relationship is no necessary any longer; ideally this derivation could be from more than one object, although commonly this feature is restricted to a single reference in many of the implementations of the prototype-based computational model, such as Python. Typing on RDF works like the pure prototype-based model: each RDF object could have several types, with no restrictions. Figure 2 illustrates these differences on typing. Other concepts, such as inheritance and consistency, only appears with RDFS and OWL. Following section describes how type inference works.

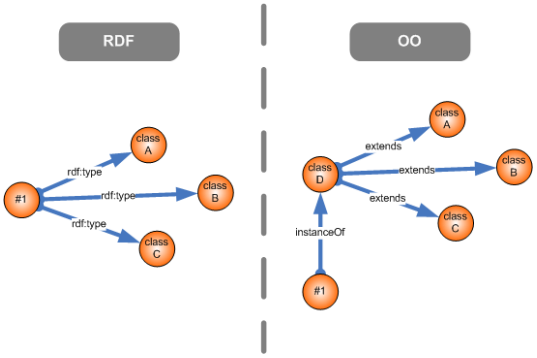


Figure 2: Differences on typing.

3.1.2 Type inference

Type inference is a feature to automatically deduce the type of a value in a programming language, both at compilation-time and run-time depending on the concrete language. It is a common feature in dynamically typed programming languages, but also in some strongly statically ones, such as C#. On programming that inference affects to the type of variables (or attributes in the case of object-oriented programming). By contrast, in RDF relationships are declared using properties. And those properties are global, i.e. the same can be used to link different objects. In RDFS (and OWL) each property can declare both a domain and a range: the type that respectively the subject and the value may have. Therefore given a property P with a domain restriction D and a range restriction R, with a triple (a,P,b) it would be inferred that a is of type D and b of type R. Needless to say that support this behaviour would be difficult on statically typed programming language, but it would be achievable in dynamic languages.

3.2 Expressiveness

As commented above, OWL would be not supported by Trioo. However, Horst proved that entailment for RDFS is decidable, NP-complete, and in P if the target graph does not contain blank nodes. Therefore probably it would more realistic just to use the RDFS semantics (ter Horst, 2005). Moreover, this level of expressiveness could be achieved without perform a full reasoning process, because it could be vastly implemented directly inside the object model (in fact, there is already an implementation in Python of this

concept[21]).

3.3 Integrity Constraints

Integrity constraints are used to ensure accuracy and consistency of data. Those constraints are directly ensured by the engines for other families of stores (for instance, by the entity-relationship model). However, detecting constraint violations is out of the scope of RDF Schema and OWL due to the Open World Assumption adoption. Under OWA, a statement cannot be inferred to be false on the basis of a failure to prove, i.e. if something is not said, it is not possible to know whether it is true or false. As a consequence, everything may be true unless proven otherwise. The OWA assumes incomplete information by default and it is really useful for knowledge reusability in contexts such as Linked Data, limiting the the kinds of deductions just to those that follow from statements that are known to be true. However, with respect to Integrity Constraints, OWL and RDF Schema consistency checking only detects logical inconsistencies in the data, but missing information does not cause an inconsistency. Consider the following RDF graph:

```
foaf:name  rdfs:domain  foaf:Person .
:sergio    foaf:name    "Sergio" .
```

Under OWA, the domain restriction implies that `:Sergio` is a `foaf:Person`, but there is no violation of the domain of the property. Missing values of data do not cause inconsistencies, rather new information of data is inferred from the application of the RDF Schema entailment rules. These inferences may be counterintuitive for users who need data validation, like in some real scenarios and applications. As other frameworks for managing object-oriented domain models with data repositories, Trioo is envisioned to provide support for integrity and validation over RDF data. In order to check these inconsistencies, Trioo must explore how to introduce the Closed World Assumption alternatively to OWA. CWA interpretation means that an assertion is considered false if it is not explicitly known whether it is true or false. This semantics is usually supported by classical logical programming systems implementing Negation as Failure (NAF). There are several ways to come up with Integrity Constraints from the OWL and RDF Schema axioms of an ontology (Sirin and Tao, 2009; Motik and Rosati, 2007). One of the most interesting approaches is the translation of OWL axioms to SPARQL queries (following the entailment regime defined by Sirin et al. (Sirin and Parsia, 2007)). SPARQL is equivalent to non-recursive datalog and NAF may be encoded using the well-known OPTIONAL/FILTER/!BOUND pattern. The Pellet Integrity Constraint Validator[22] implements this approach, automatically translating OWL axioms into ASK SPARQL queries, so an OWL ontology can be used to validate RDF data integrity. The previous domain restriction of the `foaf:name` property may be written as the following integrity constraint:

```
ASK WHERE {
  ?person  foaf:name  ?name .
  OPTIONAL { ?person2  rdf:type foaf:Person }
  FILTER ( bound(?person2) )
  FILTER ( ?person = ?person2 )
}
```

3.4 Target languages

The success of the project strongly depends largely on the achieved impact. And the impact is directly linked to the programming language chosen, both for its popularity and its technical features. With respect to the popularity of programming languages the TIOBE Programming Community index gives an indication of it[23]. On the top of this ranking is the Java programming language (Gosling et al., 2005) since 2001. Java is general-purpose, concurrent, and class-based object-oriented programming language. It has a simpler object model and fewer low-level facilities, such as direct access to memory. Java has single inheritance, but conceptually that is not entirely true since the inclusion of Interfaces (reference types without implementation). Java applications are compiled to an intermediate language (known as"bytecode") that runs on a virtual machine. The `java.lang.reflect` package provides introspection capabilities that allows to examine Java applications at run-time; for providing some computational reflection capabilities the package simulates them with an implementation of the Proxy pattern (Gamma et al., 1994), known as "Dynamic Proxy Classes". Additionally to all these features of the language, Java would a good candidate for implementing Trioo because all the good Semantic Web tools implemented in Java: Jena, OWL API, Pellet, etc.

