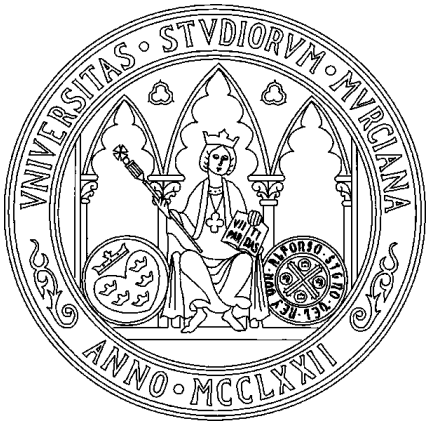
Trabajo Fin de Máster

Visualización de Esquemas y Datos en Bases de Datos NoSQL

Alumno
Alberto Hernández Chillón

Directores
Jesús J. García Molina
Diego Sevilla Ruiz



Julio de 2016

Agradecimientos

Quiero dedicar unas palabras de merecido agradecimiento a una serie de personas cuya colaboración, de forma activa o pasiva, me ha ayudado a llegar a este punto del proyecto. Sin vuestro apoyo, sin vuestros ánimos, habría sido difícil seguir adelante. Esto va por todos vosotros.

A mis padres, Salvador y Pilar, gracias por educarme en los valores que me definen, por inducirme la curiosidad por las cosas, por asegurarme una formación adecuada. Por animarme en cada decisión que he tomado y apoyarme cuando he fallado, haberme dado tanto y haberme pedido tan poco. A mi hermana, Raquel, gracias por estar siempre ahí dándome tu apoyo incondicional en mis momentos de duda. Por lo que me has enseñado y te queda por enseñarme. A mi pareja, Alba, gracias por dar sentido a todo, por tu paciencia y ánimo. Por tu inestimable punto de vista complementario. Por motivarme para darlo todo, porque nuestro proyecto acaba de empezar.

A mis tutores de proyecto, Diego y Jesús, gracias por confiar en mí desde el primer instante. Por prestarme vuestra ayuda, darme consejo, motivarme y ofrecerme vuestro siempre interesante punto de vista. Por demostrarme que sois unos grandes profesionales, unas grandes mentes y unas grandes personas.

A la Cátedra SAES-UMU y en especial a Gregorio, gracias por darme la oportunidad que necesitaba, y que siempre agradeceré, de formarme y ayudarme a crecer como profesional. A Pedro, Fran y Víctor, gracias por vuestras valoraciones, por ofrecerme una vía de escape cuando es necesario, por enseñarme cada día algo nuevo.

Al resto de incontables amigos involucrados. En todo momento he tenido presente vuestras muestras de afecto, y agradezco que hayáis estado conmigo durante todo este tiempo.

A todos los que confiáis en mí, gracias de corazón. Por ser como sois. Porque con vosotros todo merece la pena. Porque lo mejor está por llegar.

Índice general

Declaración firmada sobre originalidad del trabajo

De acuerdo al Reglamento que regula los trabajos fin de máster en la Universidad de Murcia, DECLARO que asumo la originalidad del trabajo presentado y que todas las fuentes utilizadas han sido debidamente citadas.

Murcia, 24 de Junio de 2016
Nombre y firma
Firmado Alberto Hernández Chillón

Resumen

Los sistemas relacionales han evidenciado serias limitaciones cuando se aplican a las nuevas aplicaciones que están surgiendo. Por ejemplo, los requisitos de escalabilidad y el manejo de datos complejos de las aplicaciones Web 2.0 no pueden ser satisfechos por los sistemas relacionales. Por ello, a lo largo de esta década han aparecido los sistemas NoSQL para dar respuesta a los desafíos planteados por las aplicaciones modernas. El término NoSQL se usa para referirse a sistemas que soportan diferentes paradigmas de modelado de datos, entre los que destacan: Clave-valor, basados en documentos, familias de columnas y basados en grafos.

La mayoría de sistemas NoSQL comparten algunas características entre las que destaca la ausencia de un esquema de datos (*schemaless*) como sucede con los datos semiestructurados. Esta propiedad se suele utilizar para expresar la mayor flexibilidad de estos sistemas frente a los sistemas relaciones donde el almacenamiento de datos requiere definir antes un esquema de la base de datos. La naturaleza *schemaless* permite que puedan existir diferentes versiones para una misma entidad o tener campos opcionales. En realidad, el esquema de los datos es siempre necesario ya que debe ser manejado tanto por diseñadores de bases de datos como por los programadores que escriben aplicaciones para manejarlas. Además, el esquema debe ser inferido por algunas herramientas que ofrecen funcionalidad para sistemas NoSQL, como por ejemplo un motor de consultas SQL. Por esta razón, recientemente se han propuesto diferentes enfoques para inferir el esquema de bases de datos NoSQL.

Esta tesis de máster parte del proceso de inferencia definido en un trabajo previo de los tutores en el que se obtienen esquemas NoSQL con versiones de entidades. Dichos esquemas son definidos por un metamodelo que ha sido un elemento clave en el trabajo realizado. En esta tesis se ha abordado la visualización de los esquemas y la visualización de datos para bases de datos NoSQL. Para ello se han construido dos herramientas de visualización. Por un lado, una herramienta capaz de mostrar diferentes diagramas del esquema. Cada uno de estos diagramas organiza la información del esquema de diferente forma para facilitar la comprensión de un determinado aspecto, por ejemplo obtener un esquema global de la base de datos con todas las versiones de entidades y todas las relaciones de agregación y referencia existentes. Por otro lado, una herramienta de visualización de los datos almacenados en la propia base de datos NoSQL clasificados en función de la versión a la que pertenecen.

Para abordar este objetivo se plantea una solución basada en la aplicación de técnicas MDE a bases de datos NoSQL. Con este planteamiento se consigue, por un lado, trabajar con bases de datos NoSQL con independencia de la implementación de las mismas, y por otro lado emplear transformaciones de modelos en modelos y modelos en datos para automatizar las soluciones. Cabe destacar que estas herramientas son de las primeras propuestas de visualización de esquemas de bases de datos NoSQL y de datos NoSQL, y que pueden ser muy útiles para apoyar a los diseñadores y programadores de bases de datos NoSQL.

1 Introducción

1.1 Contexto

En los últimos años ha crecido el interés por los sistemas de gestión de bases de datos NoSQL y continuamente aumenta el número de empresas que confían en ellos. Las modernas aplicaciones surgidas, principalmente, con la Web 2.0 requieren algunas prestaciones que no pueden ser satisfechas por los sistemas relacionales, como son el almacenamiento de grandes volúmenes de datos, debido a que los datos tienen una estructura compleja y se requiere una alta escalabilidad. Para abordar estos requisitos han ido surgiendo más de un centenar de sistemas no relacionales para los que se usa el término *'NoSQL'* para referirse a todos ellos. En realidad estos sistemas son un reflejo de diferentes paradigmas de modelado de datos, entre los que destacan: Clave-valor, basados en documentos, familias de columnas y basados en grafos [40].

La mayoría de sistemas NoSQL comparten algunas características entre las que destaca la ausencia de un esquema de datos como sucede con los datos semiestructurados [40]. Esta propiedad se suele utilizar para expresar la mayor flexibilidad de estos sistemas frente a los sistemas relaciones donde el almacenamiento de datos requiere definir antes un esquema de la base de datos. En realidad, el esquema de los datos es siempre necesario ya que debe ser manejado tanto por diseñadores de bases de datos como por los programadores que escriben aplicaciones para manejarlas. El diseñador debe tener en mente un esquema cuando toma decisiones sobre la estructura del almacenamiento que se ajusta a los requisitos, y los programadores deben tener en mente el esquema implícito en la estructura de los datos y en el código.

El esquema debe ser inferido por algunas herramientas que quieren ofrecer alguna funcionalidad para sistemas NoSQL, como por ejemplo un motor de consultas SQL, como es el caso de Apache Drill. Además, la inferencia del esquema puede ser útil para apoyar a los programadores que escriben código para datos NoSQL al ofrecer utilidades de visualización o generación automática de validadores. Por ello, recientemente se han propuesto algunos enfoques para la inferencia de esquemas a partir de bases de datos NoSQL [37, 20, 48].

Recientemente, un informe de DataDiversity [25] ha subrayado la importancia del modelado para bases de datos NoSQL y señalado tres tipos de utilidades que serán esenciales en un futuro inmediato: Visualización de modelos, bien creados directamente o inferidos de los datos, generación automática de código a partir de esquemas, como son validadores de datos o código requerido por *NoSQL-Object mappers*, y manejo de metadatos en escenarios como la integración de herramientas.

1.2 Objetivos

El objetivo de este trabajo de fin de máster ha sido abordar la visualización de esquemas y datos NoSQL a partir de los esquemas inferidos siguiendo el enfoque descrito en [37]. En concreto, se han diseñado e implementado dos herramientas de visualización para sistemas NoSQL: visualización de esquemas y visualización de propiedades de los datos relacionadas con las versiones existentes de esquemas y datos en la base de datos.

A lo largo de esta década la Ingeniería del Software Dirigida por Modelos (*Model-Driven Software Engineering*, *MDE*) ha emergido como una nueva forma de desarrollo de software que incrementa la productividad por medio de la generación automática de código a partir de modelos del software (código y datos). Las dos herramientas de visualización se han diseñado e implementado como soluciones MDE definiendo cadenas de transformación de modelos. La visualización de esquemas se ha llevado a cabo mediante la definición de un lenguaje de modelado o lenguaje específico del dominio (*Domain Specific Language*, *DSL*) creado con la herramienta *Sirius*. Por otro lado, la visualización de datos se ha realizado utilizando la librería *D3.js* para mostrar los modelos obtenidos de los datos NoSQL.

En la actualidad, y en la medida que sabemos, solo la herramienta *ER/Studio* ofrece una visualización de esquemas NoSQL, y solo para el sistema *MongoDB*. También *CA ERwin* ha anunciado esta funcionalidad en sus productos pero todavía no es soportada [47]. En cualquier caso, estas herramientas no contemplan la existencia de diferentes versiones de una misma entidad, como sí se hace en nuestro trabajo, siendo un aspecto clave del mismo. En cuanto a la visualización de datos, *MongoDB Compass* es una utilidad que ofrece gráficos que muestran propiedades de los datos, pero en nuestro caso la información visualizada tiene que ver con las diferentes versiones que pueden existir de cada entidad y a la estructura de cada una de ellas, y se ha introducido el concepto de versión de un esquema.

1.3 Metodología

Con el fin de alcanzar los objetivos introducidos para este trabajo de fin de máster, se ha seguido una metodología *DSRM* (*Design Science Research Methodology*), descrita

en [32, 45]. El proceso de diseño consiste en seis actividades principales, como puede observarse en la figura 1.1: (i) Identificación del problema y motivación, (ii) definición de objetivos para una solución, (iii) diseño y desarrollo, (iv) demostración, (v) evaluación y (vi) conclusiones y comunicación.

Este es un proceso con etapas iterativas en el que el conocimiento adquirido al construir y evaluar soluciones prototipo o parciales sirve, a su vez, como retroalimentación para implementar la solución final. El proceso comienza identificándose el problema y la motivación. En este capítulo se está tratando este aspecto, y en el mismo se ha puesto de manifiesto la utilidad y el interés por desarrollar una serie de herramientas de visualización de versiones de esquemas y datos para bases de datos NoSQL.

La segunda actividad consiste en desarrollar una solución a llevar a cabo, hacer un estudio del arte para analizar el punto de partida, conocer alternativas existentes y escoger las tecnologías más adecuadas para llevar a cabo esta solución. Para ello se han estudiado una serie de herramientas existentes y se han descrito sus limitaciones. Toda esta información puede verse reflejada en los capítulos 4 y 5.

La tercera actividad consiste en dos tareas bien diferenciadas: El diseño y desarrollo de dos herramientas de visualización de esquemas NoSQL y datos contenidos en una base de datos. El diseño de las herramientas se ha hecho en base a las necesidades de visualización existentes, y el desarrollo de las mismas se ha efectuado de forma iterativa y agregando funcionalidad de manera incremental. Las herramientas desarrolladas en esta actividad son descritas en los capítulos 6 y 7.

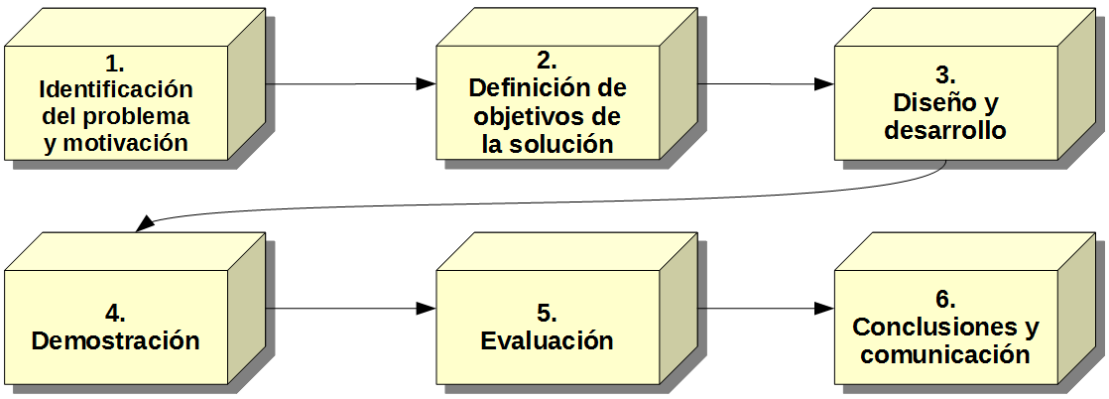


Figura 1.1: Diagrama de la metodología *DSRM* seguida.

Para la cuarta y quinta actividad se han efectuado distintas pruebas y demostraciones de funcionamiento mediante la creación de diversos casos de estudio y bancos de pruebas que aquí se recogen al final del capítulo dedicado a cada herramienta. Para ello se han implementado utilidades para generar casos de prueba a partir de modelos de entrada, y se han efectuado diversas visualizaciones de los resultados.

Finalmente, la última actividad de *conclusiones y comunicación* se ha completado en el capítulo 8. En este capítulo se han identificado diversas líneas de investigación futuras y mejoras prometedoras al proyecto. Un objetivo propuesto tras la finalización de este proyecto de fin de máster consistirá en escribir artículos describiendo el trabajo realizado y publicarlos en revistas y conferencias, con el fin de dar a conocer los resultados obtenidos.

A lo largo del transcurso de este proyecto se han obtenido habilidades y conocimientos relacionados con tecnologías y herramientas de los que se carecía al inicio del mismo, entre los que se pueden destacar:

- Creación de *DSLs* gráficos con Sirius para crear, editar y visualizar modelos.
- Manipulación de modelos mediante la API Java generada del metamodelo *Ecore*.
- Definición de clases mediante *Xtend* utilizado para generar código.
- Conocimientos de programación con la librería de visualización *D3.js*.
- Creación de *plugins* de metamodelos *Ecore* para una posterior distribución.

1.4 Estructura del documento

Este documento se ha organizado en los siguientes capítulos. El presente capítulo 1 ha presentado el contexto, motivación, objetivos y la metodología seguida. Los capítulos 2 y 3 introducen conceptos básicos de MDE tales como el concepto de *MDE*, *metamodelo* y *DSL*, y conceptos de sistemas NoSQL como *schemaless* y *orientación a la agregación* necesarios para comprender el resto del documento. El capítulo 4 presenta el estado del arte del que se parte, trabajos relacionados y aproximaciones a la visualización anteriores, y el capítulo 5 el proceso de inferencia básico para entender de dónde provienen los esquemas con los que se va a trabajar. Los capítulos 6 y 7 presentan las herramientas desarrolladas que, partiendo de estos esquemas presentados, permiten visualizar dichos esquemas o bien los datos contenidos en una base de datos NoSQL, haciendo uso de una serie de tecnologías. Por último el capítulo 8 recoge las conclusiones obtenibles del trabajo realizado, así como una serie de vías futuras con el propósito de ampliar y extender el trabajo realizado.

2 Fundamentos de MDE

2.1 Ingeniería dirigida por modelos (MDE)

La Ingeniería basada en modelos (*Model-Driven Engineering*, *MDE*) es una disciplina dentro de la Ingeniería del Software que se ocupa de la utilización sistemática de modelos de software para mejorar la productividad y otros aspectos de la calidad del software, como el mantenimiento y la interoperabilidad entre sistemas. La visión de la construcción del software que propugna esta nueva disciplina ha mostrado su potencial para dominar la complejidad arbitraria del software al proporcionar un mayor nivel de abstracción y elevar también el nivel de automatización. Estos beneficios no sólo son aplicables en la construcción de nuevas aplicaciones sino en la modernización de aplicaciones *legacy* y en la configuración dinámica de sistemas en tiempo de ejecución [23, 28].

En este trabajo se han utilizado técnicas MDE en la creación de herramientas de visualización de esquemas y datos NoSQL. Además, se ha partido de un proceso de inferencia de esquemas basado en modelos y que será explicado en la sección 5.

Existen varios paradigmas MDE, como *MDA* o *Arquitectura Dirigida por Modelos* [35] y el desarrollo específico del dominio [34], los cuales se basan en los mismos principios: (i) uso de modelos para representar aspectos del software a un cierto nivel de abstracción, (ii) los modelos se expresan mediante lenguajes específicos del dominio (*Domain-Specific Languages*), (iii) la sintaxis abstracta de los DSL se expresa mediante metamodelos y (iv) se usan transformaciones de modelos para automatizar el desarrollo de software. A continuación se van a introducir brevemente estos principios.

2.1.1 Metamodelado

Un metamodelo es un modelo que describe conceptos y relaciones en un determinado dominio. Un metamodelo es normalmente definido como un modelo conceptual orientado a objetos expresado en un lenguaje de metamodelado como Ecore [42] o MOF [6]. Un lenguaje de metamodelado es a su vez descrito por un modelo llamado el *metametamodelo*. Los lenguajes de metamodelado proveen cuatro elementos principales para expresar metamodelos:

- Clases (también llamadas metaclases) para representar conceptos del dominio.
- Atributos para representar propiedades de los conceptos del dominio.
- Relaciones de asociación (agregaciones y referencias) entre pares de clases para representar conexiones entre conceptos del dominio.
- Generalizaciones entre clases hijas y clases padres para representar especializaciones entre conceptos del dominio.

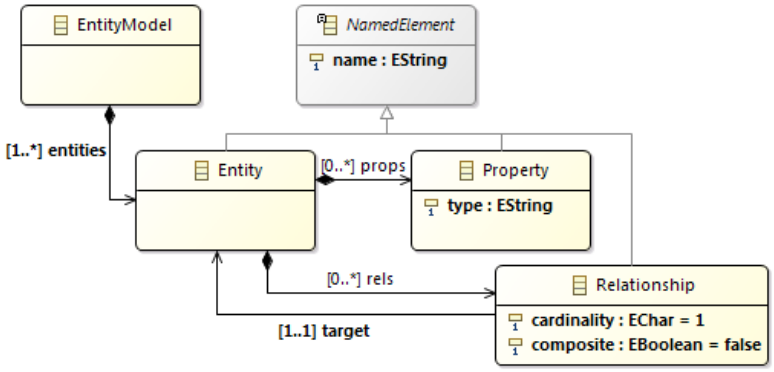


Figura 2.1: Metamodelo sencillo de entidades, propiedades y relaciones.

Normalmente la notación de diagramas de clases UML se usa para representar estos conceptos y relaciones. Por ejemplo, la figura 2.1 muestra un metamodelo para expresar modelos de entidades de negocio formados por entidades que tienen propiedades y relaciones entre entidades. Existen dos tipos de relaciones de asociaciones entre metaclases de un metamodelo: *Agregación* y *Referencia*. Una relación de *Referencia*, que es representada como una asociación directa en UML, expresa que una instancia de la metaclase origen tendrá una referencia a una o más instancias de la metaclase destino. Por ejemplo, en el metamodelo ejemplo de la figura 2.1, aparece una referencia de *Relationship* a *Entity*. Por otro lado una relación de *Agregación*, que es representada como una agregación UML, expresa una relación *es parte de* de un elemento contenido en un elemento contenedor.

En la figura 2.1 se puede observar que un objeto *EntityModel* está formado (relación *entities*) por una o más *Entities* que a su vez están formados por *Properties* (relación *props*) y *Relationships* (relación *rels*). *Entity*, *Property* y *Relationship* heredan de *NamedElement* el atributo *name*. Una propiedad también tiene un tipo de dato (atributo *type*) y una relación tiene una cardinalidad (atributo *cardinality*) y un atributo booleano *composite* para indicar si es una agregación o referencia. Nótese que este metamodelo tiene conceptos similares a los que se usan para representarlo.

Un metamodelo también incluye un conjunto de reglas que establecen restricciones sobre los modelos que se pueden crear y que no se pueden expresar en el lenguaje de metamodelo, expresadas en lenguajes como *OCL* [29] o basados en OCL. Este conjunto de

reglas se puede utilizar, por ejemplo, para asegurar que dos entidades no pueden tener el mismo nombre o que una entidad raíz no puede ser agregada por ninguna otra entidad.

2.1.2 Transformaciones de modelos

Una solución MDE consiste en una cadena de transformaciones de modelos que terminan generando unos artefactos software determinados a partir de uno o varios modelos de partida. Existen tres tipos de transformaciones: *modelo a modelo* (*m2m*), *modelo a texto* (*m2t*) y *texto a modelo* (*t2m*).

Las transformaciones *m2m* generan un modelo destino a partir de un modelo origen estableciendo un *mapping* entre los elementos definidos en ambos metamodelos. Estas transformaciones suelen ser usadas como etapas intermedias de una cadena de transformaciones para reducir el salto semántico antes de pasar a una transformación *m2t*. Las transformaciones suelen ser expresadas en lenguajes declarativos o imperativos como *ATL* [31] y *QVT* [4].

Las transformaciones *m2t* generan información textual (por ejemplo, código fuente o documentos XML) a partir de un modelo de entrada. Estas transformaciones generan los artefactos objetivo de las cadenas de transformación, y suelen ocupar el último lugar de las cadenas de transformaciones. Las transformaciones suelen ser expresadas en lenguajes de plantillas como *Xtend* [15].

El caso de transformaciones *t2m* es algo especial y específico de escenarios en los que se realiza reingeniería de software e interoperabilidad de herramientas [23]. Con el fin de obtener un modelo de entrada se recurre a una *inyección* consistente en analizar un artefacto textual y generar un modelo equivalente. Un lenguaje de transformaciones de este tipo es *Gra2MoL* [30].

2.1.3 Lenguajes específicos del dominio (DSL)

La aparición de los lenguajes de programación de propósito general (*General Purpose Languages*, *GPL*) ha sido la innovación que mayor incremento de productividad ha producido en el desarrollo de software. No obstante, el salto semántico entre el dominio del problema y los conceptos proporcionados por estos lenguajes es todavía grande y se requiere un gran esfuerzo para plasmar las soluciones en código GPL.

En contraste a estos *GPLs*, los lenguajes específicos del dominio (*Domain-Specific Languages*, *DSLs*) son lenguajes diseñados para solucionar problemas de un dominio muy específico con el fin de reducir el salto semántico. Los DSL favorecen escribir soluciones más simples y legibles, y producen beneficios similares a los que supuso el paso del ensamblador a los GPLs. En el contexto MDE, los términos DSL y lenguaje de modelado se utilizan para referirse a los lenguajes utilizados para construir modelos cuya estructura

viene determinada por un metamodelo. La creación de pequeños DSLs para dominios muy específicos se está convirtiendo en el principal uso de MDE en las compañías de software [49]. Un DSL se compone de tres elementos básicos:

- Una sintaxis abstracta que describe el conjunto de conceptos del lenguaje, sus relaciones y las reglas para combinarlos.
- Una sintaxis concreta que describe la notación del DSL. Puede ser textual, gráfica o una combinación de ambas [33]. El desarrollo de un DSL textual suele ser más sencillo que el de un DSL gráfico, por lo que en ocasiones se suele comenzar con un DSL textual como prueba de concepto y, mediante sucesivas iteraciones, llegar a un DSL gráfico [46].
- Una semántica asociada que describe cómo se interpretan las especificaciones DSL (los modelos DSL). La semántica se define normalmente a través de la generación de código o bien mediante la interpretación del código del DSL.

Las técnicas de implementación de DSLs se clasifican en tres categorías [26]: DSLs externos, DSLs internos y DSLs creados con herramientas basadas en metamodelos. Esta última categoría en realidad termina produciendo DSLs externos, por lo que se consideran únicamente las dos primeras categorías.

Un DSL externo se crea desde cero. Una vez que la gramática se ha definido, un analizador (*parser*) es creado para reconocer sentencias de DSL. La estructura de datos generada por el analizador se usa para construir un compilador o intérprete que define la semántica. Existen dos grupos de herramientas para facilitar la tarea de creación de DSLs textuales externos: Orientadas por la gramática (como *Xtext*[1]) y orientadas por el metamodelo (como *EMFText*[2]). En cuanto a herramientas para crear DSL gráficos, destacan *MetaEdit*[3] y *Sirius*[4].

Un DSL interno, o embebido, utiliza un GPL como lenguaje huésped. Este tipo de DSL se desarrolla mejor en el contexto de GPLs funcionales u orientados a objetos y para ello se crea una librería de métodos que proveen la sintaxis concreta del DSL. Ejemplos de DSLs embebidos son RubyTL [43] en Ruby o Kiama [17] en Scala.