However dynamic programming languages are closer to RDF, because actually they implement a prototype-based model. Probably the most pure and complete of these languages could be Self (Ungar and Smith, 1987). Self is a prototype-based dynamic object-oriented programming language, environment,

[21] `http://www.ivan-herman.net/Misc/PythonStuff/RDFSClosure/`

[22] `http://clarkparsia.com/pellet/icv`

[23] Available online at `http://www.tiobe.com/index.php/tiobe_index`, retrieved by February of 2010.

and virtual machine centred around the principles of simplicity, uniformity, concreteness, and liveness. Unfortunately it is less usable than other modern languages. Due to this fact, and also taking into account popularity criteria, Python would be one of the most interesting choices for provide the second reference implementation of Trioo. Python is a prototype-based general-purpose and dynamic typed programming language (van Rossum, 1989), designed with the philosophy of emphasise the code readability. It offers a modern and versatile computational model, supporting several paradigms such as object-oriented and structured programming. Python also includes some important features, such as introspection, structural reflexion (the union of the latter two is called metaprogramming), multiple inheritance and generative programming, converting its computational model in one of the most usable and versatile between modern programming languages.

But, apart from the commitment of initially apply Trioo on these two widely used object-oriented programming languages, Trioo aims to offer much more, covering other scenarios that would be very interesting to complete this research work. For instance, the powerful dynamic features available on JavaScript offer an interesting scenario where more conclusions could be extracted. For instance, given the trend to use AJAX (Asynchronous JavaScript and XML) for improving the user experience on Web interfaces, together with the W3C's effort to enrich the markup with technologies such as RDFa (Adida et al., 2008), it could convert Trioo in a suitable technology to provide support for developing semantically-enriched user interfaces on (X)HTML due the interesting dynamic features of the JavaScript language. Additionally, the support for dynamic typing included in C# 4.0 (Hejlsberg, 2008) open an interesting area of applicability in the Microsoft's .NET framework. These, and others, lines open amazing opportunities on unexplored research fields in many of the potential target languages where Trioo could be applied and/or extended.

3.5 Mechanism

How Trioo would be applied on all target languages is a quite important question that may not be answered yet for all of them. Although probably such mechanism would need to be supplemented with a external configuration file (XML, YAML or whatever suitable format). A priori there are some requirements that should be taken into account: simplicity (it should follow the DRY philosophy, reducing as possible the repetition of information), non intrusive (the mechanism used should not modify the original design, e.g. inheritance would not be a suitable construction when the language only supports single inheritance), legible (it should not dirty the readability of source code), accessible at run-time (due to the intrinsically dynamic features of the project, the mechanism chosen should offer that meta-information at run-time), customisable (such mechanism should support to be parametrise, in order to allow implementations to be customisable when would be needed). All these requirements could be covered with annotations, that are supported by the two first chosen target languages. Annotations were introduced in the version 5.0 of the Java programming language (Bracha, 2004). They are a special form of syntactic metadata that can be added to Java source code and can be reflectively retrieved at run-time. On the contrary, annotations in Python are syntactic sugar that actually provides a generic implementation of the decorator pattern (Gamma et al., 1994), enhancing the action of the function or method they decorate. In Python prior to version 3, decorators only apply to functions and methods; however class decorators are supported from Python 3.0 (Winter, 2010).

3.6 Persistence

It is obvious that, in this concrete moment when W3C is running a working group[24] to extend SPARQL technology, including some of the features that the community has identified as both desirable and important for interoperability based on experience with the initial version of the standard (Kjernsmo and Passant, 2009), formally known as SPARQL 1.1. For instance, the new properties path (Seaborne, 2010) converts SPARQL in a more powerful query language. Henceforth all these new features should be exploited by new tools.

A SPARQL query contains a set of triple patterns, where triple patterns are like RDF triples except that each term may be a variable. A basic graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables and the result is RDF graph equivalent to the subgraph. The expressiveness of SPARQL is powerful: it is a query language equivalent in expressive power to Relational Algebra (Angles and Gutierrez, 2008). Therefore, in a similar way that was previously done for relational databases (Cyganiak, 2005), it would be necessary to additionally describe a transformation from object-oriented programming languages into the algebra of SPARQL.

[24] http://www.w3.org/2009/05/sparql-phase-II-charter

Consequently the innovative approach of the Trioo project would be to provide a purely standards-based implementation for store RDF data. So that means only using SPARQL 1.1, not implement proprietary interfaces, allowing developers to be independent from a concrete RDF store. Obviously, this purist approach would be supplemented with the additional possibility of using specific dialects of SPARQL[25], in a similar way as Hibernate does, in order to maximise the performance where would be possible, but at the same time without favouring the vendor lock-in allowing developers to switch to other RDF store just changing a configuration line.