2.2 Eclipse Modeling Framework y Ecore

Durante la realización de este proyecto se han desarrollado una serie de utilidades de visualización utilizando *Eclipse Modeling Framework* (*EMF*) [42]. Esta herramienta es un framework que proporciona la infraestructura básica para desarrollar proyectos de

[1] `https://eclipse.org/Xtext/`.
[2] `http://www.emftext.org/index.php/EMFText`.
[3] `http://www.metacase.com/mep/`.
[4] `https://eclipse.org/sirius/`.

modelado sobre la plataforma Eclipse. EMF es el framework de modelado más conocido, utilizado y extendido y ha contribuido significativamente a que el desarrollo MDE se haya extendido al ámbito académico e industrial. EMF se compone de un lenguaje de metamodelado denominado *Ecore* y de utilidades para crear y manipular modelos y metamodelos Ecore. Ecore proporciona los elementos básicos para construir metamodelos que fueron señalados anteriormente. La figura 2.2 muestra la jerarquía de clases que forman Ecore.

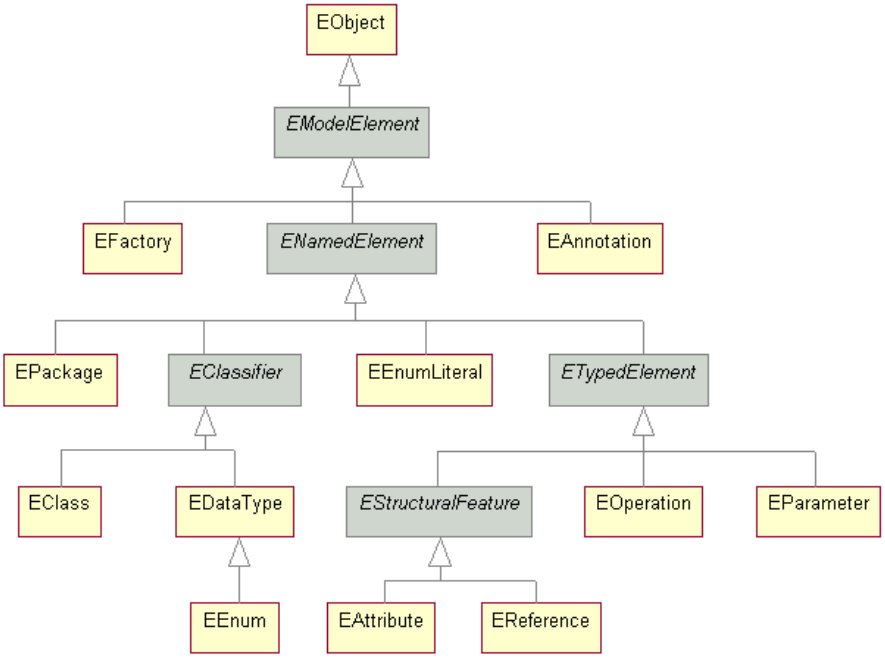


Figura 2.2: Jerarquía de clases de *Ecore*.

Finalmente, EMF provee una serie de herramientas para producir clases Java para un modelo, una serie de clases adaptadores que permiten la visualización y edición de modelos así como un editor básico del que se hará uso durante el desarrollo del proyecto.

2.3 Sirius

Sirius [12] es una herramienta de creación de DSLs gráficos desarrollada por *Obeo* [11] en el año 2007 para desarrollo interno del grupo *Thales* [13], una compañía francesa dedicada al desarrollo de sistemas de información. El proyecto pasó a ser una herramienta *Open Source* bajo el amparo de la *Eclipse Software Foundation* [2] en el año 2013 [22]. La apertura de la herramienta a la comunidad ha influido de manera notable en la ampliación de funcionalidad y opciones de la misma.

Dado un metamodelo, Sirius permite especificar una sintaxis gráfica y genera un editor con la capacidad de crear, visualizar y manipular modelos de forma sencilla e intuitiva sin generar código intermedio.

El funcionamiento general de Sirius consiste en definir una serie de vistas o diagramas (árboles, diagramas, tablas) y para estas vistas definir elementos visuales asociados a elementos del metamodelo. Es decir, para cada elemento del metamodelo es posible crear una representación visual a mostrar en el diagrama. Los lenguajes *OCL* [29] y *Acceleo* [7] se utilizan para expresar las propiedades y restricciones sobre los diagramas.

El propósito de los editores generados por Sirius es la creación o edición de modelos que conforman un metamodelo, más que servir como herramientas de visualización. Sin embargo, dado un modelo, el editor creado para su metamodelo lo visualizará directamente sobre un lienzo. Como veremos en el capítulo 6, en este trabajo se ha aprovechado esta característica de Sirius para visualizar esquemas de bases de datos NoSQL.

En Sirius es posible indicar, además de cómo se representa cada elemento de un metamodelo, operaciones para crear elementos definidos en el metamodelo y operaciones para definir el comportamiento de elementos cuando estos se seleccionan, se arrastran o se conectan.

2.3.1 Definición de un DSL gráfico con Sirius

La instalación de Sirius puede realizarse a través de la página web de la herramienta [5] y de distintas formas: Mediante *drag and drop*, mediante *Eclipse Marketplace* [16] o mediante un *Update site* [18]. Una vez instalado Sirius, se debe lanzar la ejecución de una instancia de Eclipse a partir del proyecto que guarda el metamodelo que va a ser representado. En esta nueva instancia de Eclipse es posible crear un *Viewpoint Specification Project*. Este proyecto contiene un fichero de descripción de vistas con extensión *odesign*, donde el usuario puede trabajar.

Los ficheros *odesign* permiten especificar vistas de distintos tipos. El proceso general de creación de cada vista es el siguiente:

1. Diseño de la estructura general de la vista: Qué elementos van a mostrarse, con qué nivel de anidamiento y qué partes de los mismos van a ser destacadas.
2. Asignación de una representación a cada elemento a mostrar. Esta consistirá en una forma geométrica con un color, una etiqueta y un comportamiento determinado.
3. Definición de relaciones visuales entre los objetos. En algunos casos las relaciones visuales tienen un equivalente en el metamodelo.
4. Agregación de aspectos adicionales tales como filtros de un tipo de clase u operaciones de navegabilidad entre vistas para mejorar la experiencia del usuario.

Restaría un paso de desarrollo consistente en indicar operaciones para la creación de elementos con el fin de poder agregar nuevos objetos a un modelo cualquiera desde una paleta con estos elementos. Aunque durante el proceso de aprendizaje de la herramienta llevado a cabo esta habilidad ha sido adquirida, en esta tesis de máster no se ha utilizado dado que la manipulación de un modelo de estas características no tiene sentido ni efecto sobre la base de datos NoSQL subyacente.

Para facilitar la comprensión del capítulo 6 en el que se explicará el uso de Sirius para visualizar esquemas NoSQL, se va a mostrar un DSL para el metamodelo simple mostrado en la figura 2.1 llamado *EntityMM*. Solo se explicará cómo especificar la representación de las clases *Entity* y *Property*:

En primer lugar, se define una vista de tipo diagrama para el metamodelo *EntityMM*. Para ello se debe indicar la clase inicial de la que se parte: *entity.EntityModel*. A continuación se crea un nodo contenedor, *Container*, con los siguientes parámetros de interés:

```
Id: Entity_container
Label: Entity_container
Domain_Class: entity.Entity
Semantic_Candidates_Expression: [self.entities/]
```

- *Id*: El identificador de este tipo de nodos en el diagrama. Debe ser único.
- *Label*: La etiqueta asociada al identificador. Por defecto coincide con este.
- *Domain class*: La clase del metamodelo a la que se corresponde esta representación: *entity.Entity*.
- *Semantic candidates expression*: Una expresión en *Acceleo3* utilizada para indicar, partiendo del elemento raíz, qué elementos semánticos del tipo indicado en *Domain class* se van a representar de esta forma. En este caso existe una relación que parte de *entity.EntityModel*, llamada *entities*, que se puede seguir para llegar hasta *entity.Entity*.
- *Children presentation*: Con esta opción se puede indicar si se desea que los elementos anidados se muestren como nodos manipulables por el usuario (cajitas que el usuario puede arrastrar, mover y redimensionar) o como elementos de una lista.

Una vez se han capturado los elementos a representar, falta únicamente decidir la representación con un estilo y más propiedades, de las cuales destacan:

- La etiqueta: De un tamaño determinado, en negrita y con una expresión. Esta expresión se define de nuevo en *Acceleo3* y puede ser estática, como "`['My label'/]`", o dinámica, como "`[self.<field>/]`". Nótese cómo en este punto se ha *cambiado de contexto*, y las referencias a *self* ya no se refieren a *EntityModel*, como en el punto anterior, sino a *entity.Entity*: Sirius ha *navegado* hasta este elemento al representarlo.
- La representación: Es posible escoger un gradiente con colores de inicio y fin.

El siguiente paso consiste en representar objetos *entity.Property*. Para ello se definirá de forma anidada la siguiente representación:

```
Id: Property_node
Label: Property_node
Domain_Class: entity.Property
Semantic_Candidates_Expression: [self.props/]
```

La figura 2.3 muestra cómo quedaría el fichero *odesign* de diseño para este metamodelo en Sirius, y la figura 2.4 muestra los resultados obtenidos a partir de un modelo de entrada mostrando también las relaciones entre entidades como flechas. Esto último se conseguiría definiendo una representación para el elemento *entity.Relationship* como un conector entre elementos.

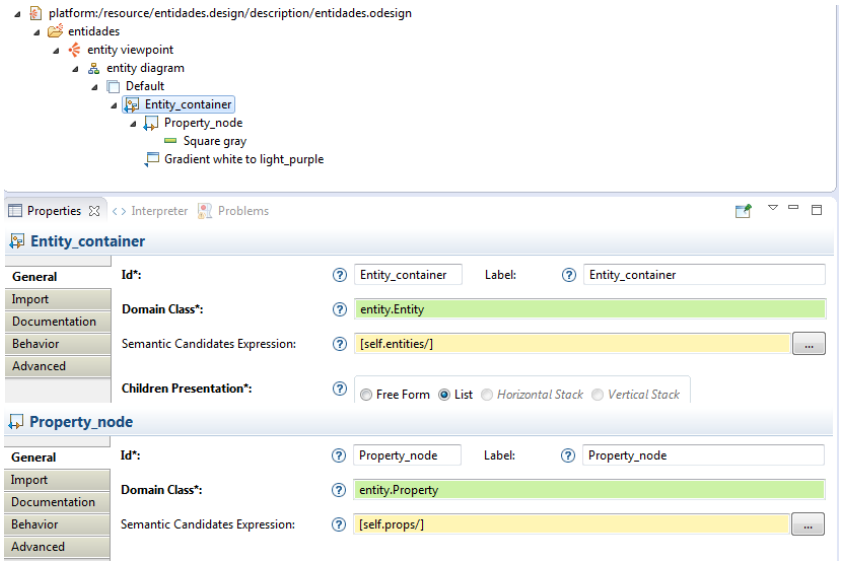


Figura 2.3: Ejemplo trivial de cómo representar elementos con Sirius.

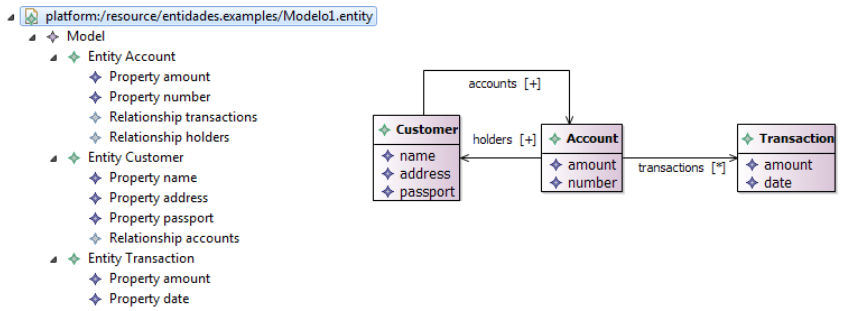


Figura 2.4: Modelo de ejemplo del metamodelo *entityMM* y su representación con Sirius.

2.4 Xtend

El lenguaje **Xtend** [15] ha sido el escogido para escribir transformaciones *modelo a texto* (*m2t*) de modelos a código Javascript. Xtend es un dialecto de Java tipado estáticamente que, cuando se compila, genera código Java 5. La salida Java generada puede ejecutarse directamente y tiende a proporcionar una eficiencia similar al código Java puro. Este lenguaje está disponible como un *plugin* para Eclipse y, a partir de su instalación, se pueden crear ficheros de código Xtend organizados en clases.

Xtend proporciona un mecanismo de definición de plantillas para la generación de texto a partir de modelos. Se proporcionan estructuras de control y construcciones del lenguaje de las que Java carece, y que permiten recorrer fácilmente un modelo de entrada proporcionado. A medida que se recorre este modelo de entrada se puede especificar el texto de salida en función de cada elemento que conforma este modelo. Otra ventaja de Xtend es que permite declarar expresiones de retorno, y este es un mecanismo muy útil cuando se desea, como en este caso, formar el código final como una cadena de texto de forma incremental.

Se han implementado dos clases en este lenguaje, que son accesibles desde código Java sin ningún problema de interoperabilidad, reciben un modelo de entrada para el que generar código y generan código devuelto en un objeto *CharSequence*. En este objeto se almacena el código generado en función de los elementos que se recorren del modelo mediante invocación de métodos polimórficos, que aseguran que existiendo sobrecarga de métodos con distintos parámetros, siempre se ejecute el método cuya entrada se ajusta al objeto enviado como parámetro.

3 Fundamentos de bases de datos NoSQL

Las aplicaciones modernas que deben tratar con grandes colecciones de datos han puesto de manifiesto las limitaciones de los sistemas gestores de bases de datos relacionales. Este hecho ha motivado el desarrollo de un cada vez mayor número de sistemas no relacionales, con el propósito de superar los requisitos de dichas aplicaciones. Especialmente, la habilidad de representar datos complejos sin renunciar a la escalabilidad para manejar conjuntos de datos grandes y el incremento del tráfico de datos. Para definir esta nueva generación de sistemas de bases de datos se emplea el término *NoSQL* (*Not SQL/Not Only SQL*).

El concepto de *esquema* de base de datos juega un papel central en sistemas de bases de datos relacionales, pero la mayoría de sistemas de bases de datos NoSQL no tienen un esquema, son *schemaless*. Esta falta de definición de esquema proporciona una gran flexibilidad al facilitar el almacenamiento de información no uniforme y la evolución de dicha información. Sin embargo, el no poseer esquema a su vez acarrea la pérdida de ciertos beneficios como son la comprobación por parte del sistema gestor que los datos introducidos son conformes al esquema o que el código que manipula datos pueda también aprovechar el esquema.

3.1 Tipos de sistemas NoSQL

Los sistemas NoSQL se pueden clasificar en distintas categorías de acuerdo a su modelo de datos, además de que sistemas en la misma categoría pueden mostrar distintas prestaciones. Sin embargo, la mayoría de los sistemas NoSQL tienen una serie de características comunes: No utilizan el lenguaje SQL, los esquemas no son necesarios para especificar la estructura de datos, la ejecución en clústers es el principal factor para determinar su diseño, y son desarrollados como iniciativas *open source* [40]. Los cuatro sistemas NoSQL más extendidos son:

- *Clave-valor*: Las bases de datos *clave-valor* proveen una forma sencilla de almacenamiento de datos mediante pares *clave-valor*. Esta estructura es similar a una colección de tipo *Map* de cualquier lenguaje de propósito general, y la base de datos no asume ninguna estructura en los datos.
- *Basados en documentos*: Las bases de datos basadas en documentos también almacenan la información como una serie de pares *clave-valor*, pero en este caso cada valor está formado por un documento estructurado por el que es posible navegar y realizar búsquedas para obtener un dato en particular.
- *Basados en familias de columnas*: Las bases de datos basadas en familias de columnas están organizadas como una colección de filas, cada una de ellas con una clave y una serie de familias de columnas que conforman a su vez un conjunto de pares *clave-valor*.
- *Basadas en grafos*: Las bases de datos basadas en grafos se organizan en entidades, que se almacenan como nodos con propiedades, y relaciones entre entidades, que lo hacen como aristas con propiedades y dirección. Estas relaciones con dirección son las que permiten interpretar los datos almacenados.

3.2 Bases de datos orientadas a la agregación

Los datos complejos son tratados comúnmente en bases de datos relacionales a través de *joins* y claves ajenas o referencias entre tablas. Sin embargo las referencias y agregaciones entre objetos son una forma más apropiada de representar este tipo de datos. De entre ellas, las agregaciones de objetos son normalmente preferidos a las referencias de objetos en las bases de datos NoSQL, porque los datos se encuentran distribuidos a través del clúster para obtener escalabilidad, y las referencias de objetos podrían suponer contactar distintos nodos remotos del clúster. Esta preferencia, la *orientación a la agregación* se ha identificado como una característica compartida entre los modelos de datos de tres de los sistemas NoSQL más extendidos [40]: *clave-valor*, *basados en documentos* y *basados en familias de columnas*.

Estos sistemas almacenan información semiestructurada, y como se ha comentado de manera *schemaless*, lo que proporciona una gran flexibilidad. De esta forma la información que tiene distinta estructura para una entidad puede ser almacenada sin problemas. La evolución de los datos almacenados ocurre también de una forma más sencilla dado que no hay necesidad de modificar el esquema para soportar esta evolución, y distintas versiones de datos pueden coexistir simultáneamente.

Los datos semiestructurados se caracterizan porque tienen una estructura implícita y no uniforme, que puede evolucionar rápidamente [21]. Estos datos son expresados en

formatos como XML o JSON, que permiten representar información de forma jerárquica mediante etiquetas o símbolos utilizados como separadores.

JSON [3] es un estándar de formato de texto utilizado para representar datos. Esta notación está superando a XML como formato de intercambio de datos porque es más simple y legible. Un objeto o documento JSON está formado por una serie de pares *clave-valor*, y estos valores pueden ser de tipo primitivo (numérico, string, booleano), un objeto, o un array de valores. JSON es usado en la mayoría de sistemas NoSQL como formato para registrar los datos y para la importación/exportación de los mismos.

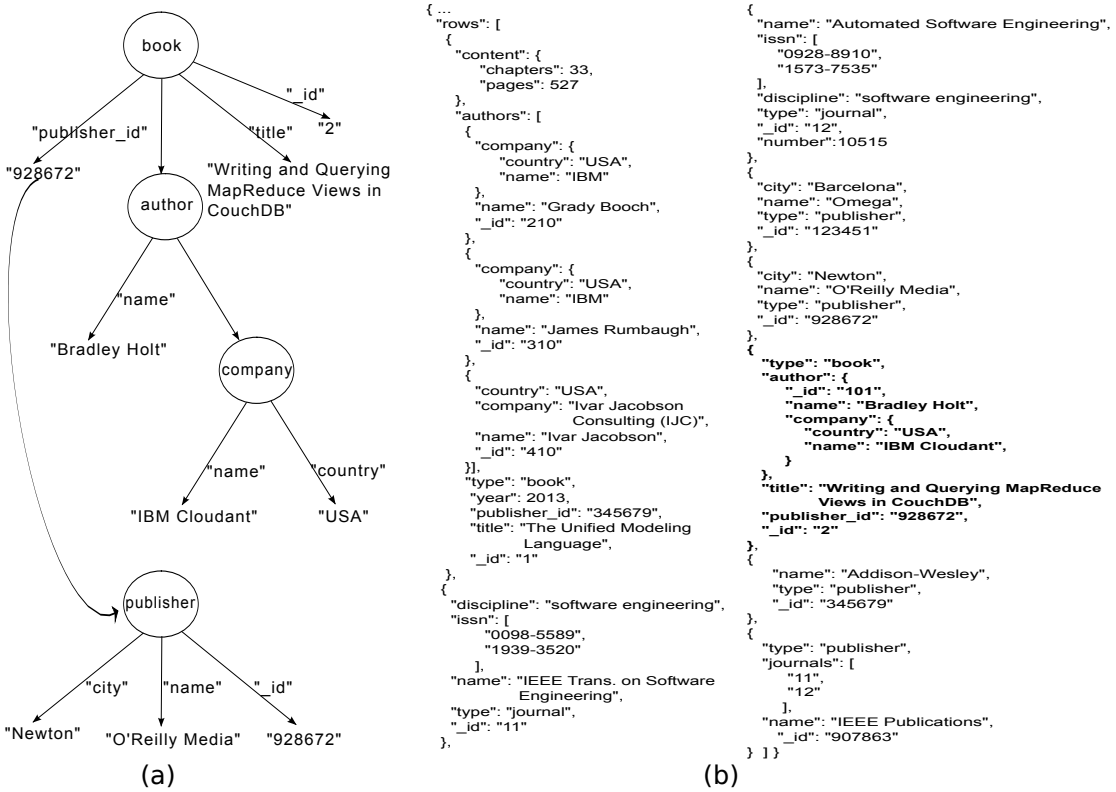


Figura 3.1: Base de datos de ejemplo y una representación en árbol de *Book*.

Un conjunto de datos semiestructurados se puede representar como un árbol cuyos nodos hoja son valores de tipos primitivos (cadenas de caracteres, enteros, números de punto flotante, booleanos), y la raíz y los nodos intermedios son objetos (tuplas) o arrays de objetos y valores [24]. Las aristas son etiquetadas con el nombre de los atributos. Una raíz o un nodo intermedio tendrán un nodo hijo por cada atributo del objeto asociado. En la figura 3.1 se muestra el árbol que corresponde al objeto JSON que representa un objeto *book* con el campo *_id=2*. Las referencias entre datos son expresadas haciendo que el valor atómico del atributo (*publisher_id* en *book*) coincida en valor con otro atributo de un objeto *publisher* (en este caso, el campo *id*).

El término *agregación* se utiliza normalmente para referirse a una estructura consistente

en un objeto raíz que embebe a otros objetos. En la figura 3.1, el objeto *book* agrega a un objeto *author* que, además, agrega a un objeto *company*.

3.3 Ejemplo particular de base de datos NoSQL

Para este trabajo se considera una base de datos NoSQL como una colección arbitrariamente grande de objetos JSON que incluyen:

- Un campo de nombre *type* que describe el tipo de entidad.
- Algún tipo de identificador único para cada objeto, dado por el atributo *_id*.

Este formato es independiente de una base de datos NoSQL concreta, pero a su vez la mayoría de bases de datos NoSQL proporcionan un formato parecido:

- Las guías de *CouchDB*[1] recomiendan el uso de un campo *type* con este propósito.
- *MongoDB*[2] crea colecciones para cada tipo de objeto, por lo que el nombre de la colección es el valor del atributo *type* definido.
- En *HBase*[3] el campo *type* se podría obtener de una de las *familias de columnas*.
- En otras bases de datos NoSQL, sin embargo, puede ser necesario especificar estos campos manualmente.

La figura 3.1 muestra una base de datos simple que almacena objetos de tipo entidad *book*, *publisher* y *journal*. El primer objeto *book* agrega un array de entidades *author* (campo *authors*) y un objeto embebido para el contenido (campo *content*). Este campo *author* agrega otro objeto embebido que almacena la compañía para el que este trabaja (campo *company*). Tanto los objetos *book* como *publisher* contienen referencias. El objeto *book* tiene una referencia a su publicador (campo *publisher_id*), y *publisher* guarda una referencia a una lista de objetos *journal* (campo *journal*).

[1] `http://couchdb.apache.org/`.
[2] `https://www.mongodb.com/es`.
[3] `https://hbase.apache.org/`.

4 Estado del arte

DataDiversity llevó a cabo una encuesta a desarrolladores expertos que usaban NoSQL en la industria, a partir de la cual se elaboró un informe sobre el papel del modelado en bases de datos NoSQL [25]. La encuesta evidenció que una de las principales críticas en contra de los sistemas NoSQL es la falta de herramientas que proporcionen funcionalidad a la que las empresas están acostumbradas para el manejo de datos. A esto se añade la complejidad de manejar diferentes modelos de datos NoSQL.

Los autores del informe destacan que es preciso analizar el código para conocer el modelo de datos que se ha definido para una base de datos NoSQL, dado que la naturaliza *schemaless* de estas bases de datos implica que el esquema se encuentra implícito en este código. Por ello, están en marcha muchos trabajos de ingeniería inversa para extraer el esquema de los datos a partir del código. Se asegura que los modelos de datos son vitales para abordar los principales problemas que tiene la industria de desarrollo de software interesada en los sistemas NoSQL: Modelado de datos, gobernanza de datos, documentación y herramientas de referencia.

El informe concluye señalando que las herramientas para soporte del desarrollo NoSQL son inmaduras y la industria demanda funcionalidad similar a la disponible para sistemas relacionales desde hace unas dos décadas y que tiene que ver con el modelado de datos. En concreto, se identifican tres categorías de herramientas:

- *Visualización de modelos* que sirvan para el diseño de la base de datos, la toma de decisiones, la documentación o como resultado de procesos de ingeniería inversa.
- *Generación de código* a partir de modelos de alto nivel, lo que incrementaría la productividad y reduciría el número de errores, entre otros importantes beneficios.
- *Manejo de metadatos*, esencial en algunos escenarios como la integración o interoperabilidad de herramientas.

Además se subraya la naturaleza políglota que tendrán las aplicaciones de manejo de bases de datos en el futuro inmediato, en las que coexistirán las bases de datos relacionales con los diferentes modelos NoSQL. Por ello, las herramientas de manejo de modelos deberán tener en cuenta esta diversidad de sistemas.

ER/Studio[1] es una herramienta comercial para modelado de datos de las primeras en soportar modelos de datos NoSQL. En concreto, desde hace una año incluye la capacidad de mostrar diagramas de esquemas MongoDB que obtiene a partir de un proceso de ingeniería inversa sobre una base de datos. Sin embargo, no considera la existencia de diferentes entidades de una base de datos MongoDB y sólo soporta este sistema por el momento.