4 IMPLEMENTATIONS

Currently[26] the Trioo project is immersed in developing the two reference implementations in the two programming languages selected above (see Section 3.4). The analysis carried out in this paper establishes a solid foundation that greatly facilitated the design and implementation on both programming languages. An alpha prototype has been developed in Java with a functional version of some of the core features. The philosophy followed for the development of the Python version is far more ambitious, since it aims to provide RDF support in a more *natural* way, trying to exploit the full power of the prototype-based computational model. Therefore it is still too early to assess the results. In a nutshell, unfortunately both are far from being full functional implementations according the criteria described in this paper, and they still would require more effort on development. Of course, needless to say that when they would be ready for a real usage, all these implementations will be released as open source.

5 CONCLUSIONS AND FUTURE WORK

This paper has described the ongoing work carried out by the Trioo project to leverage the power of the RDF data model into object-oriented programming languages, trying to keep the semantics of data safe and sound into the applications. But the final aim is not only to persist data on RDF stores, but also to allow consuming and publishing RDF data linked on the Web.

[25]SPARQL proprietary extensions are widely implemented by many vendors.

[26]At March 5th, 2010

The analysis accomplished on Section 3 shows that it is possible to integrate RDF data model into object-oriented computational models with a good enough level of commitment between both semantics. The prototype-based model fits best with RDF than the class-based, but that should not be an impediment to also implement Trioo on class-based programming languages. And since the approach followed to perform this analysis has been completely language independent, Trioo could be potentially applied to any object-oriented programming language.

Trioo gives rise to the necessity to have software tools for accessing RDF data from software applications. The learning curve of some of the current RDF tools is not always accessible to all developers, while they require advanced knowledge of both the data model and query language. And, as with other technologies, developers should not need to be aware of all the details about how their applications store the data. Therefore Trioo should abstract developers of such details. For the moment the results of the two current implementations are still promising. As these implementations progress to a more mature level of development, many of the current open questions will have a more clear answer. The lessons learned from these implementations will serve as feedback for the current groups working on the related standards involved in Trioo, in such a way that all these standard technologies can really serve to fulfil the objectives set forth in the project initially.

REFERENCES

Adida, B., Birbeck, M., McCarron, S., and Pemberton, S. (2008). RDFa in XHTML: Syntax and Processing. Recommendation, W3C.

Angles, R. and Gutierrez, C. (2008). The Expressive Power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference (ISWC2008)*.

Bauer, C. and King, G. (2004). *Hibernate in Action*. Manning Publications.

Berners-Lee, T. (2006). Linked Data: design issues.

Berners-Lee, T., Fielding, R., and Masinter, L. (2005). RFC3886: Uniform Resource Identifier (URI): Generic Syntax. Request For Comments, The Internet Society.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web: a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*.

Blaschek, G. (1994). *Object-Oriented Programming with Prototypes*. Springer-Verlag New York, Inc.

Booch, G. (1993). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.

Borning, A. H. (1986). Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 30–40.

Bracha, G. (2004). JSR 175: A Metadata Facility for the Java Programming Language. Java Specification Request, Sun Microsystems.

Brickley, D., Guha, R., and McBride, B. (2004). RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation, W3C.

Broekstra, J. and Kampman, A. (2003). SeRQL: a second generation RDF query language. In *Proceedings of SWAD-Europe Workshop on Semantic Web Storage and Retrieval*.

Cyganiak, R. (2005). A relational algebra for SPARQL.

DeMichiel, L. (2009). JSR 317: Java Persistence API, Version 2.0. Java Specification Request, Sun Microsystems.

DeMichiel, L. and Keith, M. (2006). JSR 220: Enterprise JavaBeans 3.0. Java Specification Request, Sun Microsystems.

Evins, M. (1994). Objects Without Classes. *Computer IEEE*, Volume 27:104–109.

Fensel, D., Decker, S., Erdmann, M., and Studer, R. (1998). Ontobroker: The Very High Idea. In *Proceedings of the 11th International Florida Artificial Intelligence Research Symposium (FLAIRS98)*.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). RFC2616: Hypertext Transfer Protocol HTTP/1.1. Technical report, The Internet Society.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional.

Hayes, P. and McBride, B. (2004). RDF Semantics. Recommendation, W3C.

Hejlsberg, A. (2008). The Future of C#. In *the Microsoft's Professional Developers Conference (PDC)*.

Kjernsmo, K. and Passant, A. (2009). SPARQL New Features and Rationale. Working Draft, W3C.

Klyne, G. and Carroll, J. J. (2004). Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, W3C.

Logozzo, F. (2004). Separate Compositional Analysis of Class-based Object-oriented Languages. In *Proceedings of the 10th International Conference Algebraic Methodology and Software Technology (AMAST 2004)*.

Meijer, E., Beckman, B., and Bierman, G. (2006). LINQ: reconciling object, relations and XML in the .NET framework. In *ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois.

Motik, B. and Rosati, R. (2007). A Faithful Integration of Description Logics with Logic Programming. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 477–482.

Noble, J., Taivalsaari, A., and Moore, I. (1999). *Prototype-Based Programming: Concepts, Languages and Applications*. Springer Publishing Company.

Oren, E., Heitmann, B., and Decker, S. (2008). ActiveRDF: embedding Semantic Web data into object-oriented languages. *Journal of Web Semantics*, 6:191–202.

Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF. Recommendation, W3C.

Sauermann, L. and Cyganiak, R. (2008). Cool URIs for the Semantic Web. Interest Group Note, W3C.

Schenk, S. and Gearon, P. (2010). SPARQL 1.1 Update. Working Draft, W3C.

Schneider, M., Carroll, J., Herman, I., and Patel-Schneider, P. F. (2009). OWL 2 Web Ontology Language, RDF-Based Semantics. Recommendation, W3C.