CA ERwin[2] es otra conocida herramienta de modelado de datos y de acuerdo con [47] sus desarrolladores trabajan en el soporte del modelado de datos NoSQL. Se utiliza un modelo de datos unificado definido para ERwin (*Unified Data Modelling*) para representar los esquemas de bases de datos relacionales y NoSQL. En el caso de sistemas NoSQL se realiza un proceso de inferencia y se muestran los diagramas de los esquemas obtenidos, aunque no se proporcionan detalles de cómo se llevará a cabo la inferencia y la visualización. Tan solo se menciona que se aplicarán diferentes tipos de técnicas para la ingeniería inversa como *machine learning*, inferencia guiada por el usuario a través de interfaces gráficas y mejora continua de la precisión del proceso de inferencia.

MongoDB Compass[3] es una herramienta gráfica incluida en MongoDB que permite analizar y comprender el esquema de la base de datos, e incluso construir consultas visualmente. Dada una colección, Compass muestra el tipo de sus campos y en el caso de colecciones con campos que no están en todos los objetos se muestra el porcentaje de objetos que tienen ese campo, o en el caso de campos *dispersos*, que no tienen un valor en todos los objetos, se muestra el porcentaje de objetos con un valor para ese campo. También muestra información estadística sobre los valores de un campo, por ejemplo un histograma para los valores de un campo de tipo entero.

Algunas herramientas ya soportan la inferencia de esquemas de bases de datos como se comenta en [37], pero en ellas no se aborda la visualización de los esquemas inferidos. Por ejemplo, *MongoDB-Schema* [39] es un prototipo inicial de herramienta destinada a inferir esquemas de objetos JSON y colecciones MongoDB. Por otro lado, recientemente, se ha publicado un algoritmo y una estructura de datos (*eSiBu-Tree*) [48] para encontrar esquemas versionados de bases de datos NoSQL basadas en documentos, el cual se ha validado con bases de datos reales (DBpedia, IMDB, etc.).

En [20] se presenta un enfoque para encontrar valores atípicos (*outliers*) en bases de datos NoSQL. Dada una base de datos NoSQL basada en agregación, se aplica un proceso de ingeniería inversa para obtener un esquema representado en formato JSON. El algoritmo de inferencia adapta estrategias propuestas para extraer DTDs de documentos XML a objetos JSON. Sin embargo, no se tratan las versiones de entidades ni las referencias entre entidades, y el esquema JSON extraído no representa explícitamente la agregación de entidades, solo maneja entidades de primer nivel. Primero se obtienen

[1] `https://www.idera.com/er-studio-enterprise-data-modeling-and-architecture-tools`.
[2] `http://erwin.com/products/data-modeler`.
[3] `https://www.mongodb.com/blog/post/getting-started-with-mongodb-compass?jmp=twt`.

resultados estadísticos como el porcentaje de aparición de una propiedad en un documento y entonces estos valores se usan para detectar *outliers* como 'propiedad adicional', o campo que aparece en menos de cierto valor umbral de los documentos.

El trabajo realizado en esta tesis de máster es parte de un proyecto más amplio destinado a definir un conjunto de herramientas para aplicar *NoSQL Data Engineering*, como se explica en [38]. En la línea de las conclusiones del informe [25], se pretende desarrollar un *toolkit* de herramientas para la visualización de esquemas y datos, la generación de código y el manejo de metadatos. La visualización se ha abordado en este trabajo; la inferencia de esquemas y generación de código es parte de una tesis doctoral que arrancó hace dos años, en la que ya se han obtenido algunos resultados como el algoritmo de inferencia basada en modelos que genera documentación gráfica y textual sobre las versiones de las entidades y que es descrito en [37], y la generación del código necesario para integrar una base de datos en el *Object-NoSQL mapper* `mongoose`[4] (código de definición de esquemas y de validación de datos).

A diferencia de ER/Studio, la solución propuesta en este trabajo abarca cualquier sistema NoSQL basado en agregación, y tiene en cuenta versiones de entidades y referencias. En cuanto a ERwin, cabe destacar que el proyecto aquí desarrollado coincide con los objetivos planteados en [47], aunque hay algunas diferencias significativas. El metamodelo que se manejará en este documento para representar los esquemas es más expresivo que el *Unified Data Model* utilizado en ERwin (por ejemplo, permite capturar absolutamente todas las variaciones de tipos, y por lo tanto, toda la información de metadatos de la base de datos), y se ofrece una solución MDE basada en metamodelos y transformaciones de modelos.

Las visualizaciones sobre los porcentajes de objetos que pertenecen a cada versión están relacionadas con los *outliers* obtenidos en [20], y las visualizaciones de MongoDB Compass fácilmente podrían ser implementadas en la herramienta desarrollada utilizando la clasificación de objetos definida en el capítulo 7 utilizando *D3.js* para la visualización.

[4] `http://mongoosejs.com/`.

5 Proceso de inferencia de esquemas NoSQL

En este capítulo se explica el proceso de inferencia definido en [37] para extraer esquemas de bases de datos NoSQL de sistemas basados en agregación (clave-valor, basados en documentos o familias de columnas). Estos esquemas de versiones serán el elemento central en torno al cual girará todo el desarrollo de visualización que se llevará a cabo.

Como se ha señalado anteriormente, la falta de esquema explícito (*schemaless*) es probablemente la propiedad más atractiva de las bases de datos NoSQL. Esta naturaleza *schemaless* provee de flexibilidad dando la capacidad de almacenar datos no uniformes y permitir de forma sencilla la evolución de dichos datos, como se apuntó en la sección 2. Sin embargo la ausencia de un esquema explícito no debe confundirse con la falta de esquema, dado que siempre existe un esquema implícito en los datos de una base de datos. Cualquier desarrollador debe tener presente este esquema cuando accede a la base de datos, introduciendo correctamente los nombres y tipos de los campos cuando trata con operaciones de inserción o búsqueda, por ejemplo. Esto, unido al hecho de que simultáneamente pueden existir distintas versiones para una misma entidad, representa una tarea propensa a errores.

La ausencia de esquema, por otro lado, tiene ciertos inconvenientes. Cuando un esquema se define formalmente para una base de datos relacional, existen comprobaciones estáticas para asegurar que únicamente se manipulan (insertan, buscan) datos que conforman este esquema, lo que lleva a la capacidad de detección estática de errores por parte de los desarrolladores que interactúan con la base de datos.

Por tanto, la idea consiste en combinar la aproximación *schemaless* con mecanismos que garanticen un correcto acceso a los datos [27, 20]. Esta aproximación está comenzando a ser explorada [20, 39, 50] motivada por ciertas herramientas de bases de datos NoSQL que se beneficiarían de la existencia de esquemas, tales como herramientas que permiten ejecutar consultas estilo SQL, o migrar datos.

Un esquema de un modelo de datos orientado a agregaciones está formado por un conjunto de entidades conectadas a través de dos tipos de relaciones: *agregación* y *referencia*. Cada entidad tendrá uno o más campos especificados por un nombre y un tipo. La inferencia de esquemas en una base de datos NoSQL es un proceso de ingeniería inversa

en el que se puede hacer uso de técnicas MDE. Los metamodelos proveen un formalismo para representar el conocimiento acumulado a un alto nivel de abstracción, y la automatización se puede llevar a cabo utilizando transformaciones de modelos. Se puede diseñar, por tanto, una solución MDE de ingeniería inversa de obtención de esquemas versionados a partir de bases de datos NoSQL orientadas a agregación.

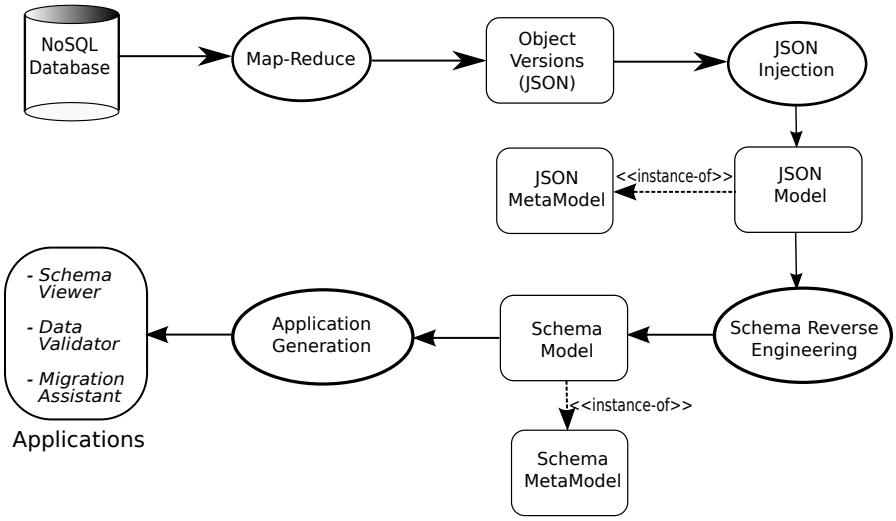


Figura 5.1: Vista general del proceso MDE implementado.

La figura 5.1 muestra la arquitectura de la solución MDE ideada para inferir los modelos de esquemas NoSQL que son la entrada a las herramientas de visualización desarrolladas en este trabajo de fin de máster. Dicha arquitectura se ha organizado en tres etapas:

- Se aplica una operación *map-reduce* para extraer una colección de objetos JSON que contengan un objeto por cada versión de cada entidad. De esta forma se obtiene el mínimo número de objetos necesarios para efectuar el proceso de inferencia.
- Se inyecta esta colección en un modelo que conforma el metamodelo JSON, que se puede obtener fácilmente haciendo un *mapping* de los elementos de la gramática de JSON en los elementos del metamodelo.
- Por último se efectúa el proceso de ingeniería inversa como una transformación *m2m* cuya entrada es el modelo JSON, y que genera un modelo que conforma el metamodelo *NoSQL_Schema* de la figura 5.2.

Los modelos *NoSQL_Schema* inferidos se pueden visualizados, como se verá más adelante, o utilizados para desarrollar herramientas que se pueden clasificar en dos categorías:

- Utilidades de bases de datos que requieren conocimiento de su estructura.
- Ayudas para el desarrollo de soluciones a problemas causados por la ausencia de un esquema explícito. Por ejemplo validadores de datos, *scripts* de migración o diagramas de esquemas.

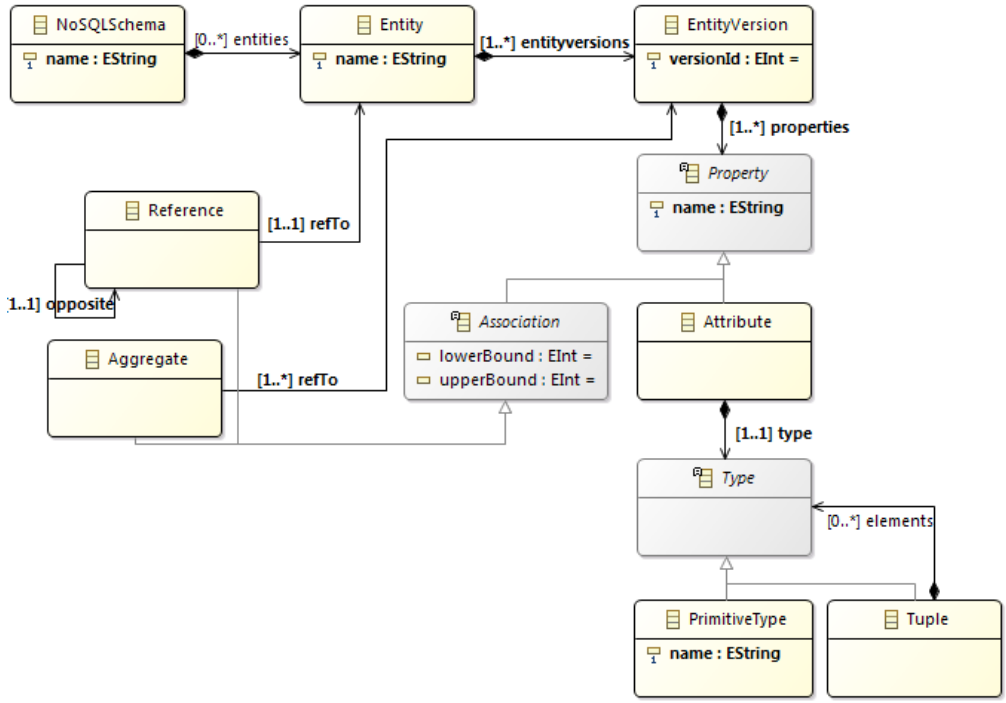


Figura 5.2: Metamodelo *NoSQL_Schema* utilizado para representar esquemas.

La figura 5.2 muestra el metamodelo *NoSQL_Schema* que representa esquemas de bases de datos NoSQL orientadas a la agregación. Un esquema (metaclase *NoSQL_Schema*) está formado por una colección de entidades (*Entity*) y una o más versiones (*EntityVersion*) existen para cada entidad. Una versión se define a partir de una serie de propiedades (*Property*) que pueden ser de tipo *Attribute* o *Association*, dependiendo de si la propiedad representa un tipo (*PrimitiveType* o *Tuple*) o bien una relación entre dos entidades. Una tupla denota una colección que puede contener otros tipos primitivos y otras tuplas. Una asociación puede ser un agregado (*Aggregate*) o una referencia (*Reference*). La cardinalidad de la asociación se almacena en los atributos *lowerBound* y *upperBound*, y puede tomar valores 0, 1 o -1.

Una agregación se conecta con una o más versiones (*[1..*]refTo*) porque un objeto embebido puede agregar un array con objetos de distintas versiones. Por el contrario una referencia está conectada a una entidad (*[1..1]refTo*) debido a que se necesita almacenar que una versión referencia a cierta entidad. La autoreferencia *opposite* de una metaclase *Reference* se utiliza para establecer la relación bidireccional.

Como se verá en los siguientes capítulos este metamodelo es crucial para los procesos de visualización. El trabajo desarrollado ha consistido en idear representaciones visuales para modelos conformes a este metamodelo y crear un generador de código que permite clasificar los objetos de una base de datos de acuerdo a la versión a la que pertenecen.

6 Visualización de esquemas NoSQL

6.1 Introducción

Este capítulo describe el proceso seguido para desarrollar una herramienta para la visualización de esquemas de datos NoSQL obtenidos a partir de los datos almacenados en la bases de datos según el proceso de inferencia explicado en la sección 5.

Dicho proceso obtiene los esquemas como modelos conformes al metamodelo *NoSQL_Schema*. Dado que existen versiones de entidades, no existe un único esquema de la base de datos sino un conjunto de versiones de esquemas formadas por versiones de entidades. El propósito de la herramienta es visualizar esquemas versionados.

Se ha diseñado e implementado una solución *MDE* que consta de una transformación *m2m* y de la definición de una sintaxis gráfica para los modelos generados. La transformación *m2m* enriquece un modelo *NoSQL_Schema* con cierta información que facilita su representación visual, en concreto la noción de versión del esquema o esquema versionado que no existe en *NoSQL_Schema*. Este nuevo metamodelo se ha denominado *Extended_NoSQL_Schema*. Como se comentó en la sección 2.3, la visualización ha sido llevada a cabo mediante *Sirius*.

Por último, se ha comprobado el correcto funcionamiento de todo este proceso con varios modelos de entrada. El proceso completo a detallar puede observarse en la figura 6.1.

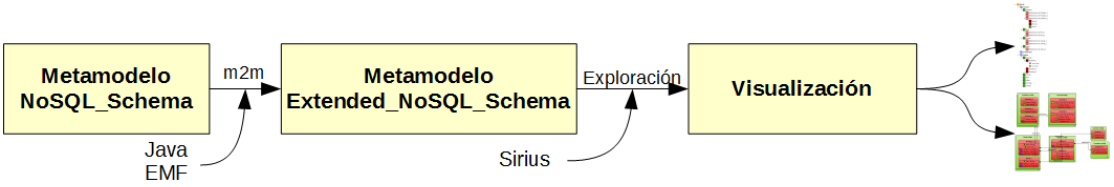


Figura 6.1: Proceso de visualización de esquemas NoSQL.

6.2 Diseño del metamodelo Extended_NoSQL_Schema

El metamodelo *NoSQL_Schema*, si bien mantiene información sobre entidades y sus versiones, no maneja el concepto de *versión de esquema* que se desea representar visualmente. Se define por *versión de esquema* aquella agrupación de entidades y versiones tal que una de las versiones actúa de **versión raíz** del esquema y el resto de entidades y versiones del esquema son navegables desde esta versión raíz. Formalmente, una *versión de esquema* está compuesta por:

- Una *versión raíz* del esquema: Aquella que no es agregada por ninguna otra versión.
- Un conjunto de entidades a las que o bien se puede acceder navegando por una relación *Reference* desde la *versión raíz*, o bien desde alguna otra versión incluida en el esquema de relación.
- Un conjunto de versiones a las que se puede acceder navegando por una relación *Aggregate* desde la *versión raíz*, o bien desde alguna otra versión incluida en el esquema de versión, o bien añadida porque la entidad que contiene a esta versión ha sido añadida al esquema, siguiendo las indicaciones del punto anterior.

Nótese que la existencia tanto de entidades como de versiones de entidades en una versión de un esquema es motivada por el hecho de que en los modelos *NoSQL_Schema* las referencias son hacia entidades mientras que las agregaciones agregan a versiones de entidades.

Aunque las versiones de un esquema se podrían obtener recorriendo un modelo *NoSQL_Schema*, este proceso podría ser costoso si el modelo es grande. Por ello, se ha decidido definir el metamodelo *Extended_NOSQL_Schema* (mostrado en la figura 6.2) que registra de forma explícita las versiones de esquema, y aplicar una transformación *m2m* para convertir modelos *NoSQL_Schema* en modelos *Extended_NoSQL_Schema*. Esta transformación realiza una copia del modelo de entrada y añade la siguiente información:

- *SchemaVersion*: Concepto de *versión de esquema*.
- Relación *1..1 root* de *SchemaVersion*: Para indicar la versión que actúa como raíz del *versión de esquema*.
- Relación *0..* entityVersions* de *SchemaVersion*: Para indicar todas las versiones que no son raíces y pertenecen al esquema.
- Relación *0..* entities* de *SchemaVersion*: Para indicar todas las entidades incluidas en el esquema.

El proceso de transformación de un modelo a otro sigue el siguiente proceso:

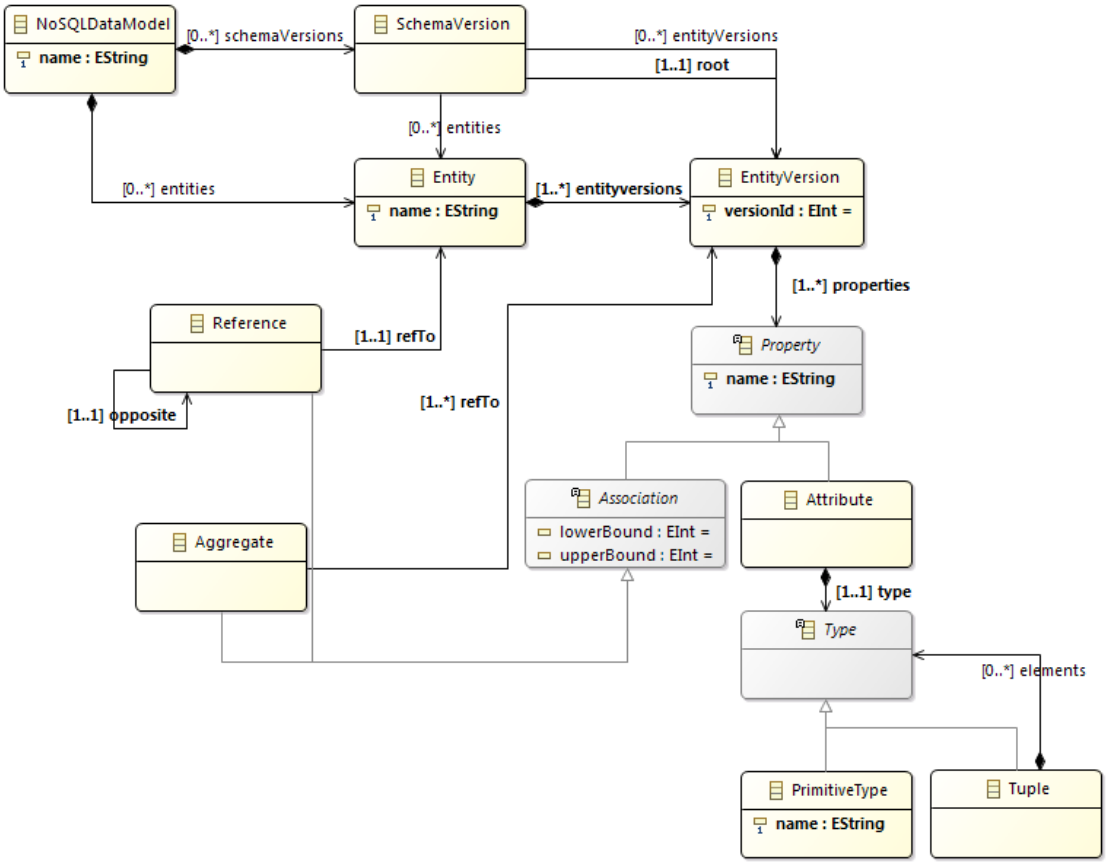


Figura 6.2: Metamodelo *Extended_NoSQL_Schema*.

- Para cada elemento del modelo *NoSQL_Schema* se debe crear un elemento equivalente del modelo *Extended_NoSQL_Schema*. Lo único reseñable de este proceso es que para un correcto funcionamiento las relaciones del tipo *Referencia* y *Agregación* deben resolverse en último lugar, para asegurar que los elementos que deben relacionarse existen.
- Una vez realizada la copia anterior, se deben crear las *versiones de esquemas* existentes:
  1. Se obtiene una colección de las *versiones raíz*. Primero se forma una colección con todas las versiones y se van eliminando a aquellas que son agregadas.
  2. Para cada versión raíz se deben recorrer todas sus relaciones *Agregación* y *Referencia*. Cada versión accedida se añade a la versión del esquema y será explorada de esta misma forma.
  3. Cada entidad a la que se navegue de esta forma se agregará al conjunto de la *versión de esquema*, y las versiones definidas para esta entidad se explorarán de la misma forma.

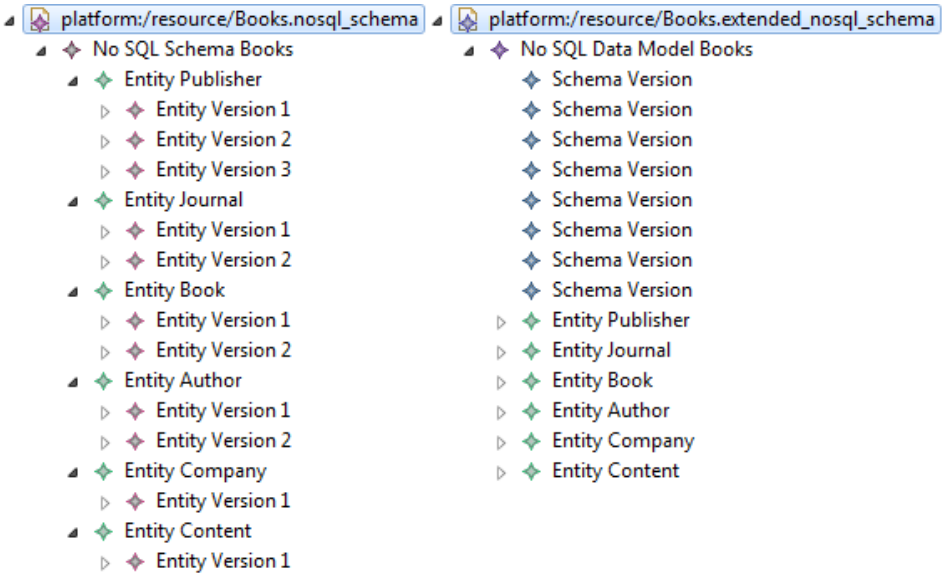


Figura 6.3: *Books.nosql_schema* y su equivalente *Books.extended_nosql_schema*.

4. Finalmente se agregan las *versiones de esquema* obtenidas al modelo *Extended_NoSQL_Schema*.

La figura 6.3 muestra un ejemplo de transformación de un esquema en un esquema extendido. En concreto, a un modelo *NoSQL_schema* se le añaden 8 versiones de esquema, una por cada versión raíz. El proceso de transformación se encuentra recogido en los proyectos Java **NoSQLSchema**, **NoSQLSchema.edit** y **NoSQLSchema.editor** del código fuente del proyecto (véase el anexo A). En concreto, la transformación *m2m* descrita se puede observar en *src/schema.analyzer/SchemaCollector.java*.

6.3 Diseño de las vistas de esquemas

Como paso previo a decidir la forma de implementar la visualización, se identifican el conjunto de requisitos que debería ser satisfecho por la herramienta de visualización de *versiones de esquemas* NoSQL. También se identifica qué vistas del esquema serían apropiadas para los desarrolladores y diseñadores de bases de datos. Los requisitos de la herramienta son:

1. Debe ser fácilmente integrable con toda la infraestructura de *EMF*.
2. Debe ser eficiente a la hora de proveer las distintas vistas del modelo, además de intuitiva y amigable con el usuario.
3. Debe ofrecer al usuario una vista de árbol desde la que navegar a todas las *versiones de esquemas* con todos sus elementos (*versión raíz*, entidades y versiones).

4. Debe ofrecer al usuario una vista global desde la que ver todas las entidades, versiones, atributos y relaciones del modelo.

5. Debe ofrecer al usuario una vista detallada de una *versión de esquema* en particular, con todos los elementos que la componen y sus detalles (entidades, versiones, atributos y relaciones).