Seaborne, A. (2010). SPARQL 1.1 Property Paths. Working Draft, W3C.

Shinavier, J. (2007). Ripple: Functional Programs as Linked Data. In *Proceedings of the 3rd international workshop on Scripting for the Semantic Web (SFSW2007), co-located with the 4th European Semantic Web Conference (ESWC2007)*.

Sintek, M. and Decker, S. (2002). TRIPLE—A Query, Inference and Transformation Language for the Semantic Web. In *Proceedings of the 1st International Conference on Semantic Web*, volume Volume 2342/2002, pages 364–378, Sardinia, Italy. Springer Berlin / Heidelberg.

Sirin, E. and Parsia, B. (2007). SPARQL-DL: SPARQL Query for OWL-DL. In *Proceeddings of the 3rd OWL Experiences and Directions Workshop (OWLED 2007)*.

Sirin, E. and Tao, J. (2009). Towards Integrity Constraints in OWL. In *Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED 2009)*.

ter Horst, H. J. (2005). Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, Vol. 3:pp. 79–115.

Ungar, D., Chambers, C., Chang, B.-W., and Hlzle, U. (1991). Organizing programs without classes. *LISP and Symbolic Computation*, Volume 4:223–242.

Ungar, D. and Smith, R. B. (1987). SELF: The Power of Simplicity. In *Proceedings of The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, pages 227–241.

van Rossum, G. (1989). The Python programming language. http://www.python.org/.

Winter, C. (2010). PEP 3129 – Class Decorators. Python Enhancement Proposal, Python Software Foundation.

B REFERENCIAS

- ActiveRDF <http://www.activerdf.org/>
- ActiveRecord Castle Project <http://www.castleproject.org/activerecord>
- Amazon SimpleDB <http://aws.amazon.com/simpledb/>
- Amazon's Dynamo - All Things Distributed <http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html>
- Apache CouchDB: The CouchDB Project <http://couchdb.apache.org/>
- Apache Xindice <http://xml.apache.org/xindice/>
- ARQ - Property Paths <http://jena.sourceforge.net/ARQ/property_paths.html>
- ARQ - SPARQL/Update <http://jena.sourceforge.net/ARQ/update.html>
- Artificial by Free Css Templates <http://ibatis.apache.org/>
- asmx-sameas4j - Project Hosting on Google Code <http://code.google.com/p/asmx-sameas4j/>
- Berkeley DB Backend Storage Engine for DURUS <http://www.jcea.es/programacion/durus-berkeleydbstorage.htm>
- bigdata <http://www.systap.com/bigdata.htm>
- Category:Features - SPARQL Working Group <http://www.w3.org/2009/sparql/wiki/Category:Features>
- Clark <http://clarkparsia.com/>
- clarkparsia's Empire at master - GitHub <http://github.com/clarkparsia/Empire>
- Code Generation Library - Code Generation Library <http://cglib.sourceforge.net/>
- Creative Commons <http://creativecommons.org/licenses/by/3.0/es/>
- datagraph's spira at master - GitHub <http://github.com/datagraph/spira>
- DataNucleus Access Platform - DataNucleus Access Platform <http://www.datanucleus.org/products/accessplatform/>

- DataTables - moriarty - Moriarty DataTables - Project Hosting on Google Code <http://code.google.com/p/moriarty/wiki/DataTables>
- Django | Django documentation | Django documentation <http://docs.djangoproject.com/>
- Django | Model instance reference | Django documentation <http://docs.djangoproject.com/en/dev/ref/models/instances/>
- Durus: a Software Package from MNX <http://www.mems-exchange.org/software/durus/>
- eXist-db Open Source Native XML Database <http://exist.sourceforge.net/>
- Feature:PropertyPaths - SPARQL Working Group <http://www.w3.org/2009/sparql/wiki/Feature:PropertyPaths>
- Franz Inc. - Semantic Web Technologies <http://www.franz.com/agraph/>
- Google Accounts <http://appengine.google.com/>
- Grails - GORM <http://www.grails.org/GORM>
- Grails - The search is over. <http://www.grails.org/>
- Guido's Personal Home Page <http://www.python.org/~guido/>
- Hercules - A simple JavaScript library for O/Rdf mapping <http://www.arielworks.net/works/codeyard/hercules>
- hgweb: log <http://bitbucket.org/exogen/telescope>
- Hibernate - JBoss Community <http://www.hibernate.org/>
- http://basex.org/ <http://basex.org/>
- Hypertable: An Open Source, High Performance, Scalable Database <http://hypertable.org/>
- ICSOFT 2010 5th International Conference on Software and Data Technologies <http://www.icsoft.org/>
- InfoQ: Object Oriented Programming: The Wrong Path? <http://www.infoq.com/news/2010/07/objects-smalltalk-erlang>
- InfoQ: Ralph Johnson, Joe Armstrong on the State of OOP <http://www.infoq.com/interviews/johnson-armstrong-oop>
- Java Data Objects (JDO) - Home <http://db.apache.org/jdo/>
- java.com: Java + You <http://www.java.com/>
- Jena Semantic Web Framework <http://openjena.org/>
- jenabean - Project Hosting on Google Code <http://jenabean.googlecode.com/>