En un primer momento dos opciones de implementación fueron consideradas: Utilizar *D3.js* o una librería similar de visualización o bien definir un DSL con una herramienta de creación de DSLs basada en metamodelos. A pesar de enfrentar un problema de visualización se optó por la segunda opción ya que se valoró que sería la que menos coste de desarrollo presentaría dados los requisitos del lenguaje de esquemas.

Además de esto, el DSL creado también podría ser usado para crear directamente esquemas, no solo esquemas inferidos. Aunque las bases de datos NoSQL no requieren la definición de un esquema antes de introducir los datos (naturaleza *schemaless* como se ha indicado en la sección 3), los desarrolladores necesitan diseñar inicialmente el esquema y este lenguaje podría serles útil. Además se obtendría un modelo que podría ser aplicado en la automatización de tareas como la generación del código requerido por *NoSQL-object mappers*. Sirius ha sido utilizado para crear el DSL por tratarse de una de las herramienta de creación de DSLs más potentes entre las que hay disponibles en Eclipse como se ha comentado en la sección 2.3.

6.3.1 Diseño de la vista de árbol

La vista de árbol es el eje central a partir del cual se puede navegar al resto de vistas. Como requisito de partida esta vista debe listar, como mínimo, todas las *versiones de esquemas* existentes. Sin embargo y dadas las posibilidades de Sirius aplicables a este caso, esta vista se ha enriquecido para mostrar, además, un índice inverso de versiones y un listado de las entidades existentes. Estas tres vistas son unidas en un único árbol con una raíz común, a partir de la cual se inicia el desarrollo. Esta raíz se debe crear indicando como parámetro que el metamodelo que se desea representar es *Extended_NoSQL_Schema*.

6.3.1.1 Árbol de versiones de esquemas

La figura 6.4 muestra la vista del árbol de *versiones de esquema* desarrollado para un ejemplo sencillo *Books.extended_nosql_schema*. En ella se observa un nodo que actúa a modo de *carpeta contenedora de versiones de esquemas*, llamado *Schemas*, y a continuación una serie de *versiones de esquemas* ordenados por la entidad a la que pertenecen. Una *versión de esquema* pertenece a una entidad si su *versión raíz* pertenece a dicha entidad, y es una forma sencilla de agrupar *versiones de esquemas* con el fin de navegar

fácilmente entre ellas. Por último dentro de cada *versión de esquema* se listan las entidades y versiones de dicho esquema, para que el usuario pueda determinar de un vistazo el contenido de cada esquema.

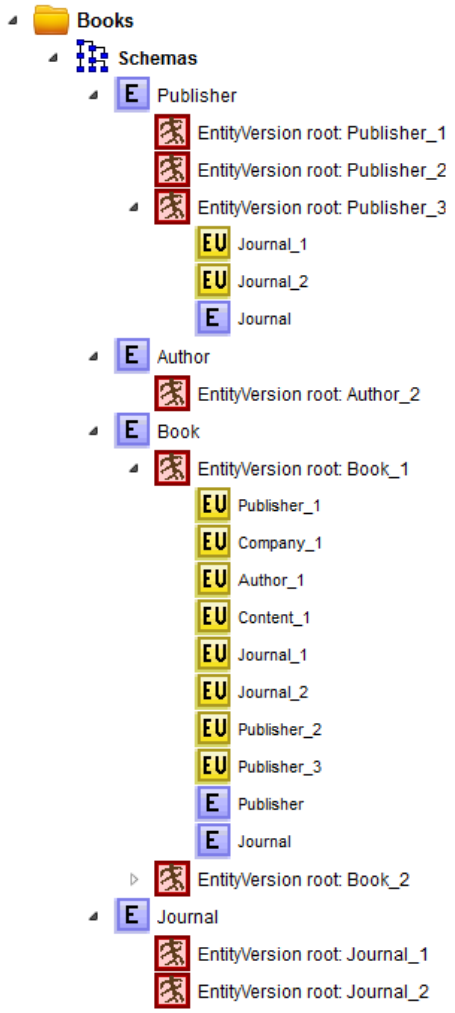


Figura 6.4: *Versiones de esquemas* sobre *Books.extended_nosql_schema*.

Para la representación visual se ha optado por diseñar un estilo compuesto por una etiqueta descriptiva y un icono distintivo con el fin de que el usuario pueda, de un vistazo, identificar el elemento que visualiza:

- Para las entidades se ha establecido un icono púrpura pálido con la letra *'E'* y como etiqueta el nombre de la entidad.
- Para las versiones se ha creado un icono amarillo con las iniciales *'EV'* y como etiqueta el nombre de la entidad a la que pertenece la versión seguido del número de versión.

- Para las *versiones de esquemas* se ha creado un icono que simboliza una raíz y una etiqueta que informa de qué versión es la *versión raíz* de la *versión de esquema*.
- Para los agregados y las referencias se han diseñado unos iconos con forma de flecha y colores azul y púrpura oscuros, con etiquetas que informan tanto del nombre de la asociación como del destino de la misma.
- Para los atributos primitivos y tuplas se han creado unos iconos rosáceos de pequeño tamaño y etiquetas que denotan el tipo o tipos primitivos a los que se hace referencia, y el nombre de los mismos.

La escala cromática y las etiquetas se han conservado en todas las vistas desarrolladas para facilitar la uniformidad de las mismas y el proceso de aprendizaje de la interfaz al usuario.

Una vez presentada la vista se va a describir cómo se ha implementado en Sirius. La visualización del árbol de *versiones de esquemas* desarrollada puede observarse en la figura 6.5, y a continuación se entra en detalle de la misma.

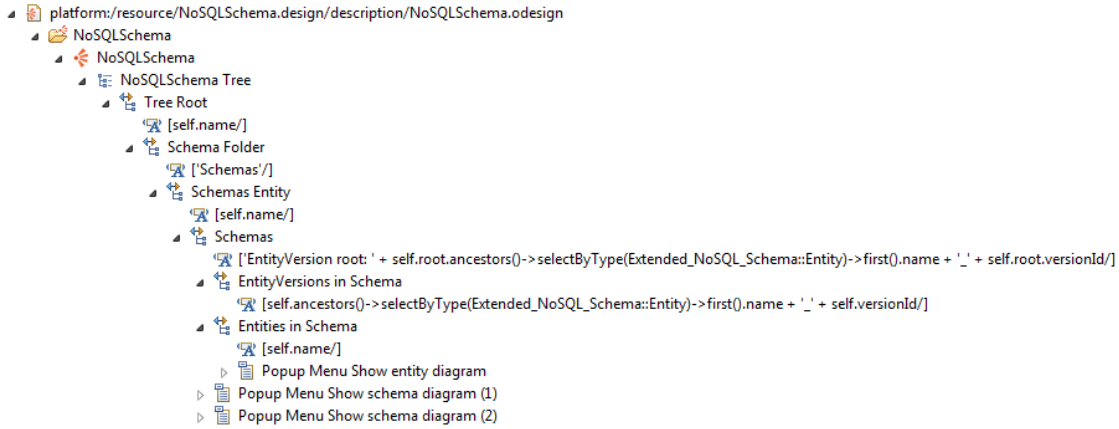


Figura 6.5: Parte del fichero *odesign* para definir visualmente *versiones de esquemas*.

Se define el primer nodo de representaciones, esto es, un nodo que recoja todas las versiones de esquemas de cualquier modelo *Extended_NoSQL_Schema* que se desee visualizar: El nodo con el texto *Schemas*. Para ello se ha creado un nuevo *Tree Element* con nombre *Schema Folder* y con las siguientes características:

```
Id: T_Schemas_Folder
Label: Schema Folder
Domain_Class: Extended_NoSQL_Schema.NoSQLDataModel
Semantic_Candidates_Expression: [NoSQLDataModel.allInstances()/]
Precondition_Expression: [self.schemaVersions->size() > 0/]
```

Con estas expresiones se consigue visualizar un único nodo *Schema Folder*, porque la expresión `NoSQLDataModel.allInstances()` únicamente va a evaluarse con un único candidato: El nodo raíz del que parte el esquema. Además de esto se añade una precondición para representar este nodo: *'que exista al menos una versión de esquema'*.

En el interior de este nodo *contenedor* se representarán todas las *versiones de esquemas* del modelo a visualizar. Como se ha dicho para facilitar la visualización se listarán *entidades* en un primer nivel y en el siguiente nivel se listarán las *versiones raíz*:

```
Id: T_Schemas_Entity
Label: Schemas Entity
Domain_Class: Extended_NoSQL_Schema.Entity
Semantic_Candidates_Expression: [self.schemaVersions.root.ancestors()/]
```

La expresión semántica para estos nodos es la siguiente: Para cada *versión de esquema* se debe navegar hasta la *versión raíz* y, de aquí, al ancestro que la contiene, que será único, y de tipo *Extended_NoSQL_Schema.Entity*. La representación de estos nodos se compondrá del nombre de la *entidad* y un icono identificador para *Extended_NoSQL_Schema.Entity*.

El siguiente nodo a mostrar está anidado con respecto al nodo anterior, y la navegabilidad se complica: No es posible navegar, de forma sencilla, desde una *entidad* hacia una *versión de esquema*, por lo que se requiere la siguiente expresión *Acceleo3*:

```
Id: T_Schemas_Schema
Label: Schemas
Domain_Class: Extended_NoSQL_Schema.SchemaVersion
Semantic_Candidates_Expression: [SchemaVersion.allInstances()
  ->select(schema : SchemaVersion | schema.root.ancestors()->count(self) = 1)
  ->sortedBy(root.versionId)/]
```

La expresión semántica indica que se deben seleccionar todas las versiones de esquema tales que el contenedor de la *versión raíz* sea la entidad inicial (*self*). Para terminar este apartado, se representarán dentro de cada *versión de esquema* las entidades y versiones que forman parte de la misma. Para las entidades:

```
Id: T_Schemas_Schema_Entity
Label: Entities in Schema
Domain_Class: Extended_NoSQL_Schema.Entity
Semantic_Candidates_Expression: [self.entities/]
```

Para las versiones que no son raíces se navegará, desde *SchemaVersion*, por la relación *entityVersions* ordenada por el atributo *versionId*.

```
Id: T_Schemas_Schema_EntityVersion
Label: EntityVersions in Schema
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression: [self.entityVersions->sortedBy(versionId)/]
```

6.3.1.2 Árbol invertido de versiones de esquema

Con la vista anterior el usuario puede buscar *versiones de esquemas* a partir de las *versiones raíz* que definen dichos esquemas. Sin embargo, otro tipo de búsqueda que podría querer realizar el usuario en otras ocasiones sería que *'dada una versión cualquiera*

(raíz o no), obtener todas las versiones de esquemas *que involucran a esta versión'*. Este es el propósito del índice inverso: Un listado de versiones a las que se anexan las *versiones de esquemas* de interés para esta versión.

Figura 6.6: Visualización del índice inverso sobre *Books.extended_nosql_schema*.

Como se observa en la figura 6.6 lo que se desea obtener ahora es una carpeta contenedora con título *Inverted Index* a la que anexar todas las versiones de las que disponga el modelo de entrada, con su característico icono de versión y con etiquetas similares a las de la vista anterior, y anidadas en cada versión las *versiones de esquema* de las que esta versión es parte. A su vez estas *versiones de esquema* aparecerán identificadas con su icono de raíz y en la etiqueta mostrarán qué versión es la raíz de la misma, de forma que sea fácilmente deducible si la versión cuyo nodo se está desplegando es la *versión raíz* de una *versión de esquema* o si no lo es.

Por otro lado y como se ve en la figura 6.7 implementar esta parte del árbol de visualización en Sirius es más sencillo que implementar la vista anterior, ya que esta parte del árbol contiene menos elementos anidados y presenta menos niveles de profundidad.

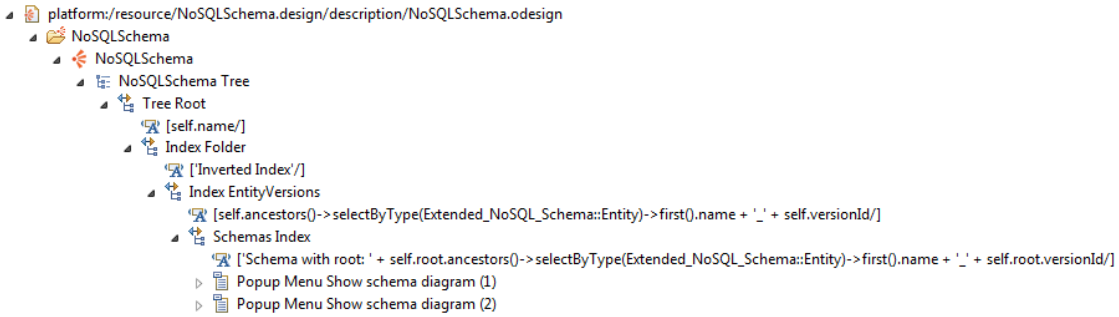


Figura 6.7: Parte del fichero *odesign* para definir el índice inverso.

En primer lugar se desea listar todas las versiones existentes para un modelo cualquiera. Esto se puede hacer con las siguientes propiedades:

```
Id: T_Index_EntityVersions
Label: Index EntityVersions
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression:
[self.entities.entityversions->flatten()->sortedBy(versionId)/]
```

Dado que el contexto se ha situado, de nuevo, en el elemento raíz del metamodelo, para navegar hasta las versiones del mismo se debe, primero, navegar hasta las entidades, y de ahí hasta las versiones, y ordenar la lista resultante. La etiqueta de estas versiones está compuesta por el nombre de la entidad contenedora y el número identificador de la versión en cuestión. Ahora, para cada versión, se pueden obtener las *versiones de esquema* relevantes de la siguiente forma:

```
Id: T_Index_Schemas
Label: Schemas Index
Domain_Class: Extended_NoSQL_Schema.SchemaVersion
Semantic_Candidates_Expression: [SchemaVersion.allInstances()
  ->select(schema: SchemaVersion | (schema.entityVersions->count(self) = 1)
    or (schema.root = self))->sortedBy(root.versionId)/]
```

Para cada versión se deberán buscar aquellas *versiones de esquema* que contengan a dicha versión o bien como raíz, o bien en su lista de versiones incluidas. La etiqueta para los resultados se compondrá del nombre de la entidad contenedora de la *versión raíz* y el número de esta *versión raíz*. Se debe recordar que una versión del índice inverso puede tener asociados distintas *versiones de esquema*, y la versión puede ser raíz de una o ninguna de ellas.

6.3.1.3 Árbol de versiones de entidad

Por último se ha decidido agregar una sección más a este árbol de visualización que liste todas las entidades del modelo de entrada, las versiones de estas entidades, y los atributos y relaciones de estas versiones.

El resultado de esta vista se puede ver en la figura 6.8, y aunque no utiliza ninguna información relacionada con *versiones de esquema*, permite observar de un vistazo la estructura del modelo de entrada. Para ello vuelve a hacerse uso de los iconos y etiquetas descritos anteriormente para cada elemento.

Dada la gran cantidad de elementos a mostrar y los distintos niveles de anidamiento existentes, la estructura a implementar en Sirius es algo más extensa de lo habitual como puede apreciarse en la figura 6.9. Sin embargo aquí se dará una idea general de cómo se ha llevado a cabo dado que esta vista aunque extensa es bastante simple, no presenta consultas semánticas complejas ni problemas de navegación o expresiones que merezcan una explicación más detallada.

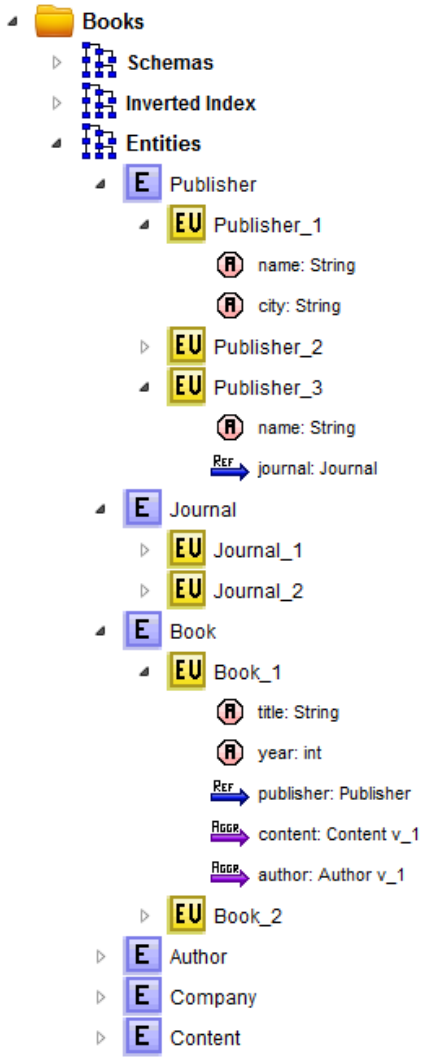


Figura 6.8: Vista de entidades y versiones sobre *Books.extended_nosql_schema*.

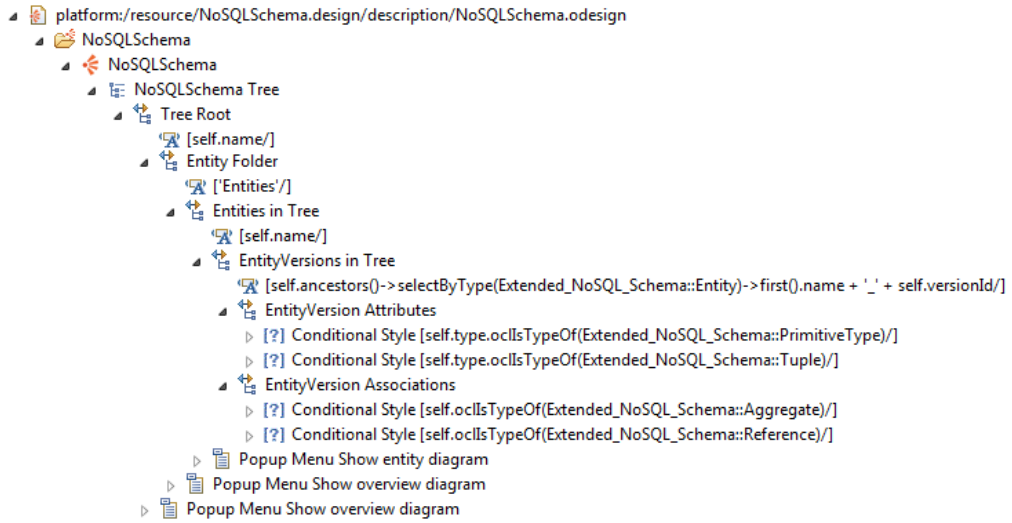


Figura 6.9: Parte del fichero *odesign* para definir el listado de entidades y versiones.

A partir de la raíz del metamodelo se pueden listar las entidades existentes de la siguiente forma:

```
Id: T_Entity_Entities
Label: Entities in Tree
Domain_Class: Extended_NoSQL_Schema.Entity
Semantic_Candidates_Expression: [self.entities/]
```

Y las versiones de la siguiente forma:

```
Id: T_Entity_Entities_EntityVersions
Label: EntityVersions in Tree
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression: [self.entityversions->sortedBy(versionId)/]
```

A partir de este punto se pueden listar las propiedades de cada versión de distintas formas. En nuestro caso se ha optado por agrupar las propiedades en dos grupos: Los atributos y las asociaciones. Para listar los atributos:

```
Id: T_Entity_EntityVersions_Attributes
Label: EntityVersion Attributes
Domain_Class: Extended_NoSQL_Schema.Attribute
Semantic_Candidates_Expression: [self.properties/]
```

Nótese que se está indicando que se van a representar *attributes*, aunque la expresión semántica `[self.properties/]` incluiría también las asociaciones. No hay problema con esto. Sirius tomará todos los elementos *propiedades* de la versión y únicamente se quedará con aquellos del tipo *Extended_NoSQL_Schema.Attributes*, por lo que no es necesario filtrar elementos manualmente. Ahora, para distinguir tipos simples de tuplas se ha optado por aplicar *filtros de estilo*:

```
Conditional_style:
[self.type.oclIsTypeOf(Extended_NoSQL_Schema::PrimitiveType)/]
    Label: [self.name + ': ' +
self.type->selectByType(Extended_NoSQL_Schema::PrimitiveType).name/]
Conditional_style: [self.type.oclIsTypeOf(Extended_NoSQL_Schema::Tuple)/]
    Label: [self.name + ': Tuple [' +
self.type.oclAsType(Extended_NoSQL_Schema::Tuple).elements
    ->selectByType(Extended_NoSQL_Schema::PrimitiveType)
    ->collect(elem : Extended_NoSQL_Schema::PrimitiveType | elem.name + '
') +
']'/]
```

Los filtros de estilo aplican un estilo si el elemento en cuestión cumple con una condición. En este caso esta condición será preguntar por el tipo del elemento, de forma que se aplicará un estilo si el elemento es *PrimitiveType* y otro estilo si el elemento es de tipo *Tuple*. En el caso de que el elemento sea *Tuple* se forma una expresión *Acceleo3* para etiquetar a este elemento con el tipo de todos los elementos que componen la tupla.

La representación de *Associations* para cada versión sigue el mismo esquema:

```
Id: T_Entity_EntityVersions_Associations
Label: EntityVersion Associations
Domain_Class: Extended_NoSQL_Schema.Association
Semantic_Candidates_Expression: [self.properties/]

Conditional_style: [self.oclIsTypeOf(Extended_NoSQL_Schema::Aggregate)/]
    Label: [self.name + ': ' + self.refTo.ancestors()
      ->first().oclAsType(Extended_NoSQL_Schema::Entity).name + ' v_' +
self.refTo.versionId/]
Conditional_style: [self.oclIsTypeOf(Extended_NoSQL_Schema::Reference)/]
    Label: [self.name + ': ' + self.refTo.name/]
```

6.3.2 Diagrama del esquema global

El diagrama del esquema global recoge la información listada por el último apartado de la vista en árbol de versiones de entidad. La representación esta vez se va a llevar a cabo en forma de diagrama, de una manera similar a los clásicos diagramas de clases **UML** [14]. Para ello, se van crear diagramas Sirius en vez de árboles.

Como se observa en la figura 6.10 un diagrama de esquema global de un modelo de entrada corresponde a una visualización general de dicho modelo y todos sus elementos. En el mismo no intervienen *versiones de esquemas* ni nada parecido, ya que ese no es el propósito de la vista, pero permite averiguar qué entidades incluye un esquema, cuántas versiones existen de cada una de estas entidades, qué atributos tiene cada versión y cómo se relacionan las versiones a través de las relaciones de referencia y agregación.

Para el diseño de esta vista se ha recurrido a una escala cromática familiar para el usuario extraída a partir de los iconos de la vista de árbol anteriormente descrita. En lugar de recurrir a iconos se ha optado por formas geométricas rectangulares de bordes redondeados con un color, una etiqueta y un contenido:

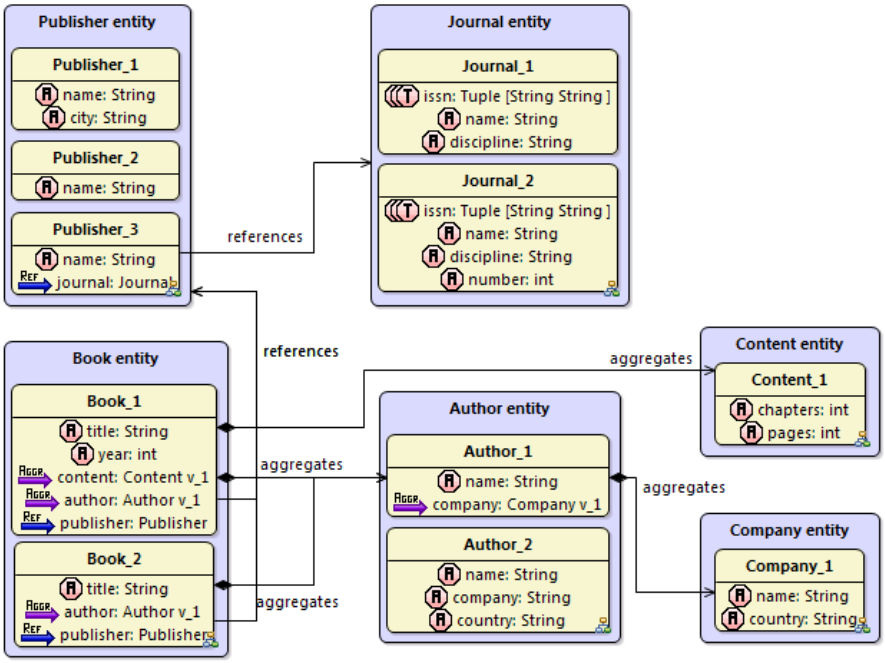


Figura 6.10: Visualización del esquema global de *Books.extended_nosql_schema*.

- Las entidades (*Entities*) se representan con una tonalidad violeta pálida y tienen como etiqueta el nombre de la entidad. En su interior se muestran todas las versiones que pertenecen a dicha entidad. Estas versiones pueden ser reordenadas o redimensionadas como el usuario desee.

- Las versiones (*EntityVersions*) se representan con una tonalidad amarillenta pálida, y tienen como etiqueta el nombre de la entidad seguido del número de versión que representan. En su interior se muestran, en forma de lista, las propiedades de cada versión.

- Los *tipos primitivos* y las *tuplas* se representan con un icono rosado en el interior de la versión a la que pertenecen. Como etiqueta estos tipos muestran el nombre de la propiedad y el tipo de la misma, o tipos en caso de tupla.

- Las *asociaciones* y *referencias* se representan de dos formas distintas:

  - Como una propiedad más en el interior de la versión de la que forma parte, con un icono característico e indicando en la etiqueta el nombre de la asociación y el nombre del destino de la misma.

  - Como una línea que conecta la versión que posee la asociación con el elemento destino. Esta línea es diferente en función del tipo de asociación que represente.

La figura 6.11 muestra la parte del fichero de diseño que define esta vista, y cuya implementación en Sirius se procederá a explicar a continuación.

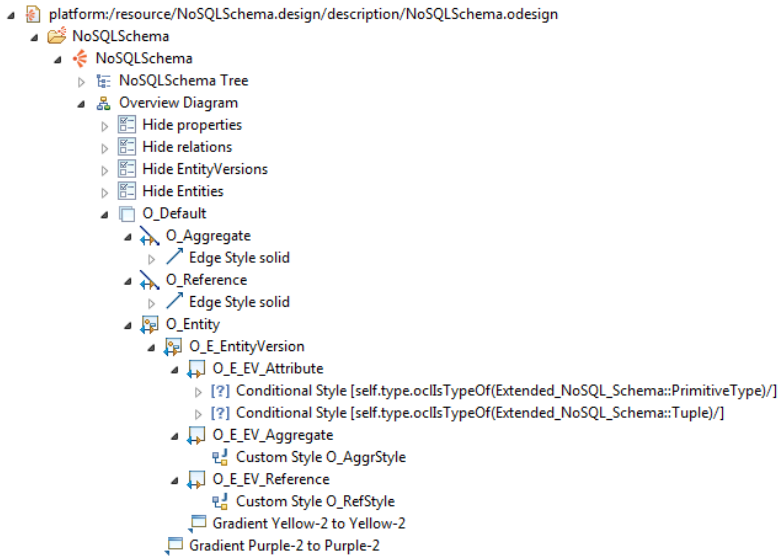


Figura 6.11: Parte del fichero *odesign* para definir la vista global.

En primer lugar al definir una vista de diagrama se debe especificar, como en el caso anterior, cuál es el elemento raíz de la misma y, de nuevo, este elemento es *Extended_NoSQL_Schema.NoSQLDataModel*. A partir de esta raíz se representan entidades:

```
Id: O_Entity
Label: O_Entity
Domain_Class: Extended_NoSQL_Schema.Entity
Semantic_Candidates_Expression: [self.entities/]
```

A continuación se definen de forma anidada en este contenedor las versiones de cada entidad:

```
Id: O_E_EntityVersion
Label: O_E_EntityVersion
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression: [self.entityversions/]
```

Cada versión se ha representado, de nuevo, como un contenedor para poder mostrar en su interior sus propiedades. Estas propiedades aparecerán en forma de lista, y se diferencian en tres ramas:

- *Attributes*: Con unos estilos condicionales para distinguir entre *PrimitiveTypes* y *Tuples*. En función de estos estilos condicionales se escogerá un icono de representación u otra, y la etiqueta contendrá información distinta.

```
Id: O_E_EV_Attribute
Label: O_E_EV_Attribute
Domain_Class: Extended_NoSQL_Schema.Attribute
Semantic_Candidates_Expression: [self.properties/]

[self.type.oclIsTypeOf(Extended_NoSQL_Schema::PrimitiveType)/]
<Estilo de PrimitiveTypes>
[self.type.oclIsTypeOf(Extended_NoSQL_Schema::Tuple)/]
<Estilo de Tuples>
```

- *References*: Se mostrará el nombre de la referencia y el nombre de la entidad referenciada.

```
Id: O_E_EV_Reference
Label: O_E_EV_Reference
Domain_Class: Extended_NoSQL_Schema.Reference
Semantic_Candidates_Expression: [self.properties/]
```

- *Aggregates*: Se mostrará el nombre de la misma y el nombre de la versión agregada.

```
Id: O_E_EV_Aggregate
Label: O_E_EV_Aggregate
Domain_Class: Extended_NoSQL_Schema.Aggregate
Semantic_Candidates_Expression: [self.properties/]
```

Además de especificar como propiedades de las versiones las referencias y las agregaciones, se ha definido una conexión visual entre los elementos relacionados. Para ello, a nivel de elemento raíz del diagrama se han definido dos elementos visuales más:

- Para *Reference*:

```
Id: O_Reference
Label: O_Reference
Source_Mapping: O_E_EntityVersion
Target_Mapping: O_Entity
Target_Finder_Expression: [self.properties
  ->selectByType(Extended_NoSQL_Schema::Reference).refTo/]
```

- Para *Aggregate*:

```
Id: O_Aggregate
Label: O_Aggregate
Source_Mapping: O_E_EntityVersion
Target_Mapping: O_E_EntityVersion
Target_Finder_Expression: [self.properties
  ->selectByType(Extended_NoSQL_Schema::Aggregate).refTo/]
```

Se observa cómo para representar conexiones es necesario definir otras propiedades:

- *Source_Mapping*: El identificador de la clase que actúa como origen de la conexión. En ambos casos, el identificador en la representación de las versiones.

- *Target_Mapping*: El identificador de la clase que actúa como destino de la conexión. En el caso de representar una referencia se trata de *Entity*. En el caso de una agregación se trata de *EntityVersion*.
- *Target_Finder_Expression*: A partir de la clase indicada por *Source_Mapping*,expresión para obtener la clase destino (*Target_Mapping*). Desde la versión se debe navegar a toda propiedad que sea del tipo indicado (*Aggregate* o *Reference*) y, de ahí, al destino mediante la relación *refTo*.

Como resultado se obtiene una serie de entidades con versiones en su interior, con listados de propiedades y relaciones visuales entre entidades y versiones.

6.3.3 Diagrama de versión de esquema

Retomando el concepto de *versión de esquema*, a continuación se proponen dos variantes de una vista utilizada para describir, en forma de diagrama, una *versión de esquema* y todos los elementos relacionados con la misma: *versión raíz*, entidades y versiones. Muchos de los conceptos y expresiones utilizados en estas vistas son idénticos a los utilizados en la vista general, por lo que aquí no se comentarán en detalle.

Se proponen además dos variantes de representación: *NoSQLSchema Diagram (1)* (o diagrama de *versión de esquema* plano) y *NoSQLSchema Diagram (2)* (o diagrama de *versión de esquema* anidado). Las dos representaciones difieren en cómo se representan las entidades y versiones directamente relacionados con la *versión raíz* del esquema. En la primera representación estas relaciones se muestran visualmente, de una forma similar a como se mostraba la vista global del modelo. En la segunda representación, en cambio, las entidades y versiones directamente relacionadas con la *versión raíz* se muestran anidadas en el interior de esta.

6.3.3.1 Diagrama de versión de esquema plano

Para el diseño de este primer diagrama de *versiones de esquema* se ha optado por reutilizar muchas decisiones de diseño del diagrama de esquema global, como se puede observar en la figura 6.12. En concreto el diseño de este diagrama se ha realizado atendiendo a los siguientes puntos:

- Se ha conservado la misma representación para entidades que anteriormente, utilizando una paleta violeta pálida y formando la etiqueta de cada entidad con el nombre de dicha entidad.
- Las versiones ahora pueden mostrarse de dos formas: Como objetos aislados o anidadas en el interior de su entidad. Si una versión a representar es de una entidad que también forma parte de la *versión de esquema*, entonces esta versión se mostrará en el interior de su versión. En caso contrario, si la versión forma parte de la *versión de esquema* pero su entidad no, entonces la versión se mostrará como un objeto aislado.

- Los atributos, tanto propiedades como asociaciones, se muestran como en la vista anterior: Con una etiqueta descriptiva y un icono apropiado. Las asociaciones, además, vuelven a representarse como conectores entre elementos.
- Además de estos elementos también se representa la *versión raíz* de la *versión de esquema*, con un tono amarillento algo más fuerte que el de una versión normal, para indicar que se trata de una *versión* distinta a las demás, con la etiqueta resaltada en negrita y un icono de *raíz de versión de esquema*.

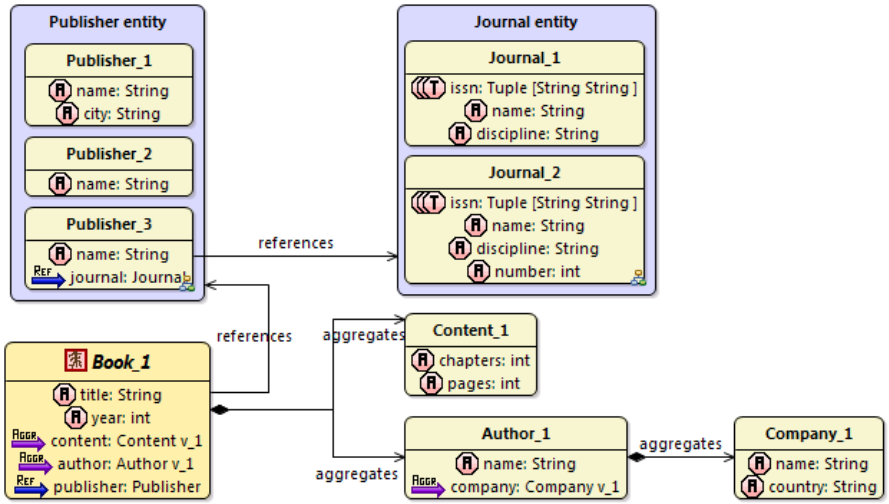


Figura 6.12: Un diagrama de versión de esquema plano de *Books.extended_nosql_schema*.

Para implementar la vista en Sirius se definen en el nivel raíz tres clases. Estas tres clases surgen a partir de la clase raíz de la vista, que en esta ocasión es *Extended_NoSQL_Schema.SchemaVersion*. La primera clase definida es la *versión raíz*:

```
Id: S1_EntityVersionRoot
Label: S1_EntityVersionRoot
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression: [self.root/]
```

Los componentes internos de esta versión (propiedades divididas en *Attributes*, *References* y *Aggregates*) se representan igual que en el apartado anterior. En adelante y mientras no se diga lo contrario, estos componentes internos se implementarán de la misma forma, por lo que se obviará su definición a partir de ahora.

La siguiente clase definida es la de entidades asociadas a la *versión de esquema*. Como se apuntó, estas entidades formaban parte de una *versión de esquema* si eran referenciadas por alguna versión de la misma:

```
Id: S1_Entity
Label: S1_Entity
Domain_Class: Extended_NoSQL_Schema.Entity
Semantic_Candidates_Expression: [self.entities/]
```

Y a partir de estas entidades, definidos en el interior sus versiones:

```
Id: S1_E_EntityVersion
Label: S1_E_EntityVersion
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression: [self.entityversions/]
```

Por último, para cada versión de estas entidades se definen sus propiedades internas de la misma forma que siempre. La última clase definida a nivel raíz es la de versiones asociadas a la *versión de esquema*. Una versión pertenece a una *versión de esquema* si esta es agregada por alguna otra versión, o bien su entidad contenedora es referenciada:

```
Id: S1_EntityVersion
Label: S1_EntityVersion
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression: [self.entityVersions
  ->removeAll(self.entities->flatten().entityversions)/]
```

Algo notable en esta definición es que para escoger candidatos semánticos a representar primero se toma todo el conjunto de versiones incluidas en el esquema pero posteriormente se eliminan aquellas que están contenidas en una entidad agregada a la *versión de esquema*. Esto se debe a que como se ha observado una versión puede formar parte de una *versión de esquema* si pertenece a una entidad referenciada (en cuyo caso, la versión se representa en el interior de dicha entidad) o si es agregada por otra versión, en cuyo caso se muestra como indica esta expresión estudiada.

Por último, se desea representar como conexiones visuales las clases *Aggregate* y *Reference*. Sin embargo, no se puede hacer uso de las expresiones anteriormente usadas para ello porque se dispone de distintos tipos de versiones de las que pueden surgir las asociaciones: Versiones en el interior de entidades, versiones a nivel raíz y la versión raíz. Estas versiones no tienen asignado el mismo identificador (*Id*) en la vista, y por ello en los atributos *Source_Mapping* y *Target_Mapping* se deben indicar todos los identificadores de versiones de las que pueden surgir o en las que pueden terminar asociaciones:

- Para *Reference*:

```
Id: S1_Reference
Label: S1_Reference
Source_Mapping: S1_EntityVersion, S1_E_EntityVersion, S1_EntityVersionRoot
Target_Mapping: S1_Entity
Target_Finder_Expression: [self.properties
  ->selectByType(Extended_NoSQL_Schema::Reference).refTo/]
```

- Para *Aggregate*:

```
Id: S1_Aggregate
Label: S1_Aggregate
Source_Mapping: S1_EntityVersion, S1_E_EntityVersion, S1_EntityVersionRoot
Target_Mapping: S1_EntityVersion, S1_E_EntityVersion, S1_EntityVersionRoot
Target_Finder_Expression: [self.properties
  ->selectByType(Extended_NoSQL_Schema::Aggregate).refTo/]
```

6.3.3.2 Diagrama de versión de esquema con anidación

Este tipo de diagrama muestra la misma información que el diagrama anterior pero reordenada de forma distinta. Como se puede ver en la figura 6.13, se muestran las entidades y versiones directamente relacionadas con la *versión raíz* de la *versión de esquema*. Mientras que en el caso anterior estos elementos se mostraban distribuidos por la representación en esta ocasión la *versión raíz* dispone de un espacio sobre los que se muestran embebidos. Las entidades y versiones que forman parte de la *versión de esquema* pero no están asociados (por agregación o referencia) con la *versión raíz* se siguen mostrando como en el caso anterior.

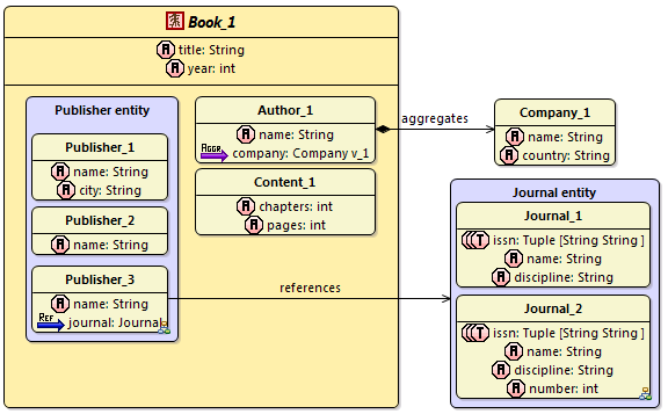


Figura 6.13: Un diagrama de *versión de esquema* con anidación de *Books.extended_nosql_schema*.

Dado que de los tres conceptos a diseñar a nivel raíz, los de *versión raíz*, entidades (referenciadas) y versiones (agregadas o con la entidad padre referenciada), únicamente el primer concepto es notablemente distinto a la primera visualización de *versión de esquema* propuesta. Por ello en esta sección únicamente se comentará el diseño de la *versión raíz* en esta vista. Tanto las entidades, como las versiones, como el resto de elementos se mantienen de forma idéntica al apartado anterior.

En primer lugar se define la *versión raíz* de la misma forma que siempre, pero en su interior se definirán dos apartados visualmente separados. El primer apartado será utilizado para mostrar las propiedades de tipo *Attribute*, y el segundo para mostrar las propiedades de tipo *Association*:

```
Id: S2_EVR_AttributePart
Label: S2_EVR_AttributePart
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression:
[
self->select(self.properties->selectByType(Extended_NoSQL_Schema::Attribu
te)->
size() > 0)/]
```

```
Id: S2_EVR_RefAggrePart
Label: S2_EVR_RefAggrePart
Domain_Class: Extended_NoSQL_Schema.EntityVersion
Semantic_Candidates_Expression: [self/]
```

La mayor ventaja de este diagrama con respecto al otro reside en la diferenciación que se ha detallado anteriormente de entidades y versiones directamente relacionados con la *versión raíz* y entidades y versiones que, aun perteneciendo a la *versión de esquema*, no están relacionados con esta *versión raíz*. Esto ayuda a visualizar de forma más ordenada la *versión de esquema*, así como a facilitar la tarea del usuario cuando este quiera encontrar una entidad o versión concreta sobre el diagrama.

6.3.4 Diagrama de entidad

La última vista diseñada es utilizada para visualizar la información sobre un determinado tipo de entidad (sus versiones, las propiedades de cada versión y sus relaciones) de forma aislada con respecto al resto del esquema del modelo de entrada. Con definiciones similares a las ya vistas es posible crear una vista que muestre, para una entidad, sus versiones (con las propiedades de estas) y las entidades y/o versiones relacionadas, de alguna forma, con esta entidad. Este diagrama de entidad tiene el aspecto mostrado en la figura 6.14 y las pautas de diseño seguidas son las siguientes:

- Se ha definido la representación para la *Entity* principal de la vista. Esta entidad se debe diferenciar del resto de entidades mostradas y por ello se ha empleado un tono visual púrpura ligeramente más oscuro del utilizado normalmente para entidades.
    - En el interior de esta entidad se representan las versiones contenidas, con sus propiedades (*Attributes* y *Associations*) en forma de listado, con el estilo habitual.
- A nivel raíz se ha definido la representación para una *Entity* que no es la principal de la vista, pero es referenciada por alguna versión.
    - En el interior de estas entidades no principales se representan las versiones contenidas deforma muy simplificada, y sin mostrar sus propiedades, en forma de listado.
- De nuevo a nivel raíz se ha definido una representación para versiones (*EntityVersion*) relacionadas de alguna forma con la entidad principal: Bien porque referencian a esta entidad, bien porque son agregadas por alguna versión de la entidad.
- Por último se han definido relaciones visuales para *Aggregate* y *Reference*.

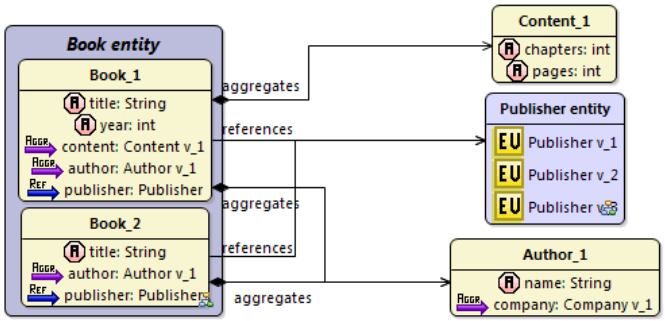


Figura 6.14: Un diagrama de entidad de *Books.extended_nosql_schema*.

Por último la implementación en Sirius se ha llevado a cabo reutilizando contenido de otros diagramas, y se muestra en la figura 6.15. Se puede observar que si bien la implementación es algo extensa las definiciones dadas, las representaciones escogidas y las expresiones semánticas utilizadas son reutilizadas de diagramas anteriores, sin cambiar un ápice de los mismos.

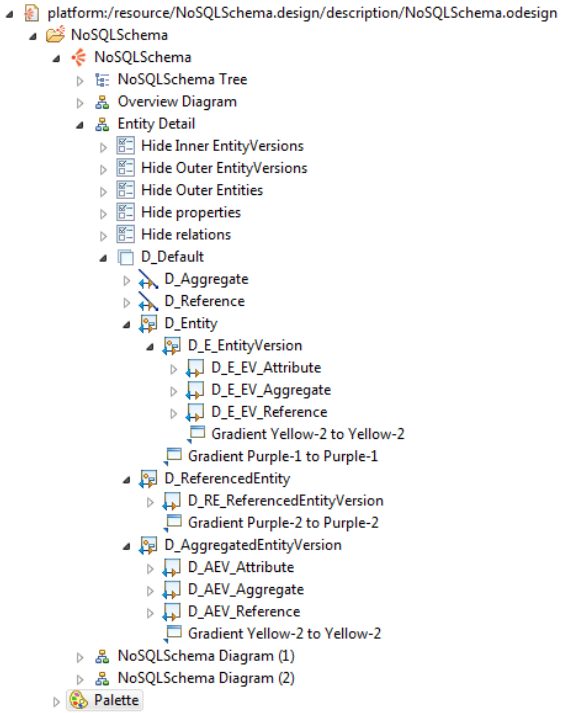


Figura 6.15: Parte del fichero *odesign* para definir la vista de entidad.

6.4 Aplicación al caso de estudio

En este último apartado se ofrecen unas pautas de visualización de un modelo de entrada *Food.nosql_schema*. Este modelo ha sido creado con el editor de *Ecore* como ejemplo para ilustrar el funcionamiento de las vistas diseñadas y se ha concebido como un ejemplo descriptivo que muestre las distintas combinaciones de *versiones de esquemas*, variedad de versiones, atributos de distintos tipos primitivos, algunas tuplas, referencias y agregaciones. Está formado por entidades y versiones del mundo de la cocina:

- Entidad *Restaurant*: Con una única versión, atributos simples y dos referencias a las entidades *Waiter* y *Menu*.
- Entidad *Waiter*: Tres versiones con distintos atributos primitivos (*strings*, *ints*, *floats*) y, en el caso de la última versión, una relación *Aggregate* a la entidad *Table*.
- Entidad *Table*: Dos versiones con atributos simples similares y una tupla de enteros para indicar la posición de la mesa en el restaurante.
- Entidad *Menu*: Una entidad con una única versión muy simple que referencia a la entidad *Dish*.
- Entidad *Dish*: Entidad utilizada para albergar distintas versiones en las que los atributos van cambiando de tipo. Todas estas versiones mantienen una relación *Aggregate* con la entidad *Ingredient*.
- Entidad *Ingredient*: Entidad de la que existen dos versiones muy similares, siendo únicamente la versión 2 la agregada por otras versiones.

La transformación *m2m* descrita se ha ejecutado para este modelo de entrada y se ha obtenido un modelo llamado *Food.extended_nosql_schema*. Este nuevo modelo conforma al metamodelo *Extended_NoSQL_Schema*, de modo que incluye las entidades, versiones de entidades y versiones de esquema.

Tras instalar el *plugin* de visualización y examinar un modelo *extended_nosql_schema* se inicializarán todos los diagramas posibles para este modelo. A continuación se comentarán los distintos diagramas. La figura 6.16 muestra el diagrama de árbol del modelo en el que no solo se visualiza la estructura del modelo (en la rama del árbol titulada *Entities*), sino también la variedad de *versiones de esquemas* que ofrece. Para obtener una vista global de todo el esquema se puede recurrir al diagrama de esquema global. Una vez recreado este diagrama y ordenados sus elementos, como en la figura 6.17, se observa de un único vistazo cómo está compuesto este modelo.

Las vistas se centran en la versión 1 de la entidad *Restaurant*. Desde el árbol (o desde el diagrama de esquema global) y haciendo uso de menús contextuales disponibles se puede acceder a cualquiera de los dos diagramas de *versiones de esquemas* (plano o anidado), como se puede apreciar en las figuras 6.18 y 6.19. En ambas figuras se visualiza la misma

información (aunque de distinta forma) y en ellas se puede observar cómo la *versión de esquema* está compuesta por una *versión raíz* (etiquetada como *Restaurant 1*) y distintas entidades y versiones.

Como se comentó en el diseño e implementación de los diagramas de *versiones de esquemas*, el diagrama plano se asemeja más a la estructura mostrada en el diagrama de esquema global, y sitúa entidades y versiones al mismo nivel (sean referenciadas por la *versión raíz* directamente o no), mientras que el diagrama de *versión de esquema* con anidación permite diferenciar qué entidades y versiones están asociadas con la *versión raíz* directamente.

Por último se puede acceder a los diagramas de entidades de nuevo mediante menús contextuales o desde los iconos que se muestran en la parte inferior derecha de todas las entidades (en cualquier diagrama). La vista se centra en, por ejemplo, la entidad *Restaurant*, como puede apreciarse en la figura 6.20. De esta entidad se muestran sus versiones y las entidades con las que se relacionan, de alguna forma, estas versiones.

Como conclusión, este sencillo caso de estudio ha permitido probar la solución definida para mostrar esquemas de bases de datos NoSQL, así como las diferentes vistas ideadas. El editor creado con Sirius permite visualizar diagramas que describen el modelo obtenido. La navegación entre diagramas garantizada por la propia herramienta y la posibilidad de interactuar con los elementos (ocultación, reordenación y redimensión) de los diagramas aseguran que la visualización se adapte a las necesidades del usuario, y que la experiencia del mismo con la herramienta sea satisfactoria.

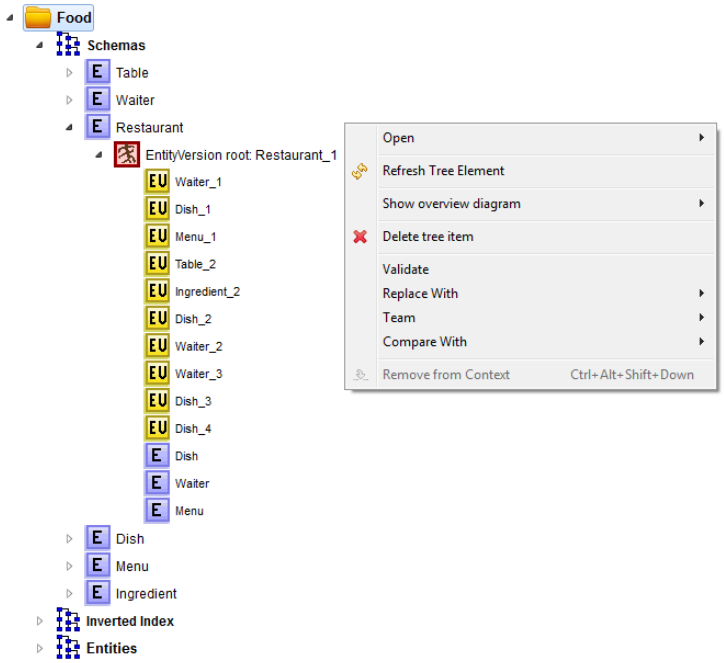


Figura 6.16: Diagrama de vista de árbol para *Food.extended_nosql_schema*.