- JSave - Protege Wiki <http://protegewiki.stanford.edu/index.php/JSave>
- jtrioo - Project Hosting on Google Code <http://jtrioo.googlecode.com/>
- linqtordf - Project Hosting on Google Code <http://linqtordf.googlecode.com/>
- MarkLogic - MarkLogic Server <http://www.marklogic.com/product/marklogic-server.html>
- MemcacheDB: A distributed key-value storage system designed for persistent <http://memcachedb.org/>
- MongoDB <http://www.mongodb.org/>
- moriarty - Project Hosting on Google Code <http://moriarty.googlecode.com/>
- namespace lookup for RDF developers | prefix.cc <http://prefix.cc/>
- neo4j open source nosql graph database <http://www.neo4j.org/>
- NOSQL Databases <http://nosql-database.org/>
- Oort : Home <http://oort.to/>
- OpenLink Software <http://www.openlinksw.com/>
- OWL API <http://owlapi.sourceforge.net/>
- owl:sameAs <http://sameas.org/>
- Pellet Integrity Constraint Validator <http://clarkparsia.com/pellet/icv>
- Pellet: The Open Source OWL 2 Reasoner <http://clarkparsia.com/pellet/>
- penry - Project Hosting on Google Code <http://code.google.com/p/penry/>
- Platform API - n <http://n2.talis.com/wiki/Platform_API>
- Porting code to Python 3 with 2to3 - Dive Into Python 3 <http://diveintopython3.org/porting-code-to-python-3-with-2to3.html>
- Properties (C# Programming Guide) <http://msdn.microsoft.com/en-us/library/x9fsa0sw(28VS.80).aspx>
- pryoo - Project Hosting on Google Code <http://pryoo.googlecode.com/>
- Python Package Index : dobbin 0.2 <http://pypi.python.org/pypi/dobbin>

- Python Package Index : RDFobject 0.4.2 <http://pypi.python.org/pypi/RDFobject>
- Python Programming Language – Official Website <http://www.python.org/>
- RDF Next Steps Workshop: Final Straw Poll <http://www.w3.org/2010/06/rdf-work-items/table>
- RDF.rb: Linked Data for Ruby <http://rdf.rubyforge.org/>
- RDF2Go - semanticweb.org <http://rdf2go.semweb4j.org/>
- RDFLib <http://www.rdflib.net/>
- RdfPath - ESW Wiki <http://esw.w3.org/RdfPath>
- RDFReactor - semanticweb.org <http://rdfreactor.semweb4j.org/>
- RDFS Closure <http://www.ivan-herman.net/Misc/PythonStuff/RDFSClosure/>
- Redirecting to Semweb4j... <http://semweb4j.org/>
- Redland RDF Library - Ruby Interface <http://librdf.org/docs/ruby.html>
- ripple - Project Hosting on Google Code <http://ripple.googlecode.com/>
- Ruby on Rails <http://rubyonrails.org/>
- RubyForge: ActiveRecord: Project Info <http://rubyforge.org/projects/activerecord/>
- Serialising Java Objects to RDF with Jersey <http://blogs.sun.com/bblfish/entry/serialising_java_objects_to_rdf>
- SPARQL Working Group <http://www.w3.org/2001/sw/DataAccess/>
- SPARQL Working Group Charter <http://www.w3.org/2009/05/sparql-phase-II-charter>
- Spira: A Linked Data ORM for Ruby - The Datagraph Blog <http://blog.datagraph.org/2010/05/spira>
- SQLAlchemy - The Database Toolkit for Python <http://www.sqlalchemy.org/>
- surfrdf - Project Hosting on Google Code <http://surfrdf.googlecode.com/>
- Talis Platform - Home <http://www.talis.com/platform/>
- Tamino | The XML Database <http://www.softwareag.com/Corporate/products/wm/tamino/default.asp>
- TeRRAS <http://terras.sourceforge.net/>

- The Apache Cassandra Project <http://incubator.apache.org/cassandra/>
- Timeline of programming languages - Wikipedia, the free encyclopedia <http://en.wikipedia.org/wiki/Timeline_of_programming_languages>
- TIOBE Software: The Coding Standards Company <http://www.tiobe.com/index.php/tiobe_index>
- trioo, keeping the semantics of rdf data safe and sound into object-oriented software <http://trioo.wikier.org/>
- W3C RDB2RDF Incubator Group <http://www.w3.org/2005/Incubator/rdb2rdf/>
- W3C Workshop <http://www.w3.org/2009/12/rdf-ws/>
- Welcome to Apache Hadoop! <http://hadoop.apache.org/>
- What's new in Python 2.7 <http://docs.python.org/dev/whatsnew/2.7.html>
- What's new in Python 3.0 <http://docs.python.org/dev/3.0/whatsnew/3.0.html>
- World Wide Web Consortium (W3C) <http://www.w3.org/>
- ZODB - a native object database for Python <http://www.zodb.org/>
- Zorba: The XQuery Processor <http://www.zorba-xquery.com/>

BIBLIOGRAFÍA

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996. (Citado en las páginas 33, 34 y 35.)

[2] Ben Adida, Mark Birbeck, Shane McCarron, and Steven Pemberton. RDFa in XHTML: Syntax and Processing. Recommendation, W3C, October 2008. (Citado en las páginas 46 y 51.)

[3] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. (Citado en la página 7.)

[4] Franz Baader. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003. (Citado en la página 38.)

[5] Cosmin Basca. SuRF, Tapping into the Web of Data. In *The European Python Conference (EuroPython 2009)*, 2009. (Citado en la página 27.)

[6] Christian Bauer and Gavin King. *Hibernate in Action*. Manning Publications, 2004. (Citado en la página 11.)

[7] Dave Beckett. RDF/XML Syntax Specification (Revised). Recommendation, W3C, February 2004. (Citado en la página 38.)