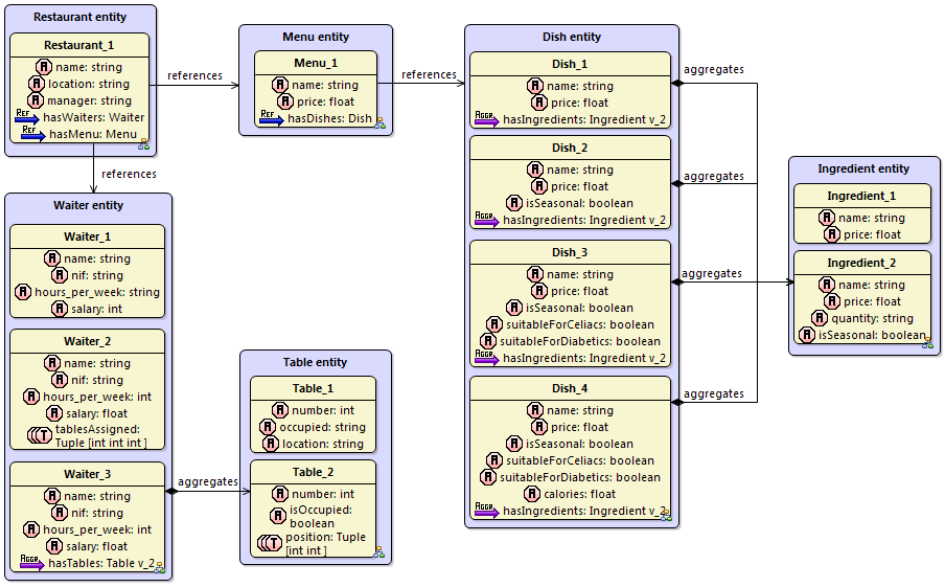


Figura 6.17: Diagrama de esquema global para *Food.extended_nosql_schema*.

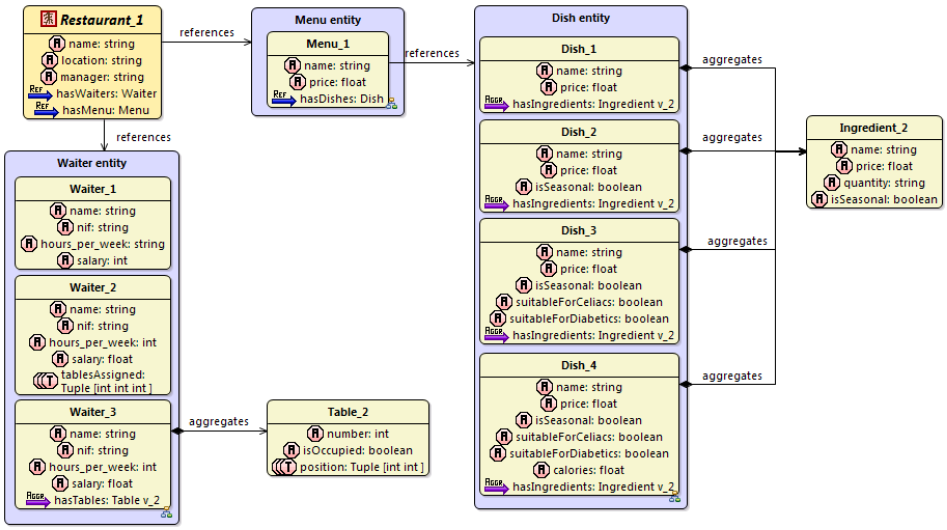


Figura 6.18: Un diagrama de *versión de esquema* plano para *Restaurant 1* de *Food.extended_nosql_schema*.

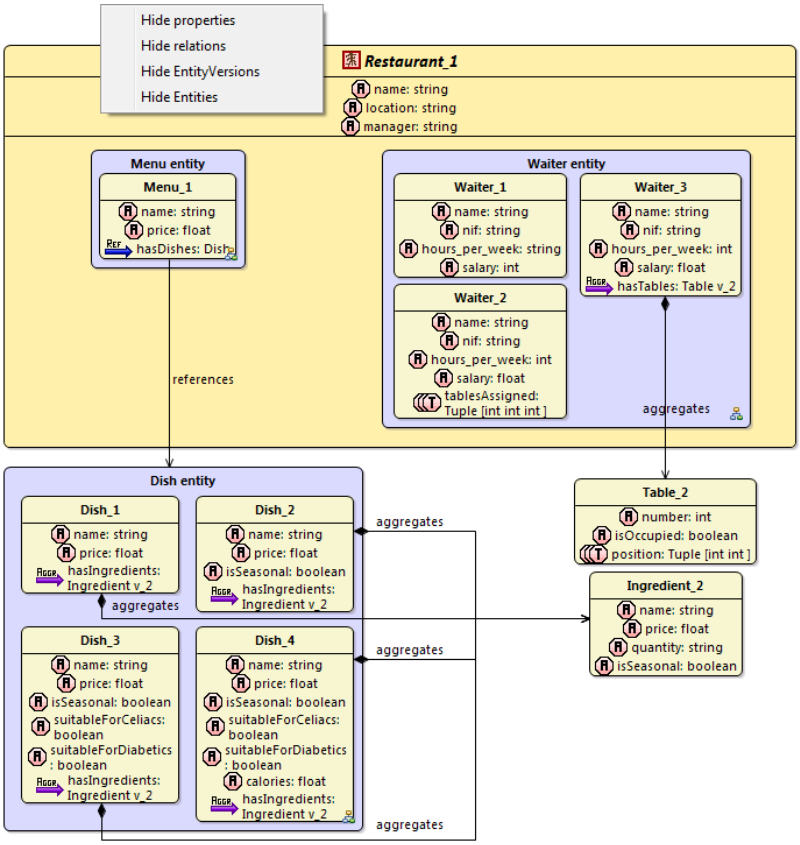


Figura 6.19: Un diagrama de *versión de esquema* con anidación para *Restaurant 1* de *Food.extended_nosql_schema*.

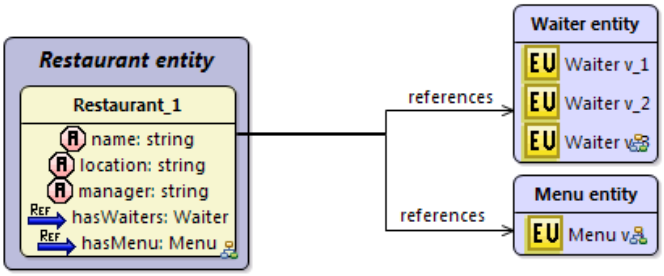


Figura 6.20: Un diagrama de la entidad *Restaurant* de *Food.extended_nosql_schema*.

7 Visualización de datos NoSQL

7.1 Introducción

Este capítulo presenta el proceso de desarrollo de una herramienta de visualización de datos almacenados en una base de datos NoSQL. Mientras en el capítulo anterior se presentó una solución para la visualización de las *versiones de esquemas* inferidas a partir de los datos, ahora se centrará la atención en visualizar propiedades sobre datos que corresponden a versiones de las entidades del esquema.

Antes de visualizar los datos estos deben ser clasificados y ordenados, y para ello se ha definido un nuevo metamodelo que reorganiza la información expresada en un modelo *NoSQL_Schema* de modo que sea más fácil realizar la clasificación de los objetos. Este nuevo metamodelo se ha denominado *metamodelo de diferenciación*, o *Version_Diff*, y una transformación *m2m* implementa la conversión de un modelo *NoSQL_Schema* obtenido en el proceso de inferencia en un modelo del nuevo metamodelo. Entonces, una transformación *m2t* se encarga de generar código en algún lenguaje que permita determinar para un cierto objeto cuál es su entidad y su versión.

Al ejecutar el código generado para una determinada colección de objetos se obtiene una lista de objetos ordenados en una estructura por entidad y versión a la que pertenecen. Esta estructura es fácilmente representable mediante cualquier sistema de visualización que se escoja. En este caso el lenguaje elegido para generar el código de comparación ha sido *Javascript*, y la librería de visualización escogida ha sido *D3.js*.

A continuación se va a describir el *metamodelo de diferenciación*, o *Version_Diff* y las etapas del proceso mostrado en la figura 7.1.

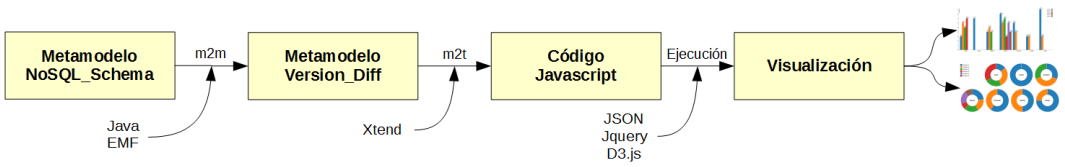


Figura 7.1: Proceso de generación de código de visualización de datos.

7.2 Diseño del metamodelo de diferenciación Version_Diff

Como se ha indicado, con el fin de facilitar la identificación de la versión de una entidad a la que pertenece un objeto de la base de datos, se ha definido el metamodelo *Version_Diff* y se ha implementado una transformación *m2m* que genera un modelo conforme a este metamodelo a partir del modelo *NoSQL_Schema*. El *metamodelo de diferenciación* describe el tipo de las versiones de las entidades que aparecen en el esquema.

Un modelo *Version_Diff*, como puede observarse en la figura 7.2, parte de un objeto *NoSQLDifferences* con el nombre del modelo del que se parte en *NoSQL_Schema*, y a su vez agrega a una serie de objetos *TypeDifference*, uno por cada versión. Estos objetos incluyen a su vez uno o más *TypeHints* que registran información sobre los campos: nombre del campo y una referencia *FieldType* a su tipo. Con todas estas clases es posible definir las particularidades de cada versión a partir de los campos de atributo de esta. Se consideran dos tipos de *TypeHints*:

- *HasField*: Usado para indicar que una versión de una entidad **tiene** un campo de un nombre concreto y un determinado tipo.
- *HasNotField*: Utilizado para indicar que una versión de una entidad **no tiene** un campo de un nombre concreto y un determinado tipo, pero que este campo **existe para otra versión de la misma entidad**.

La idea detrás de estos conceptos será poder identificar cada versión a partir de una serie de campos definidos con el concepto *HasField*, y distinguirla de otras versiones indicando que esta no tiene conceptos que otras sí tienen, mediante *HasNotField*.

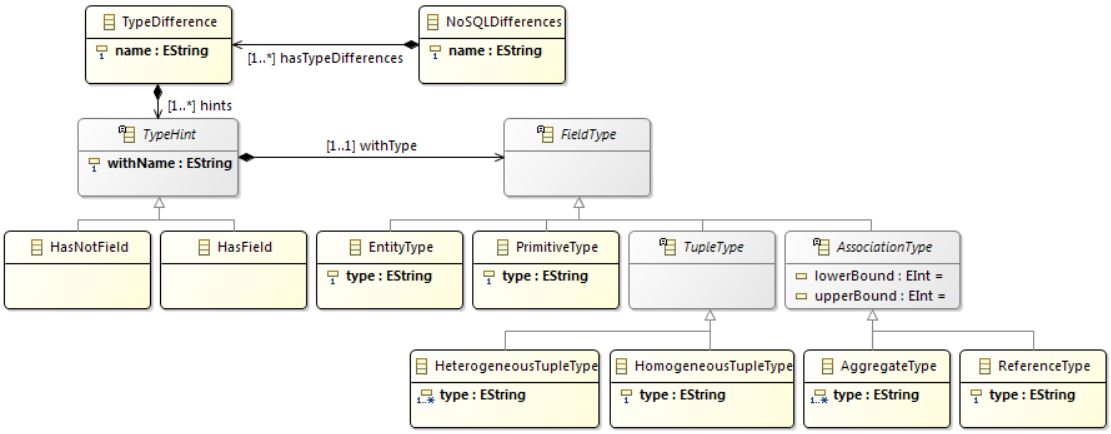


Figura 7.2: Metamodelo *Version_Diff*.

La información que se almacena para describir el tipo de una versión es una lista de los campos que tiene junto a otra de los campos existentes en otras versiones de la misma entidad pero que no tiene, como se puede apreciar en la figura 7.3. Para representar el tipo de un campo se ha definido una jerarquía de clases con todos los posibles tipos a tratar: *Primitivo*, *Tupla* (de varios tipos), *Referencia* y *Agregación*. Esta información se utilizará para analizar el tipo de una versión. A continuación se describen las clases que representan estos diferentes tipos:

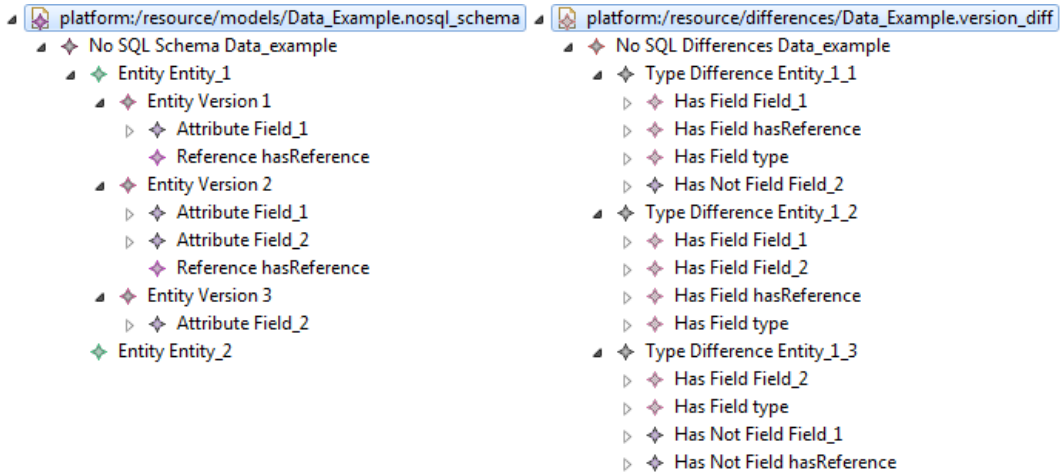


Figura 7.3: Modelo simple *NoSQL_Schema* y su equivalente *Version_Diff*.

- *PrimitiveType*: Utilizada para modelar atributos de tipo simple o primitivo. Estos tipos vienen definidos únicamente por una cadena de texto con el nombre del tipo. Nótese que este concepto de tipo primitivo no está asociado a ningún lenguaje específico, por lo que se tiene la libertad para interpretar cualquier tipo como tipo primitivo, si procediera. Ejemplos básicos de tipos primitivos comunes pueden ser *string*, *float*, *boolean* o *number*.

- *EntityType*: Empleada para identificar de forma específica la entidad a la que pertenece una versión. La motivación detrás de esta idea es que, según la definición de identificación de versiones dada mediante *HasField* y *HasNotField*, no es posible diferenciar dos versiones de entidades distintas si tienen los mismos atributos, y esto representa un problema. *EntityType* consta de un campo *type* para registrar el nombre de la entidad del objeto analizado. Cada versión tendrá asociado un, y solo un, campo de tipo *EntityType* en el *metamodelo de diferenciación*. Si la versión en el modelo de entrada *NoSQL_Schema* ya tenía un atributo con nombre *'type'* o *'Type'*, el valor de este atributo será el valor para el campo *EntityType*. Si la versión no disponía de ningún atributo llamado así se creará uno a tal efecto en la transformación.

- *TupleType*: Una tupla en el *modelo de diferenciación* consiste, como en *NoSQL_Schema*, en una colección ordenada de tipos. Una tupla puede contener

en su interior tanto tipos primitivos como otras tuplas. Para representar tuplas de forma precisa es necesario distinguir dos tipos:

- *HomogeneousTupleType*: Una tupla es homogénea si, y solo si, contiene valores de un único tipo primitivo, como se observa en la tupla homogénea de la figura 7.4. Esto significa que la tupla puede contener uno o más valores del mismo tipo primitivo o tuplas anidadas con valores de ese tipo, o una combinación de ambos. Este tipo se representa mediante una clase del metamodelo que registra el único valor de tipo, de una forma muy similar a como se representa un tipo primitivo.
- *HeterogeneousTupleType*: Una tupla es heterogénea si contiene valores de tipos primitivos distintos, los cuales pueden ser parte de tuplas anidadas, como se puede ver en la tupla heterogénea de la figura 7.4. Este tipo se representa mediante una clase del metamodelo que registra una lista de valores en cadenas de texto. Con esta definición se puede expresar que el valor de una tupla en su posición n coincide con el valor n de la lista de tipos de la clase *HeterogeneousTupleType*.

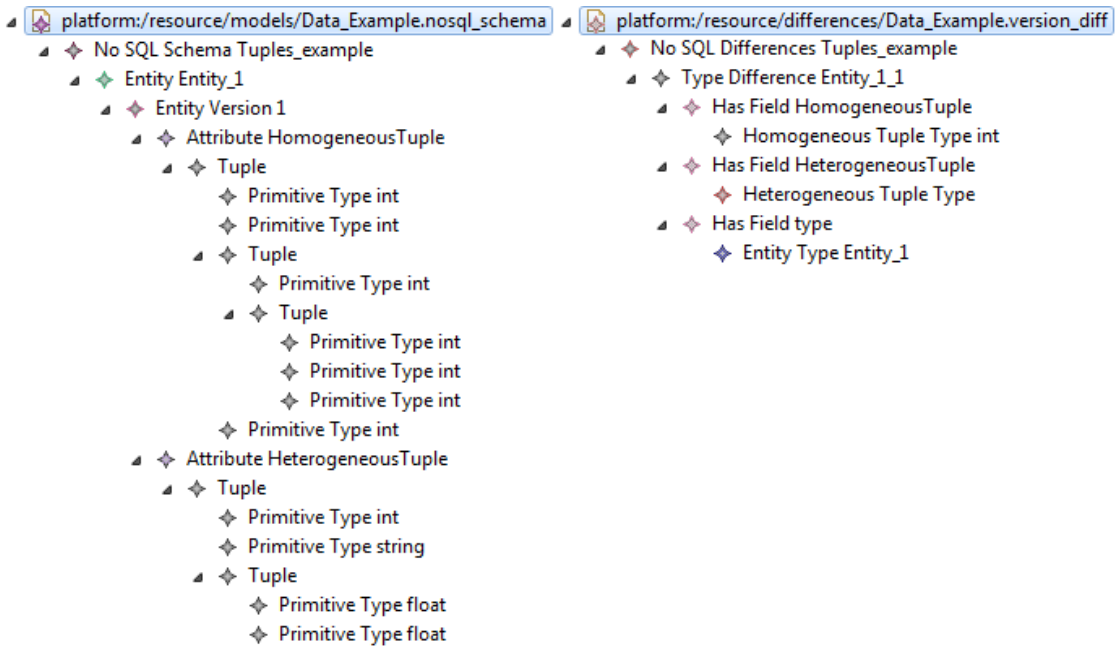


Figura 7.4: Tupla homogénea de *ints* y tupla heterogénea de *int,string,Tuple[float;float]*.

- *AssociationType*: Con este concepto se representa un tipo asociación, corresponde al concepto de asociación del metamodelo *NoSQL_Schema* del que se parte, y lleva asociados dos atributos *lowerBound* y *upperBound* para delimitar la cardinalidad de la asociación siendo transformada. Como ocurría con el metamodelo *NoSQL_Schema*, una asociación puede ser de dos tipos que es conveniente diferenciar con dos clases independientes:

- *ReferenceType*: Clase del metamodelo utilizada para expresar referencias de una versión a una entidad. Esta clase tiene un atributo de tipo *string* para identificar la entidad a la que se está haciendo referencia. Es reseñable el hecho de que, aunque esta clase es casi equivalente a *Reference* en *NoSQL_Schema*, en este caso se prescinde de la relación reflexiva *Opposite* que existía en el modelo de partida, dado que esta relación carece de significado cuando se quieren establecer diferencias entre versiones.

- *AggregateType*: Expresa un tipo de agregación de una versión a otra u otras versiones. Esta clase tiene como atributo una lista de tipos debido a que, como ocurría en *Aggregate* en *NoSQL_Schema*, una versión puede agregar a varias versiones a la vez.

7.2.1 Diferenciación de versiones

Antes de describir la transformación *m2m* entre *NoSQL_Schema* y *Version_Diff* es preciso definir el concepto de equivalencia y diferenciación para versiones y campos: Una versión A es igual a otra versión B si cada campo en A tiene un correspondiente campo en B que es igual (tiene el mismo nombre y tipo). Esto lleva la comparación de dos versiones a la comparación campo a campo de las mismas. Un campo en la versión A es equivalente a un campo en la versión B en función de su tipo, y según el siguiente criterio:

- Dos campos de tipo *PrimitiveType* son iguales si sus nombres son iguales y sus tipos también.

- Dos campos de tipo *EntityType* son iguales si sus nombres son iguales y su tipo es la misma entidad.

- Dos campos de tipo *HomogeneousTupleType* son iguales si sus nombres son iguales y sus tipos denotan el mismo tipo primitivo. Se debe tener en cuenta que dos tuplas homogéneas con el mismo tipo pueden tener distinto número de elementos o tenerlos anidados, y esto no cambia el hecho de que estas dos tuplas son iguales, como se puede observar en la figura 7.5.

- Dos campos de tipo *HeterogeneousTupleType* son iguales si sus nombres son iguales y los tipos primitivos que contienen son iguales uno a uno.

- Dos campos de tipo *ReferenceType* son iguales si sus nombres son iguales y sus tipos denotan la misma referencia, sin importar las cardinalidades inferior y superior.

- Dos campos de tipo *AggregateType* son iguales si sus nombres son iguales y sus tipos denotan la misma agregación, sin importar las cardinalidades inferior y superior.

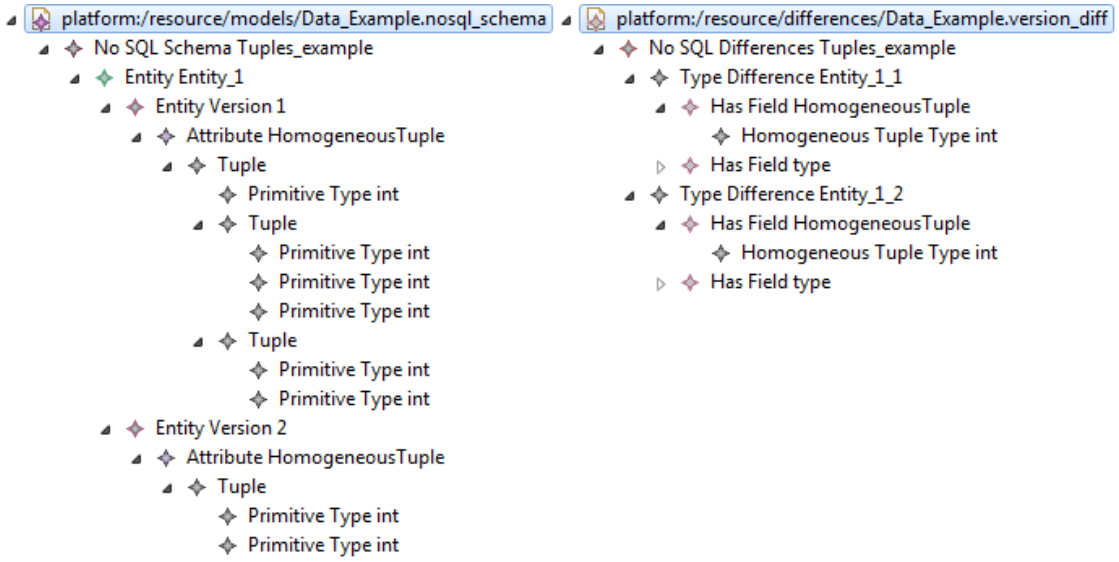


Figura 7.5: Dos tuplas homogéneas equivalentes con distinto número de elementos.

7.2.2 Transformación NoSQL_Schema a Version_Diff

Una vez introducido el metamodelo *Version_Diff* y definidos los conceptos de igualdad de campos y versiones, se va a detallar la transformación que obtiene un *modelo de diferenciación* a partir del modelo *NoSQL_Schema* inferido. El proceso aplicado es el siguiente:

1. Para cada entidad del modelo *NoSQL_Schema* se iterará sobre su conjunto de versiones para construir una tabla de pares <*versión, propiedad*> que registrará las propiedades de cada versión.

2. A continuación para cada versión de cada entidad se creará una clase *TypeDifference* con el nombre de la entidad y la versión que se está construyendo.

3. Más adelante a cada *TypeDifference* creado se le añadirá un objeto *HasField* por cada uno de los campos de la versión, con el correspondiente nombre de campo y el tipo.

4. Finalmente a cada *TypeDifference* también se le añadirán objetos *HasNotField*, uno por cada campo que tiene otra versión de la misma entidad pero no la versión a la que corresponde el *TypeDifference*.

La figura 7.6 muestra a la derecha el modelo *Version_Diff* obtenido para el modelo *NoSQL_Schema* de la izquierda. Nótese que está formado por un objeto *TypeDifference* por cada versión de entidades en el esquema, los cuales tienen objetos *HasField* y *HasNotField* de acuerdo a los campos de la versión.