[8] Oren Ben-Kiki and Clark Evans Ingy dot Net. YAML Ain't Markup Language (YAML) version 1.2 3rd edition. Technical report, 2009. (Citado en la página 48.)

[9] T. Berners-Lee, R. Fielding, and L. Masinter. RFC3886: Uniform Resource Identifier (URI): Generic Syntax. Request For Comments, The Internet Society, January 2005. (Citado en las páginas 3, 36 y 39.)

[10] Tim Berners-Lee. Cool URIs don't change. http://www.w3.org/Provider/Style/URI.html, 1998. (Citado en la página 39.)

[11] Tim Berners-Lee. Linked Data: design issues, July 2006. (Citado en las páginas 3, 24 y 46.)

[12] Tim Berners-Lee. Read-Write Linked Data: design issues, June 2010. (Citado en la página 46.)

[13] Tim Berners-Lee and Dan Connolly. Delta: an ontology for the distribution of differences between RDF graphs. http://www.w3.org/DesignIssues/Diff, 2001. (Citado en la página 44.)

[14] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. Team Submission, W3C, January 2008. (Citado en la página 41.)

[15] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web: a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, May 2001. (Citado en la página 2.)

[16] David Binge. Durus: A Persistence System for Python. In *Washington International Python Conference (PyCon 2005)*, 2005. (Citado en la página 15.)

[17] Mark Birbeck and Shane McCarron. CURIE Syntax 1.0: A syntax for expressing Compact URIs. Candidate Recommendation, W3C, 2009. (Citado en la página 47.)

[18] Guenther Blaschek. *Object-Oriented Programming with Prototypes*. Springer-Verlag New York, Inc., 1994. (Citado en la página 35.)

[19] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Recommendation, W3C, 2007. (Citado en la página 18.)

[20] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1993. (Citado en la página 34.)

[21] A. H. Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 30–40, 1986. (Citado en la página 35.)

[22] Gilad Bracha. JSR 175: A Metadata Facility for the Java Programming Language. Java Specification Request, Sun Microsystems, 2004. (Citado en la página 54.)

[23] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Recommendation, W3C, November 2008. (Citado en las páginas 3, 9 y 17.)

[24] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC 2000)*, 2000. (Citado en la página 17.)

[25] Dan Brickley, R.V. Guha, and Brian McBride. RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation, W3C, 10 February 2004. (Citado en las páginas 3 y 38.)

[26] Dan Brickley and Libby Miller. Foaf vocabulary. Namespace: http://xmlns.com/foaf/0.1/, January 2010. (Citado en la página 47.)

[27] Jeen Broekstra and Arjohn Kampman. SeRQL: a second generation RDF query language. In *Proceedings of SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, 2003. (Citado en la página 19.)

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, November 2006. (Citado en la página 17.)

[29] Surajit Chaudhuri and Moshe Y. Vardiy. On the equivalence of recursive and nonrecursive datalog programs. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1997. (Citado en la página 43.)

[30] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. (Citado en la página 9.)

[31] Kin-Man Chung, Pierre Delisle, and Mark Roth. JSR 245: JavaServer Pages 2.1, expression language specification. Java Specification Request, Sun Microsystems, May 2006. (Citado en la página 41.)

[32] James Clark and Steve DeRose. XML Path Language (XPath). Recommendation, W3C, 1999. (Citado en la página 41.)

[33] KL Clark. Negation as Failure. In *Logic and Databases*, 1978. (Citado en la página 43.)

[34] Richard Cyganiak. A relational algebra for SPARQL. Technical report, Hewlett-Packard, 2005. (Citado en la página 46.)

[35] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, 1966. (Citado en la página 33.)

[36] Linda DeMichiel. JSR 317: Java Persistence API, Version 2.0. Java Specification Request, Sun Microsystems, December 2009. (Citado en las páginas 12 y 29.)

[37] Linda DeMichiel and Michael Keith. JSR 220: Enterprise JavaBeans 3.0. Java Specification Request, Sun Microsystems, May 2006. (Citado en las páginas 12 y 29.)

[38] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. (Citado en la página 41.)

[39] M. Evins. Objects Without Classes. *Computer IEEE*, Volume 27:104–109, 1994. (Citado en la página 34.)

[40] Dieter Fensel, Stefan Decker, Michael Erdmann, and Rudi Studer. Ontobroker: The Very High Idea. In *Proceedings of the 11th International Florida Artificial Intelligence Research Symposium (FLAIRS98)*, 1998. (Citado en la página 21.)

[41] Sergio Fernández, Diego Berrueta, Miguel García Rodríguez, and Jose E. Labra. TRIOO, Keeping the Semantics of Data Safe and Sound into Object-Oriented Software. In *Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFT 2010)*, Athens, Greece, July 2010. (Citado en la página 65.)

[42] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext Transfer Protocol HTTP/1.1. Technical report, The Internet Society, June 1999. (Citado en las páginas 3 y 46.)

[43] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 2006. (Citado en la página 51.)

[44] Brian Foote. Objects, Reflection, and Open Languages. In *Proceedings of the Workshop on Object-Oriented Reflection and Metalevel Architectures (ECOOP'92)*, 1992. (Citado en la página 56.)

[45] International Organization for Standardization. ISO/IEC 9075-1:2008 SQL/Framework, 2008. (Citado en la página 9.)

[46] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002. (Citado en las páginas 7, 8 y 19.)

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. (Citado en las páginas 56, 57 y 58.)

[48] Jose Emilio Labra Gayo. *Desarrollo Modular de Procesadores de Lenguajes a partir de Especificaciones Semánticas Reutilizables*. PhD thesis, University of Oviedo, June 2001. (Citado en la página 34.)

[49] Y. Goland, E. Whitehead, A. Faizi, and D. Jensen. HTTP Extensions for Distributed Authoring: WebDAV, 1999. (Citado en la página 45.)

[50] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. (Citado en la página 33.)

[51] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, 3rd edition*. Addison-Wesley Professional, 2005. (Citado en la página 50.)

[52] Marc Hadley and Paul Sandoz. JSR 311: JAX-RS, The Java APi for RESTful Web Services. Java Specification Request, Sun Microsystems, 2009. (Citado en la página 27.)

[53] Patrick Hayes and Brian McBride. RDF Semantics. Recommendation, W3C, February 2004. (Citado en la página 37.)

[54] Ullrich Hustadt. Do we need the closed-world assumption in knowledge representation. In Franz Baader, Martin Buchheit, Manfred A. Jeusfeld, and Werner Nutt, editors, *Working Notes of the KI'94 Workshop: Reasoning about Structured Objects: Knowledge Representation Meets Databases (KRDB'94)*, volume D-94-11, page 24–26, 1994. (Citado en la página 43.)

[55] ECMA International. ECMA-262: ECMAScript Language Specification. Standard, ECMA International, 1999. (Citado en la página 51.)

[56] ECMA International. ECMA-334: C# Language Specification. Standard, ECMA International, 2006. (Citado en la página 51.)

[57] Kohsuke Kawaguchi. JSR 222: Java Architecture for XML Binding (JAXB) 2.0. Java Specification Request, Sun Microsystems, 2006. (Citado en la página 17.)

[58] Michael Kifer and Georg Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. *SIGMOD Rec.*, 18(2):134–146, 1989. (Citado en la página 21.)

[59] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, W3C, 10 February 2004. (Citado en las páginas 3, 9, 36 y 57.)

[60] Krys J. Kochut and Maciej Janik. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In *Proceedings of the 4th European Semantic Web Conference (ESWC2007)*, 2007. (Citado en la página 41.)

[61] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007. (Citado en la página 11.)

[62] J. Kunze, M. Haye, E. Hetzner, M. Reyes, and C. Snavely. Pairtrees for Object Storage. Technical report, California Digital Library, 2008. (Citado en la página 24.)

[63] Carl Lagoze, Sandy Payette, Edwin Shin, and Chris Wilper. Fedora: an architecture for complex objects and their relationships. *International Journal on Digital Libraries*, 6:124–138, 2006. (Citado en la página 24.)

[64] Francesco Logozzo. Separate Compositional Analysis of Class-based Object-oriented Languages. In *Proceedings of the 10th International Conference Algebraic Methodology and Software Technology (AMAST 2004)*, 2004. (Citado en la página 34.)

[65] Pattie Maes. Computational reflection. *The Knowledge Engineering Review*, 3(01):1–19, 1988. (Citado en la página 55.)

[66] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 6–6, New York, NY, USA, 2009. ACM. (Citado en la página 17.)

[67] Frank Manola and Eric Miller. RDF Primer. Recommendation, W3C, February 2004. (Citado en la página 36.)

[68] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, June 26-29 2006. (Citado en la página 29.)

[69] Sun Microsystems. Java Remote Method Invocation Specification (RMI). Technical report, Sun Microsystems, 1997. (Citado en la página 55.)

[70] Sun Microsystems. Dynamic Proxy Classes. Technical report, Sun Microsystems, 1999. (Citado en la página 56.)

[71] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978. (Citado en la página 35.)

[72] Iván Minguez, Diego Berrueta, and Luis Polo. *Cases on Semantic Interoperability for Information Systems Integration: Practices and Applications*, chapter CRUZAR: an application of semantic matchmaking to eTourism, pages 255–271. IGI Global, 2009. (Citado en la página 48.)

[73] Boris Motik and Riccardo Rosati. A Faithful Integration of Description Logics with Logic Programming. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 477–482, 2007. (Citado en la página 43.)

[74] James Noble, Antero Taivalsaari, and Ivan Moore. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer Publishing Company, 1999. (Citado en la página 35.)

[75] E. Oren, B. Heitmann, and S. Decker. ActiveRDF: embedding Semantic Web data into object-oriented languages. *Journal of Web Semantics*, 6:191–202, 2008. (Citado en la página 19.)

[76] Maxim Orgiyan and Marc Van Cappellen. JSR 225: XQuery API for Java (XQJ). Java Specification Request, Sun Microsystems, 2009. (Citado en la página 18.)

[77] Francisco Ortín. *Sistema computacional de programación flexible diseñado sobre una máquina abstracta reflexiva no restrictiva*. PhD thesis, University de Oviedo, 2001. (Citado en la página 56.)

[78] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language: Semantics and Abstract Syntax. Recommendation, W3C, 2004. (Citado en la página 42.)

[79] Peter F. Patel-Schneider and Ian Horrocks. A Comparison of Two Modelling Paradigms in the Semantic Web. In *Proceedings of The Fifteenth International World Wide Web Conference (WWW2006)*, Edinburgh, Scotland, May 2006. ACM Press. (Citado en la página 43.)

[80] Benjamin Peterson. PEP 373 – Python 2.7 Release Schedule. Technical report, Python Software Foundation, 2008. (Citado en la página 59.)