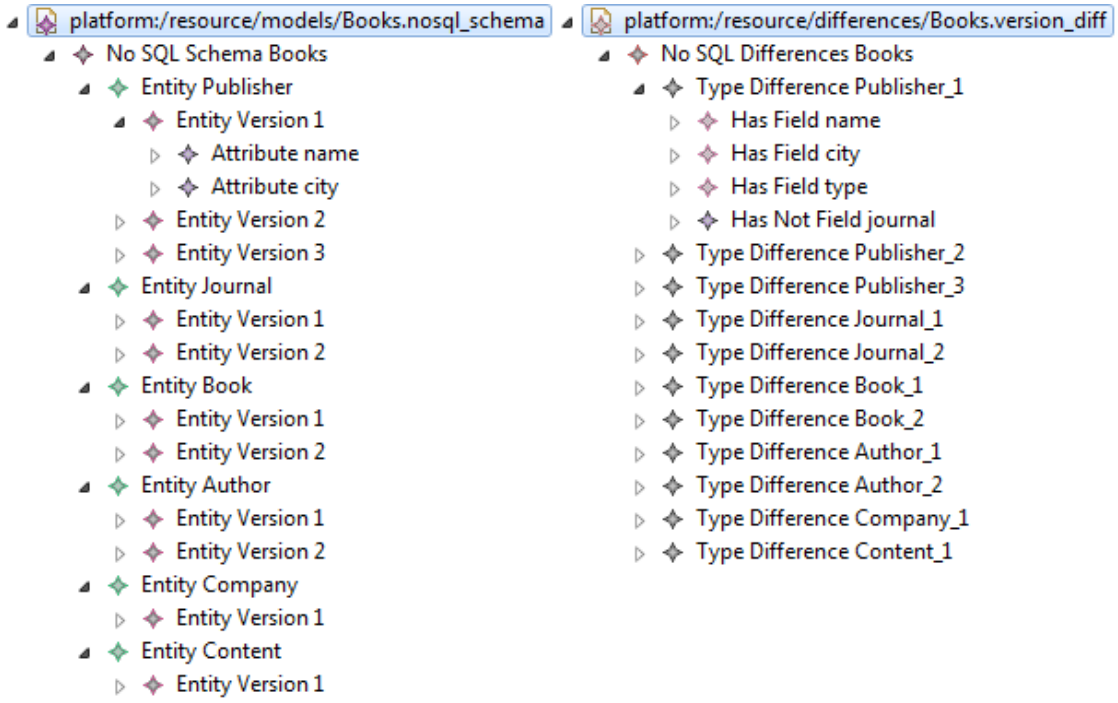


Figura 7.6: Modelo *NoSQL_Schema* y su transformación *Version_Diff*.

El proceso descrito ha sido implementado en Java tras definir el metamodelo *Version_Diff* en **Ecore** y el fichero **genmodel** a partir de él. Este fichero permite a su vez generar toda la API en Java para poder abrir, modificar y guardar *modelos de diferenciación*. El código encargado de realizar esta transformación se puede encontrar en los proyectos Java **NoSQLDataVisualization**, **NoSQLDataVisualization.edit** y **NoSQLDataVisualization.editor** del código fuente del proyecto (véase A.2).

7.3 Generación de código de visualización

En esta sección se describe el proceso de generación de código destinado a la visualización de gráficos sobre la distribución de objetos en las diferentes versiones de la base de datos. Primero se explicará la generación del código *Javascript* encargado de comprobar a qué versión pertenece un objeto dado y a continuación cómo se ha visualizado por medio de la librería *D3.js*.

Para recrear una transformación **m2t** se debe especificar el modelo de entrada conforme a un metamodelo de origen, *Version_Diff* y un fichero o una serie de ficheros de salida

en los que se escribirá el texto producido. El proceso seguido consiste en el recorrido del modelo de entrada desde su elemento inicial hasta sus elementos finales y la generación de código acorde a lo que se procesa. Por tanto un punto crítico de este proceso es el de asegurar que el modelo de entrada garantice navegabilidad hasta todos sus elementos internos con el fin de poder iterar sobre cada objeto existente. La ventaja de aplicar una transformación de estas características es que se asegura la calidad del código generado, su modularidad y la ausencia de errores en el mismo.

7.3.1 Elección del lenguaje de clasificación y visualización

Dado que es posible generar código para cualquier lenguaje se ha tenido libertad para escoger uno que se adaptara de forma satisfactoria a nuestras necesidades. Sabiendo que se va a trabajar con el contenido de bases de datos NoSQL se ha de pensar en el formato de los datos de entrada al proceso de visualización, siendo usualmente **JSON** [3] el formato usado por estas bases de datos para almacenar y/o exportar su contenido.

Aunque en la práctica cualquier lenguaje tiene disponible al menos una librería para manipular JSON, Javascript proporciona de forma nativa muchas facilidades para trabajar con objetos JSON de forma similar a como se manipulan objetos del lenguaje, unido a una gran eficiencia en el manejo de los mismos [44].

Javascript además nos permitirá agrupar en un objeto o un *namespace* un conjunto de funciones de identificación, de forma que será sencillo por medio de una construcción del lenguaje iterar por todas las funciones de clasificación de objetos generadas.

Por último Javascript posee una gran cantidad de librerías gráficas que permiten, una vez se ha realizado la clasificación de los objetos, generar diagramas, árboles y otras formas de visualización. Dada la gran libertad que ofrece la visualización web y la multitud de opciones disponibles Javascript es una elección prometedora como lenguaje de generación de código [36, 1].

Una vez elegido el lenguaje de clasificación de objetos de la base de datos, es preciso decidir cómo se visualizarán los datos y qué tecnología se usará para ello. Para este cometido se ha escogido la librería **D3.js** [8] debido a la gran libertad que ofrece al usuario de mostrar resultados en una amplia gama de formatos y estilos.

D3.js proporciona ventajas en la representación al admitir como formato de entrada a los gráficos JSON u otros similares, como *CSV*. En este caso esto permite transformar una estructura JSON obtenida como resultado de la clasificación de objetos en una estructura JSON compatible con los gráficos *D3.js*. Esta librería además permite una gran flexibilidad al crear diagramas debido a que no es una librería únicamente de representación, sino que se sitúa en un punto más generalista y lo que proporciona son facilidades para modificar el árbol *DOM* de una página web donde crear elementos visuales. Este enfoque además de otorgar flexibilidad elimina restricciones que poseen otras librerías al

estar ceñidas únicamente a representaciones en elementos tales como *canvas* o elementos *SVG* [41, 19].

Un posible punto débil de esta librería puede ser la acusada curva de aprendizaje de la misma, dado que el código resultante no es demasiado intuitivo, pero esta desventaja se puede suplir al contar la herramienta con una gran comunidad de soporte y una inmensa variedad de ejemplos de representaciones.

7.3.2 Estructura del código generado

El código generado ha sido producido de forma modular para que resulte legible, y se ha organizado en los siguientes ficheros:

- Fichero *index.html*: Un fichero de código HTML que cree la estructura de una página web y cargue los distintos ficheros de código Javascript, así como librerías auxiliares. La visualización puede realizarse abriendo este fichero en un navegador web corriente.
- Carpeta *css*: Una carpeta con un fichero *style.css* con normas de estilo para toda la página web y las representaciones visuales diseñadas como pruebas de concepto.
- Carpeta *js* con dos subcarpetas en su interior:
  - Carpeta *js/gen*: Carpeta con tres ficheros de código autogenerado que contienen objetos con métodos encargados de determinar a qué versión pertenece un determinado objeto y una estructura capaz de almacenar qué entidades se identifican, qué versiones existen para cada entidad y cuántas versiones hay de cada tipo.
  - Carpeta *js/lib*: Carpeta con ficheros Javascript que contienen cuatro representaciones a modo de ejemplo en *D3.js* y dos ficheros más de código auxiliar para arrancar la ejecución, llevar a cabo un conteo de versiones del fichero JSON de entrada, rellenar estructuras de datos y, finalmente, crear una representación visual de los datos obtenidos.

Aunque todos los ficheros enumerados anteriormente son necesarios para la correcta ejecución de la visualización de datos, solo algunos de ellos son dependientes del modelo *Version_Diff* dado como entrada. Por ello no todos los ficheros de código Javascript se generarán para cada ejemplo, sino que algunos de ellos estarán pregenerados de antemano en una librería, y se copiarán a la carpeta destino con el código generado dinámicamente.

En el próximo apartado se tratará la creación de los ficheros de código autogenerado, mientras que los ficheros pre-generados se discutirán en la sección 7.4 en la que se analizarán los tipos de representaciones.

7.3.3 Generación de código

Se han implementado dos clases *Xtend* para llevar a cabo la generación del código JavaScript:

- *DifferencesToJSObj.xtend*: Clase que genera, para un modelo *Version_Diff*, hasta dos ficheros de código según se desee llevar a cabo comparaciones inclusivas o comparaciones exclusivas. Este tipo de comparaciones suponen un refinamiento al concepto de comparación de versiones dado anteriormente:

  - Un objeto pertenece de forma inclusiva a una versión si para cada campo de la versión incluye un campo igual. Esto es, solo se tienen en cuenta los elementos *HasField* de un objeto *TypeDifference* pero no los *HasNotField*. El objeto podría tener mas campos que la versión, pero se entiende que puede actuar como un objeto de esa versión. Con esta definición un objeto puede pertenecer, a la vez, a más de una versión.
  - Un objeto pertenece de forma exclusiva a una versión si para cada campo de la versión incluye un campo igual, y ninguno más. Es decir, ahora se tienen en cuenta tanto los elementos *HasField* como los *HasNotField* de *TypeDifference*. Con esta definición un objeto únicamente puede pertenecer, como mucho, a una versión.

  La ejecución de esta transformación *m2t* generará dos archivos Javascript cada uno con un objeto cuyos campos son funciones que contendrán, como campos en su interior, diversas funciones que reciben un objeto Javascript por parámetro, y averiguan si dicho objeto pertenece a una versión en concreto o no. Estas funciones, como se verá más adelante, siguen la convención de comenzar su identificador por la cadena de texto *'isOfExactType_'*.

  El proceso que utiliza la transformación para generar código consiste en recorrer todos los elementos *DifferenceType* del modelo de entrada *Version_Diff* y para cada uno de sus elementos *HasField* y *HasNotField* se generará un código que permita comparar si el tipo *FieldType* asociado está bien formado y tiene valores correctos. Como se verá en la sección posterior, con los ejemplos de uso, en ningún momento se entra a analizar si para un objeto concreto un campo tiene un valor u otro, únicamente que estos valores son del tipo correcto.

- *DifferencesToJSData.xtend*: Esta clase implementa una transformación *m2t* que es algo más simple que la anterior, y generará también dos ficheros de salida.

  - *Fichero Javascript de estructura*: Para facilitar la visualización es necesario almacenar en una estructura apropiada qué entidades existen en el modelo de entrada y qué versiones existen para cada una. El fichero Javascript generado almacena una estructura con distintos campos con nombres de entidades y versiones e inicializados a cero. En tiempo de ejecución esta estructura se rellenará convenientemente y, posteriormente, representada.

- *Fichero index.html*: Es necesario generar un fichero *index.html* para cada modelo de ejemplo debido a que el código contenido en el mismo contiene ciertas instrucciones *'include'* dependientes del nombre de la entrada.

7.3.4 Código generado

Como resultado de ejecutar las transformaciones *m2t* propuestas se generan automáticamente, para cada modelo de entrada, cuatro ficheros de código autogenerado de los cuales se centrará la atención en los ficheros de código de clasificación, cuyos nombres son del tipo *__OBJ__E__<nombre_modelo>.js* y *__OBJ__I__<nombre_modelo>.js*.

El código generado para ambos ficheros sigue un patrón común y únicamente se diferencia en que en el fichero que aplica *comparaciones exclusivas* (*__OBJ__E__*) se ha declarado código para las clases *HasNotField*, cosa que no ocurre para las *comparaciones inclusivas*, donde estas clases son ignoradas. Dado que se entiende que las *comparaciones inclusivas* son un caso estrictamente simplificado de las *comparaciones exclusivas*, de ahora en adelante se tratarán estas últimas para poder ilustrar todos los casos posibles de comprobaciones. El siguiente fragmento de código muestra un ejemplo de código generado en el que los campos son de tipo *string* y *float*. A continuación, se comentará cómo se genera código para los diferentes tipos:

```
var DiffMethodsExclusive =
{
  isOfExactType_Journal_1: function (obj)
  {
    if (!("issn" in obj) || !(obj.issn.constructor === Array)
      || (!checkAllOf(obj.issn, "String")))
        return false;
    if (!("name" in obj) || !(typeof obj.name === "string"))
          return false;
    if (!("discipline" in obj) || !(typeof obj.discipline === "string"))
      return false;
    if (!("type" in obj) || !(typeof obj.type === "string")
      || (obj.type !== "Journal"))
        return false;
    if ("number" in obj && !(!(typeof obj.number === "number")
      || !(obj.number % 1 === 0)))
        return false;

    return true;
  },
  ...
```

Como se puede observar la estructura de código consiste en una variable Javascript de nombre *DiffMethodsExclusive* para el fichero de *comparaciones exclusivas* y de nombre *DiffMethodsInclusive* para el fichero de *comparaciones inclusivas*. Estas variables almacenan en su interior un método de comparación por cada versión identificada en el fichero de entrada capaz de distinguir si un objeto es de dicha versión o no y todos estos métodos siguen la sintaxis de nombrado *isOfExactType_<nombre_entidad_version>*.

Cada uno de estos métodos de clasificación recibe un objeto como parámetro, el objeto cuya versión se desea comprobar, y devuelve un valor *booleano*. Está compuesto por una serie de comprobaciones que siguen los siguientes pasos:

1. Si la comprobación se generó para una clase *HasField*, se comprueba si un campo con cierto nombre **se encuentra** en el objeto.
   - Si este campo no está definido en el objeto, automáticamente el objeto no satisface el chequeo de *HasField* y el resultado retornado es *false*.
   - Si este campo está definido en el objeto se pasa a comprobar el tipo del mismo. Si el tipo observado en el campo del objeto es compatible con el tipo definido en el campo *HasField*, entonces el objeto cumple esta condición. Se siguen comprobando campos.
2. Si la comprobación se generó para una clase *HasNotField*, entonces el objeto **no** debe tener ningún campo con un nombre y un tipo determinados.
   - Si este campo no está definido en el objeto, el objeto satisface el chequeo de *HasNotField* y se continúa comprobando el siguiente campo.
   - Si este campo está definido en el objeto entonces se pasa a comprobar el tipo del mismo. Si el tipo del campo es igual al declarado en *HasNotField*, el objeto no cumple la condición y se retorna un valor *false*.

Para que un objeto se considere de una versión este debe cumplir todas las comprobaciones realizadas en el método. Como se puede observar la lógica general consiste en comprobar si un campo está o no en el objeto buscando su nombre en los campos miembros de dicho objeto y, si lo está, comprobar si el tipo coincide con el tipo declarado por la clase de *Version_Diff*. Se centrará el interés interés, por tanto, en la comprobación de tipos.

7.3.4.1 Tipos EntityType

Para un campo de tipo *EntityType* el tipo se registra como una cadena de texto con el nombre de la entidad, por lo que se debe comprobar que:

1. Existe un campo con el nombre dado. Normalmente este campo se llamará *'type'*.
2. Este campo es de tipo *string*.
3. El valor de este campo es igual al valor del campo *type* de *EntityType*

Si alguna de estas condiciones no se da para un objeto determinado, entonces el objeto no supera la comprobación del método en el que se encuentra y se retorna un valor *false*, determinándose que el objeto no es de la versión indicada. Algunos ejemplos de comprobaciones de este tipo para las entidades de ejemplo *Journal*, *Company* y *Content* pueden verse continuación.

```
var DiffMethodsExclusive =
{
  isOfExactType_Journal_1: function (obj)
  {
    ...
    if (!("type" in obj) || !(typeof obj.type === "string")
      || (obj.type !== "Journal"))
        return false;
    ...
    return true;
  },
  isOfExactType_Company_1: function (obj)
  {
    if (!("type" in obj) || !(typeof obj.type === "string")
      || (obj.type !== "Company"))
        return false;
        ...
    return true;
  },
  isOfExactType_Content_1: function (obj)
  {
    if (!("type" in obj) || !(typeof obj.type === "string")
      || (obj.type !== "Content"))
        return false;
    ...
    return true;
  },
...
```

7.3.4.2 Tipos PrimitiveType

Para un campo de tipo *PrimitiveType* el tipo se registra con una cadena de texto que puede ser *string*, *int*, *float* o *boolean* y las comprobaciones son:

- Se debe comprobar que el campo existe y, además, una comprobación particular en función del tipo.
- *string*: El tipo del campo es *string*.

```
var DiffMethodsExclusive =
{
  isOfExactType_Company_1: function (obj)
  {
    if (!("name" in obj) || !(typeof obj.name === "string"))
      return false;
    ...
    return true;
  },
...
```

- *int*: El tipo del campo es *number* y el número no contiene decimales.

```
var DiffMethodsExclusive =
{
  isOfExactType_Content_1: function (obj)
```

```
  {
    if (!("chapters" in obj) || !(typeof obj.chapters === "number")
      || !(obj.chapters % 1 === 0))
        return false;
    ...
    return true;
  },
...
```

- *float*: El tipo del campo es *number* y el número contiene parte decimal.

```
var DiffMethodsExclusive =
{
  isOfExactType_Author_1: function (obj)
  {
    ...
    if (!("salary" in obj) || !(typeof obj.salary === "number")
      || (obj.salary % 1 === 0))
        return false;
    ...
    return true;
  },
...
```

- *bool o boolean*: El tipo del campo es *boolean* y su valor es igual a *'true'* o *'false'*.

```
var DiffMethodsExclusive =
{
  isOfExactType_Book_1: function (obj)
  {
    ...
    if (!("isFirstEdition" in obj)
      || !(typeof obj.isFirstEdition === "boolean")
      || ((obj.isFirstEdition !== true) && (obj.isFirstEdition !== false)))
        return false;
    ...
    return true;
  },
...
```

Es notable el hecho de que en Javascript no existe ninguna distinción entre los tipos *int* y *float* y similares (*double*, *long*, *short*...), debido a que en Javascript todos los valores numéricos atienden al tipo *number*. Por ello se debe realizar una comprobación adicional. Actualmente se comprueba si el campo es numérico y tiene (o no) una parte decimal.

Algunas librerías de Javascript utilizadas de apoyo, como **jQuery** [9], al procesar ficheros JSON pueden convertir automáticamente valores de tipo *float* en valores de tipo *int*, como '26.0' en '26'. Este caso excepcional llevaría erróneamente a considerar un campo de un tipo como otro tipo.

7.3.4.3 Tipos HomogeneousTupleType

Para un campo de tipo *HomogeneousTupleType* el tipo se almacena como una cadena de texto que indica el tipo base, por lo que las comprobaciones son:

- Existe un campo con el nombre dado.
- Este campo es un *array*.
- Todos los elementos que componen este *array* son o bien del tipo dado o bien *arrays*. Esta comprobación se debe efectuar para cada *array* recursivamente.

```
var DiffMethodsExclusive =
{
  isOfExactType_Journal_1: function (obj)
  {
    ...
    if (!("issn" in obj) || !(obj.issn.constructor === Array)
      || (!checkAllOf(obj.issn, "string")))
        return false;
    if (!("featured_authors" in obj)
      || !(obj.featured_authors.constructor === Array)
      || (!checkAllOf(obj.featured_authors, "string")))
        return false;
    ...
    return true;
  },
...
```

En el código de ejemplo mostrado se utiliza una función *checkAllOf(obj, type)* definida en un fichero de funciones auxiliares que se describirá en secciones posteriores. Basta saber por ahora que esta función comprueba si todos los elementos de un *array* son del tipo dado o bien *arrays* que son comprobados de la misma forma, y devuelve un valor *booleano*.

Como se puede observar, y tal y como se comentó anteriormente, en esta ocasión no se diferencia entre tuplas homogéneas de distinto número de elementos o distinta estructura interior. Esta decisión es particular para este caso de estudio y podría no tener sentido para otros casos. La justificación para adoptar esta postura consiste en que dos objetos con exactamente los mismos campos con los mismos nombres y los mismos tipos pero con distinto número de elementos en una tupla homogénea no representan dos versiones distintas de una misma entidad, por lo que se ha adoptado un enfoque algo más flexible en este aspecto.

7.3.4.4 Tipos HeterogeneousTupleType

Para un campo de tipo *HeterogeneousTupleType* el tipo se registra como una lista de cadenas de texto donde cada elemento de dicha lista representa el tipo de la misma posición en la tupla. Por ejemplo, para una tupla *Tuple[Int, Float, Tuple[string,string]]* se tendrá una lista de tres elementos, siendo el primero el elemento *Int*, el segundo *float* y el tercero *Tuple[string;string]*. El carácter especial *';'* es utilizado para separar elementos dentro de tuplas internas. Para cada tupla heterogénea se debe comprobar que:

- Existe un campo con el nombre dado.

- Este campo es un *array*.
- Este *array* tiene el número de elementos esperados.
- Cada uno de estos elementos es del tipo esperado. Para este paso se hará uso de las comparaciones de tuplas y de tipos primitivos.

```
var DiffMethodsExclusive =
{
  isOfExactType_Book_1: function (obj)
  {
    ...
    if (!("atTuple_0" in obj) || !(obj.atTuple_0.constructor === Array)
      || !(obj.atTuple_0.length === 2)
      || !(typeof obj.atTuple_0[0] === "string")
      || !(typeof obj.atTuple_0[1] === "number")
      || !(obj.atTuple_0[1] % 1 === 0)
    )
      return false;
    if (!("atTuple_1" in obj) || !(obj.atTuple_1.constructor === Array)
      || !(obj.atTuple_1.length === 3)
      || !(typeof obj.atTuple_1[0] === "number")
      || (obj.atTuple_1[0] % 1 === 0)
      || !(typeof obj.atTuple_1[1] === "number")
      || (obj.atTuple_1[1] % 1 === 0)
      || !(typeof obj.atTuple_1[2] === "string")
    )
      return false;
    ...
    return true;
  },
...
```

Este caso sí es importante además del tipo de cada elemento, el número de los mismos. Incluso si se encuentran tuplas anidadas a varios niveles se harán comprobaciones de tamaño y tipo de todos sus elementos recursivamente.

7.3.4.5 Tipos ReferenceType

Un campo referencia en una versión viene identificado por el nombre de dicho campo, una cardinalidad mínima, una cardinalidad máxima y la referencia o referencias, expresadas de dos posibles formas:

- Como un campo de tipo *string* con un único código de referencia (campo *journal*).

```
{
  "_id": 10,
  "type": "Publisher",
  "name": "value_176",
  "journal": "reference_98"
}
```

- Con un *array* que contenga uno o más *strings*, siendo cada uno un código de referencia (campos *journal*).

```
{
  "_id": 11,
  "type": "Publisher",
  "name": "value_177",
  "journal": ["reference_17"]
},
{
  "_id": 12,
  "type": "Publisher",
  "name": "value_178",
  "journal": ["reference_17", "reference_18", "reference_19"]
}
```

Estas dos opciones para declarar referencias son válidas, siempre y cuando se satisfagan las cardinalidades asignadas. Por ejemplo, si la cardinalidad mínima es mayor a uno no será posible declarar la referencia como *string*. Para cada *ReferenceType* se debe comprobar que:

- Existe un campo con el nombre dado.
- O este campo es un *string* teniendo una cardinalidad mínima de 1 o inferior.
- O bien este campo es un *array*, y entonces debe compararse el número de elementos de este *array* y asegurarse que se encuentra entre las cardinalidades declaradas, además de comprobar que cada elemento contenido en el *array* es de tipo *string*.
- Si el tipo del campo no es ni *string* ni *array*, la referencia está mal formada.

```
var DiffMethodsExclusive =
{
  isOfExactType_Publisher_3: function (obj)
  {
    ...
    if (!("journal" in obj)
      || (typeof obj.journal === "string" && false)
      || (obj.journal.constructor === Array
        && (1 > obj.journal.size|| !checkAllOf(obj.journal, "string"))
        || (typeof obj.journal !== "string"
          && obj.journal.constructor !== Array)))
                    return false;
    ...
    return true;
  },
...
```

Como se observa no se hace una comparación estricta de cardinalidades: Únicamente se comprueba que el número de elementos en el *array* está dentro de los límites de estas cardinalidades. Por otro lado tampoco se comprueba la corrección de los códigos de referencia, únicamente la corrección del tipo de dato. El insertar el valor *booleano false* en la comparación se debe a que en tiempo de generación de código se ha comprobado que sí es posible aceptar un *string* como referencia. No es posible declarar una referencia como un *string* si la cardinalidad mínima de la misma es de dos o mayor.

7.3.4.6 Tipos AggregateType

Un campo de agregación en una versión es muy similar a una referencia, con la diferencia de que la agregación no se da como un *string* o un *array* de *strings*:

- Se puede dar la agregación como un objeto JSON embebido (campo *content*).

```
{
  "_id": 23,
  "type": "Book",
  "title": "value_150",
  "year": 187,
  "publisher": "168",
  "content":
  {
    "_id": 53,
    "type": "Content",
    "chapters": 26,
    "pages": 8
  }
}
```

- O con un *array* que contenga uno o más objetos JSON (campo *content*).

```
{
  "_id": 23,
  "type": "Book",
  "title": "value_150",
  "year": 187,
  "publisher": "168",
  "content":
  [
    {
      "_id": 53,
      "type": "Content",
      "chapters": 26,
      "pages": 8
    }
  ]
}
```

De nuevo las dos opciones mostradas son válidas mientras se satisfagan las cardinalidades asignadas. Para cada *AggregateType* se debe comprobar que:

- Existe un campo con el nombre dado.
- O este campo es un objeto JSON teniendo una cardinalidad mínima de 1 o inferior y este objeto pasa el chequeo de su versión para comprobar si es de la versión declarada por la agregación.
- O bien este campo es un *array*, y entonces debe compararse el número de elementos de este *array*, asegurarse que se encuentra entre las cardinalidades declaradas, que todos los elementos son objetos JSON y comprobar para cada elemento si este es compatible con la versión que se ha declarado en el *array*.