[81] Axel Polleres. From SPARQL to rules (and back). In Peter F. Patel-Schneider and Prashant Shenoy, editors, *Proceedings of the 16th World Wide Web Conference (WWW2007)*, pages 787–796, Banff, Canada, May 2007. ACM Press. (Citado en la página 43.)

[82] Axel Polleres, Aidan Hogan, Andreas Harth, and Stefan Decker. Can we ever catch up with the Web? *Semantic Web Journal*, 2010. (Citado en la página 3.)

[83] Dan Pritchett. BASE: An ACID Alternative. *ACM Queue Magazine*, 6(3):48–55, 2008. (Citado en la página 17.)

[84] Eric Prud'hommeaux. SPARQL 1.1 Federation Extensions. Working Draft, W3C, 2010. (Citado en la página 46.)

[85] Eric Prud'hommeaux and Michael Hausenblas. Use Cases and Requirements for Mapping Relational Databases to RDF. Working Draft, W3C, 8 June 2010. (Citado en la página 4.)

[86] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Recommendation, W3C, January 2008. (Citado en las páginas 3, 9, 41 y 45.)

[87] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A Navigational Language for RDF. In *Proceddings of the 7th International Semantic Web Conference (ISWC2008)*, 2008. (Citado en las páginas 41 y 42.)

[88] Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources with SPARQL. In *Proceedings of the 5th European Semantic Web Conference (ESWC2008)*, 2008. (Citado en la página 46.)

[89] Andreas Reuter and Theo Haerder. Principles of Transaction-Oriented database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983. (Citado en la página 17.)

[90] John F Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 27(7):383–393, 1995. (Citado en la página 44.)

[91] John Rose. JSR 292: Supporting Dynamically Typed Languages on the Java Platform. Java Specification Request (in progress), Sun Microsystems, August 2008. (Citado en la página 11.)

[92] Craig Russell. JSR 012: Java Data Objects (JDO) Specification 1.0. Java Specification Request, Sun Microsystems, May 2001. (Citado en las páginas 14 y 29.)

[93] Craig Russell. JSR 243: Java Data Objects 2.0, an extension to the JDO specification. Java Specification Request, Sun Microsystems, February 2006. (Citado en las páginas 14 y 29.)

[94] Leo Sauermann and Richard Cyganiak. Cool URIs for the Semantic Web. Interest Group Note, W3C, December 2008. (Citado en la página 3.)

[95] Bernhard Schandl. Replication and Versioning of Partial RDF Graphs. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC2010)*, 2010. (Citado en la página 44.)

[96] Simon Schenk and Paul Gearon. SPARQL 1.1 Update. Working Draft, W3C, 26 January 2010. (Citado en las páginas 3 y 46.)

[97] Michael Schneider, Jeremy Carroll, Ivan Herman, and Peter F. Patel-Schneider. OWL 2 Web Ontology Language, RDF-Based Semantics. Recommendation, W3C, October 2009. (Citado en la página 3.)

[98] Andy Seaborne. SPARQL 1.1 Property Paths. Working Draft, W3C, January 2010. (Citado en las páginas 3 y 42.)

[99] Andy Seaborne, Geetha Manjunath, Chris Bizer, John Breslin, Souripriya Das, Ian Davis, Steve Harris, Kingsley Idehen, Olivier Corby, Kjetil Kjernsmo, and Benjamin Nowack. SPARQL Update: a language for updating RDF graphs. Member Submission, W3C, July 2008. (Citado en la página 57.)

[100] Joshua Shinavier. Ripple: Functional Programs as Linked Data. In *Proceedings of the 3rd international workshop on Scripting for the Semantic Web (SFSW2007), co-located with the 4th European Semantic Web Conference (ESWC2007)*, 2007. (Citado en la página 22.)

[101] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In *Proceeddings of the 3rd OWL Experiences and Directions Workshop (OWLED 2007)*, 2007. (Citado en la página 43.)

[102] Evren Sirin and Jiao Tao. Towards Integrity Constraints in OWL. In *Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED 2009)*, 2009. (Citado en la página 43.)

[103] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, Vol. 3:pp. 79–115, 2005. (Citado en las páginas 42 y 47.)

[104] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *LISP and Symbolic Computation*, Volume 4:223–242, 1991. (Citado en la página 36.)

[105] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *Proceedings of The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, pages 227–241, 1987. (Citado en las páginas 33, 36 y 50.)

[106] Guido van Rossum. The Python programming language. http://www.python.org/, 1989. (Citado en la página 51.)

[107] Max Völkel and Tudor Groza. SemVersion: RDF-based ontology versioning system. In *Proceedings of the IADIS International Conference WWW/Internet 2006 (ICWI 2006)*, 2006. (Citado en la página 44.)

[108] Max Völkel. RDFReactor – From Ontologies to Programatic Data Access. In *Proceedings of the Jena User Conference 2006*. HP Bristol, Mai 2006. (Citado en la página 24.)

[109] Dimitris Zeginis, Yannis Tzitzikas, and Vassilis Christophides. On the Foundations of Computing Deltas Between RDF Models. In K. Aberer et al., editor, *Proceedings of the 6th International and 2nd Asian Semantic Web Conference (ISWC2007/ASWC2007)*, volume 4825, page 637–651. Springer, 2007. (Citado en la página 44.)

[110] Zheng Zhang, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, and Chao Jin. BitVault: a highly reliable distributed data retention platform. *ACM SIGOPS Operating Systems Review*, 41(2):27–36, 2007. (Citado en la página 17.)