- Si el tipo del campo no es ni objeto JSON ni *array*, la agregación está mal formada.

```
var DiffMethodsExclusive =
{
  isOfExactType_Book_1: function (obj)
  {
    ...
    if (!("content" in obj)
      || (typeof obj.content === "object"
        && !(obj.content instanceof Array)
        && (!this.isOfExactType_Content_1(obj.content)))
      || (obj.content.constructor === Array
        && (1 > obj.content.size
          || 1 < obj.content.size || !checkAllOf(obj.content, "object")
          || obj.content[0] == null
          || !this.isOfExactType_Content_1(obj.content[0])))
      || (typeof obj.content !== "object"
        && obj.content.constructor !== Array))
        return false;
    ...
    return true;
  },
...
```

De nuevo se lleva a cabo una comparación no estricta de cardinalidades y únicamente se comprueba que el número de elementos se encuentra contenido entre las cardinalidades declaradas.

7.3.5 Código de conteo de objetos de versiones

El fichero de estructura tiene el nombre *__STRUCT_<nombre_modelo>.js* y la estructura que se muestra a continuación:

```
/**
 * The name of the NoSQLData model.
 */
var DATA_DIFF_NAME = "Food";

/**
 * A DiffStruct which will store the EntityVersions and how many of them
exist.
 */
var DiffStruct =
{
        Dish:
        {
                Dish_1: 0,
                Dish_2: 0,
                Dish_3: 0,
                Dish_4: 0
        },
        Ingredient:
        {
                Ingredient_1: 0,
                Ingredient_2: 0
        },
```

```
        Menu:
        {
                Menu_1: 0
        },
        Restaurant:
        {
                Restaurant_1: 0
        },
        Table:
        {
                Table_1: 0,
                Table_2: 0
        },
        Waiter:
        {
                Waiter_1: 0,
                Waiter_2: 0,
                Waiter_3: 0
        }
};
```

Esta estructura se corresponde con el caso de uso que se detallará en la sección siguiente. En primer lugar se define una variable con valor igual al nombre del modelo de entrada. Este nombre se utilizará para personalizar los resultados en la página web que los visualiza. También se declara la estructura *DiffStruct*, que en su interior contiene como miembros los nombres de las entidades definidas. Estos campos con nombre de entidad son únicos, ya que en un modelo de entrada no puede haber dos entidades con el mismo nombre, y cada uno de ellos tiene, a su vez, miembros para declarar las versiones que existen de los mismos. Esta información existía en el modelo *Version_Diff* al registrarse, para cada clase *TypeDifference*, el nombre de la entidad seguido de la versión que se estaba identificando mediante campos *HasField* y *HasNotField*.

Los campos numéricos se inicializan a cero. Estos campos llevan un conteo de cuántos objetos JSON se han identificado con una versión concreta. Al final del proceso de clasificación esta estructura almacenará cuántas instancias (objetos JSON) existen de cada versión y para cada entidad.

Es necesario predefinir *DiffStruct* estáticamente y no crearla en tiempo de ejecución debido a que es posible que la entrada JSON dada por el usuario no contenga ningún objeto identificado con algún tipo de versión concreta. Si se da este caso, en la representación se debe hacer notorio el hecho de que existen cero objetos de una versión particular.

7.4 Tipos de visualizaciones

Los tipos de visualizaciones se encuentran definidos en librerías de código estático. Por código estático se entiende aquel código que no es dependiente de la entrada proporcionada. Los ficheros que solo contienen código estático son aquellos usados para representar diagramas en *D3* y ficheros con funciones auxiliares. Estos diagramas *D3* sirven como

pruebas de concepto para mostrar lo fácilmente representable que es la estructura *DiffStruct* obtenida, y proporciona una idea de qué representaciones visuales pueden ser más útiles.

Todos los métodos D3 que se usan en la visualización reciben dos parámetros: La estructura *DiffStruct* a mostrar, que se reorganizará convenientemente por cada método como convenga, y un identificador de un contenedor HTML de tipo *div* donde dibujar el diagrama.

A continuación se explicarán las representaciones visuales en *D3.js*. No se explicará en detalles la funcionalidad de los métodos que las proveen, basta saber que la librería *D3.js* se basa en la manipulación del árbol *DOM* de elementos para, mediante sucesivas operaciones *append*, anexar elementos contenedores y visuales con distintos propósitos: *SVG* para almacenar gráficos, *text* para títulos o *title* para *tooltips*, por ejemplo. Para cada elemento añadido se define, además, ciertas clases, dimensiones y atributos adicionales.

- *js/lib/d3_bars.js*: Este método recrea una representación visual en forma de familias de barras. Un ejemplo de la representación puede apreciarse en la figura 7.7. Tras transformar *DiffStruct* en un objeto JSON compatible, cada barra representa una versión de una cierta entidad, y la longitud de la barra muestra cuántas instancias se han encontrado de la versión indicada. Se han agrupado las barras de versiones de una misma entidad para facilitar tareas como la identificación de la versión más numerosa para una entidad, ya que ese es el propósito principal de este tipo de gráfico: Facilitar la tarea de comparación entre versiones de una misma entidad, y la comparación de números de versiones entre entidades.

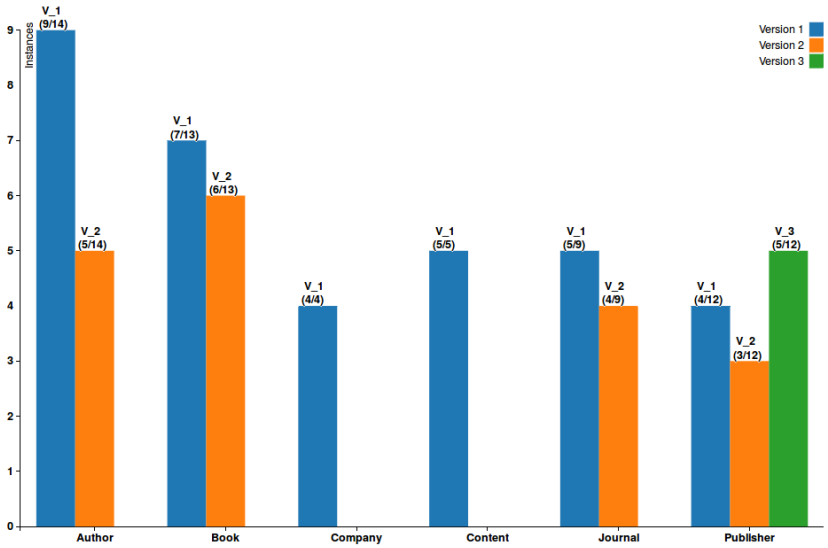


Figura 7.7: Vista de un diagrama de barras simple.

- *js/lib/d3_bubbles.js*: Este diagrama sirve como vista general y con él se puede comprobar rápidamente cuáles son las entidades más importantes del diagrama y las versiones más numerosas, como se puede ver en la figura 7.8. El diagrama consiste en una serie de burbujas anidadas donde la burbuja global representa todo el conjunto de objetos, las burbujas intermedias representan las entidades y las burbujas internas las versiones. A mayor tamaño de las burbujas mayor número de versiones contienen. Además se proporciona un *tooltip* sobre cada burbuja para agregar información sobre la misma.

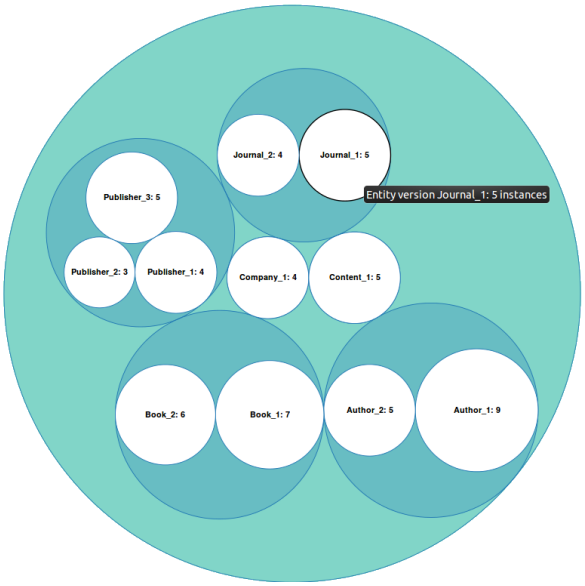


Figura 7.8: Vista de un diagrama de burbujas simple.

- *js/lib/d3_donuts.js*: La tercera representación propuesta agrega las entidades en forma de dónuts, tal y como muestra la figura 7.9. En esta ocasión la estructura *DiffStruct* es transformada en un objeto *CSV*. El diagrama de dónuts es utilizado para realizar comparaciones porcentuales. Dada la naturaleza circular de los dónuts mostrados es más complicada la comparación numérica, pero sirve para adivinar de forma acertada cuáles son las versiones más numerosas en una misma entidad. Se ha agregado un *tooltip* a esta representación para ofrecer más información al usuario.

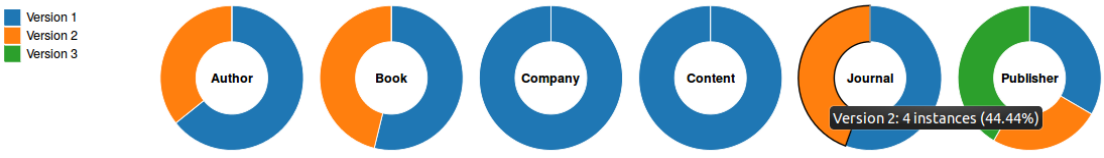


Figura 7.9: Vista de un diagrama de dónuts simple.

- *js/lib/d3_tree.js*: La última representación ordena todas las entidades y versiones en una estructura de árbol, como el de la figura 7.10. Para ello es necesario construir un objeto JSON a partir de la estructura *DiffStruct* algo más enriquecido que para el resto de representaciones. El usuario puede interactuar con el árbol para investigar las ramas y observar, para cada versión, cuántas instancias se han encontrado y qué porcentaje representan las mismas del total de instancias.

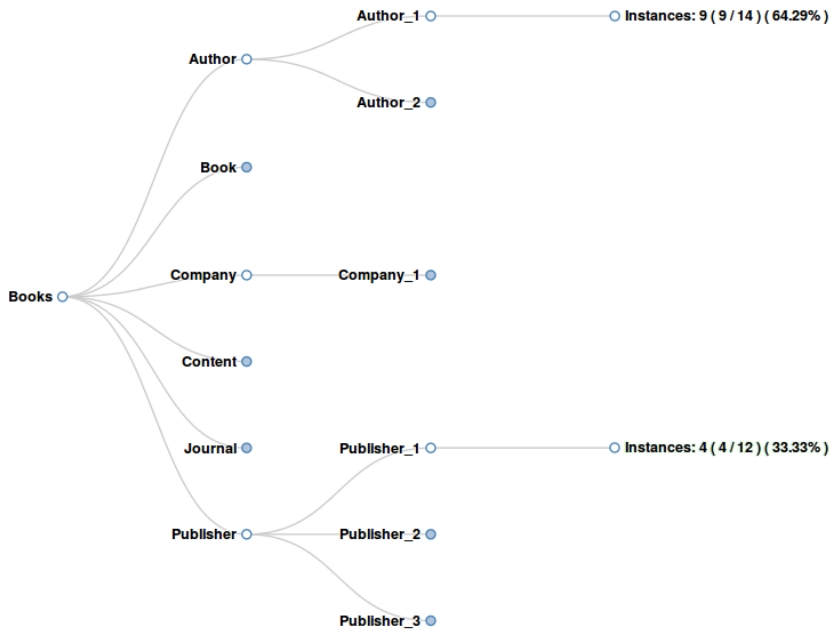


Figura 7.10: Vista de un diagrama de árbol simple.

- *js/lib/functions.js*: Fichero de funciones auxiliares de comprobación de tipos. Las funciones más utilizadas son las siguientes:

  - isOfType(obj, exclusive_bool): Función utilizada para iterar por cada campo de los objetos *DiffMethods*. Estos objetos contenían funciones para identificar a cada versión, y es posible iterar por todos sus campos para determinar qué funciones de los mismos devuelven un resultado *true*. Este resultado indica que el objeto pasado como parámetro es de la versión indicada.

    Ejecutando todos los métodos de identificación de versiones con un objeto de entrada es posible determinar todas las versiones compatibles con dicho objeto, que son retornadas en un *array*.

```
function isOfType (obj, exclusive_bool)
{
  var DiffMethods = exclusive_bool ? DiffMethodsExclusive :
DiffMethodsInclusive;

  var response = [];
```

```
  for (var i in DiffMethods)
    if (DiffMethods[i](obj))
      response.push(i.replace(FUNCTION_BASE_NAME, ""));

  return response;
};
```

- *checkAllOf(array, type)*: Función empleada para comprobar si un *array* es de un tipo determinado. Actualmente se soportan comparaciones de tipos primitivos: *string*, *int*, *float*, *boolean* y *object*. Si se encuentra un *array* interior, se iterará sobre todos sus elementos recursivamente de la forma descrita.
- Funciones de conversión de la estructura *DiffStruct* a objetos JSON: Aprovechando la sencillez con la que se puede trabajar con objetos JSON en *Javascript* se pueden crear ciertos métodos utilizados por los métodos *D3* para transformar *DiffStruct* en un objeto con un formato determinado y compatible con el diagrama que se va a representar.

- *js/lib/main.js*: El método más importante de este fichero es el método *main* ejecutado nada más cargarse la página. Este método se ejecuta aprovechando las posibilidades que ofrece **jQuery**. *jQuery* es utilizado en este proceso como una librería de apoyo auxiliar para cargar los diagramas al terminar de cargar los elementos HTML y para cargar de forma sencilla un fichero local JSON.

  Esta función se encarga de, una vez cargado este fichero de objetos, iterar sobre cada uno obteniendo las versiones a las que pertenece y sumando los resultados obtenidos a los campos apropiados de *DiffStruct*. Cuando se ha terminado de iterar sobre cada objeto JSON esta estructura se encuentra lista para ser representada. Por último el método hace las llamadas necesarias a los métodos de visualización.

En el siguiente apartado se analizará el comportamiento de las dos transformaciones descritas, con énfasis en la aplicación de los conceptos aquí explicados con un ejemplo práctico.

7.5 Aplicación al caso de estudio

Para realizar el *testing* de las transformaciones *m2m* y *m2t* implementadas se han creado distintos ficheros de entrada JSON y modelos de entrada *NoSQL_Schema* para los que generar código. En este apartado se ha seleccionado uno de estos modelos para ilustrar el funcionamiento del proceso y comprobar las visualizaciones obtenidas. La figura 7.11 muestra el modelo *NoSQL_Schema* elegidos y parte del correspondiente modelo *Version_Diff* que ha sido entrada al proceso de generación de código: *Food.nosql_schema*. Este modelo, extraído a partir del dominio del mundo del restaurante,se ha tomado del caso de estudio del capítulo 6.4 anterior.

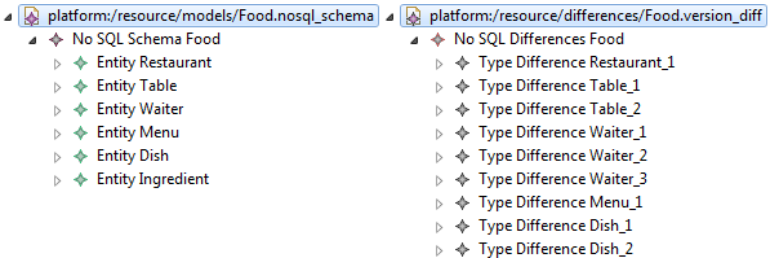


Figura 7.11: *NoSQL_Schema* y parte de *Version_Diff* del ejemplo *Food.nosql_schema*.

La transformación *modelo a modelo* consiste en la ejecución de una clase Java que tiene como entrada un modelo *NoSQL_Schema*. En la figura 7.12 se pueden ver indicados algunos de los campos *HasField* y *HasNotField* detallados con sus tipos *FieldType*.

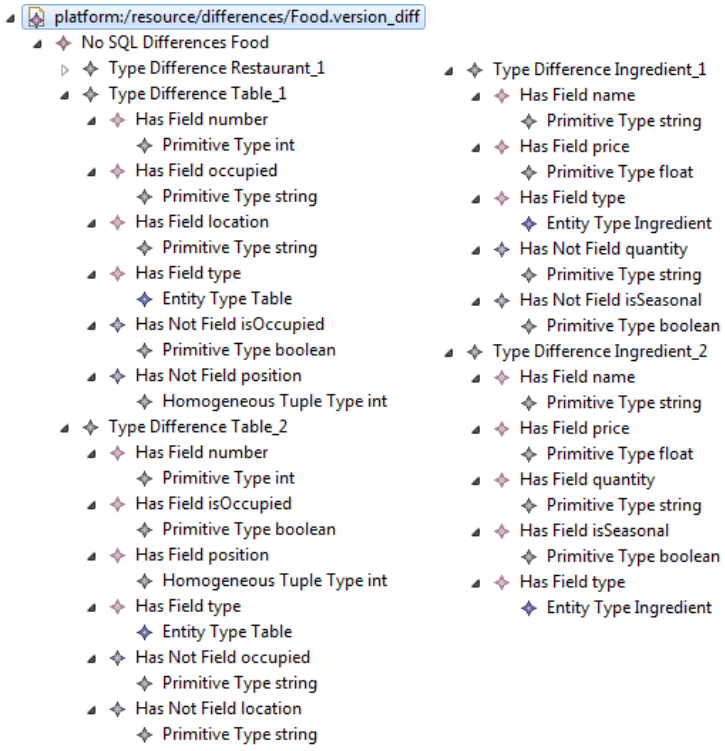


Figura 7.12: Detalle de algunos campos de *Version_Diff* para el ejemplo.

En las figuras 7.11 y 7.12 se observa que los campos generados tienen la estructura esperada, y constan de elementos *HasField* con el nombre del campo que se declara y el tipo del mismo (recordar que los tipos posibles eran *EntityType*, *PrimitiveType*, *HomogeneousTupleType*, *HeterogeneousTupleType*, *AggregateType* y *ReferenceType*). Del mismo modo

en las versiones que es conveniente (como las versiones de *Table* en *NoSQL_Schema*) se han definido elementos *HasNotField*.

7.5.1 Entrada JSON del proceso

Una vez comentado todo el código el único paso que falta para visualizar resultados es el de proveer al código generado de un fichero JSON que contenga datos NoSQL en el formato de una base de datos de documentos. Dado que se disponía de modelos *NoSQL_Schema* se ha optado por utilizarlos para crear generadores aleatorios de objetos JSON. Para ello se ha creado la clase Java denominada *random/JsonGenerator.java*, que se puede encontrar en el proyecto *NoSQLDataVisualization*. Esta clase contiene un método que recibe un modelo *NoSQL_Schema* y genera una serie de objetos JSON pertenecientes a versiones aleatorias definidas en el modelo, con los atributos esperados de los tipos apropiados. Estos valores, como se ha visto, no son comprobados en ejecución por lo que la aleatoriedad de los mismos no supone ningún impacto en el resultado final.

En un entorno real donde se extraigan los objetos JSON de la base de datos NoSQL con la que se interactúa estos objetos JSON pueden estar en un formato u otro dependiendo del tipo de base de datos con el que se trabaje. La exportación de la información de la base de datos NoSQL es dependiente del tipo de base de datos, y difiere de unas a otras, por lo que puede ser necesario crear un adaptador para transformar los objetos JSON obtenidos en una lista de objetos JSON esperada. Un ejemplo de lista de objetos JSON generados aleatoriamente es el siguiente:

```
[
  {
    "_id": 1,
    "type": "Restaurant",
    "name": "value_12",
    "location": "value_12",
    "manager": "value_64",
    "hasWaiters": [
      "85",
      "133",
      "117"
    ],
    "hasMenu": [
      "23"
    ]
  },
  ...
  {
    "_id": 14,
    "type": "Table",
    "number": 18,
    "occupied": "value_21",
    "location": "value_28"
  },
  ...
  {
    "_id": 25,
    "type": "Waiter",
```

```
    "name": "value_82",
    "nif": "value_169",
    "hours_per_week": 42,
    "salary": 2.51,
    "tablesAssigned": [
      35,
      123,
      96
    ]
  },
  ...
  {
    "_id": 40,
    "type": "Dish",
    "name": "value_114",
    "price": 31.47,
    "hasIngredients": [
      {
        "_id": 57,
        "type": "Ingredient",
        "name": "value_170",
        "price": 63.76,
        "quantity": "value_141",
        "isSeasonal": false
      }
    ]
  }
  ...
```


7.5.2 Resultados obtenidos

En esta sección se ilustrarán los resultados obtenidos al aplicar un fichero JSON de entrada al proceso descrito para el modelo de entrada *Food.nosql_schema*.

Para automatizar todo el proceso de generación de modelos, código y empaquetamiento de ficheros generados junto con la librería se ha creado una clase Java que se puede encontrar en *execution/Main.java*, del proyecto *NoSQLDataVisualization*. Esta clase generará el proceso para todos los modelos de entrada de una carpeta y creará una estructura de carpetas y ficheros listos para visualizar resultados.

Este ejemplo ha sido visualizado con el navegador web **Mozilla Firefox v46.0.1**. También ha sido probado con **Internet Explorer v11.0.9600.18314**. No ha sido posible visualizar este ejemplo con un navegador **Google Chrome** debido a que, por seguridad, este navegador impide cargar ficheros JSON locales desde código Javascript.

Como notas generales se debe reseñar que en las visualizaciones no se hace uso de todas las propiedades declaradas en el modelo *Version_Diff*. Algunas de estas propiedades agregan información que no es útil en la visualización.

El enfoque generalista tomado para definir este metamodelo puede servir en el futuro para otras tareas de visualización, como el análisis de atributos comunes a muchas versiones de objetos, o el estudio de qué atributos tienden a variar más entre versiones. En el apartado 8.1 se señalará esta posibilidad.

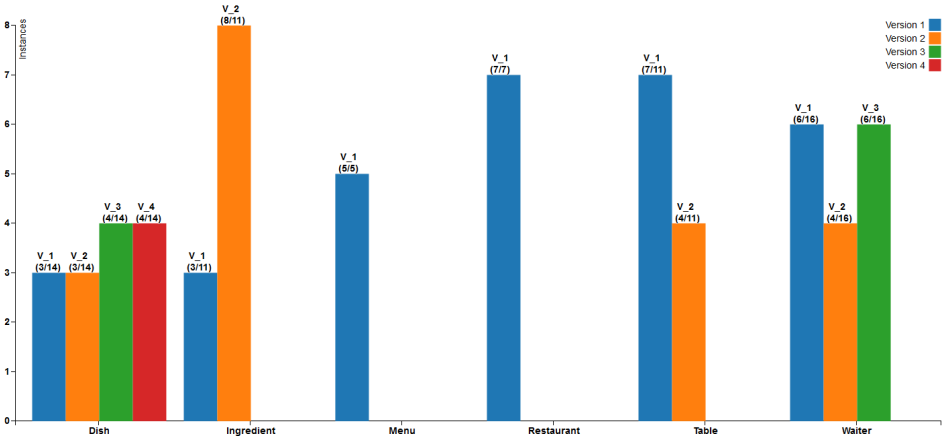


Figura 7.13: Diagrama de barras para *Food.nosql_schema*.

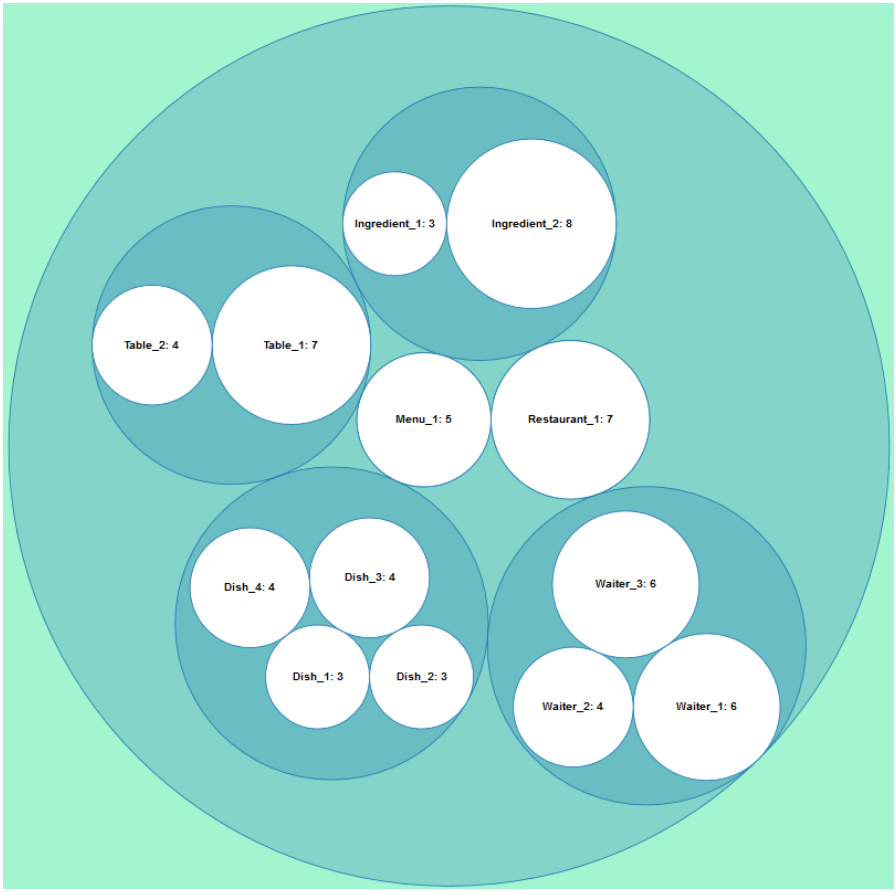


Figura 7.14: Diagrama de burbuja para *Food.nosql_schema*.

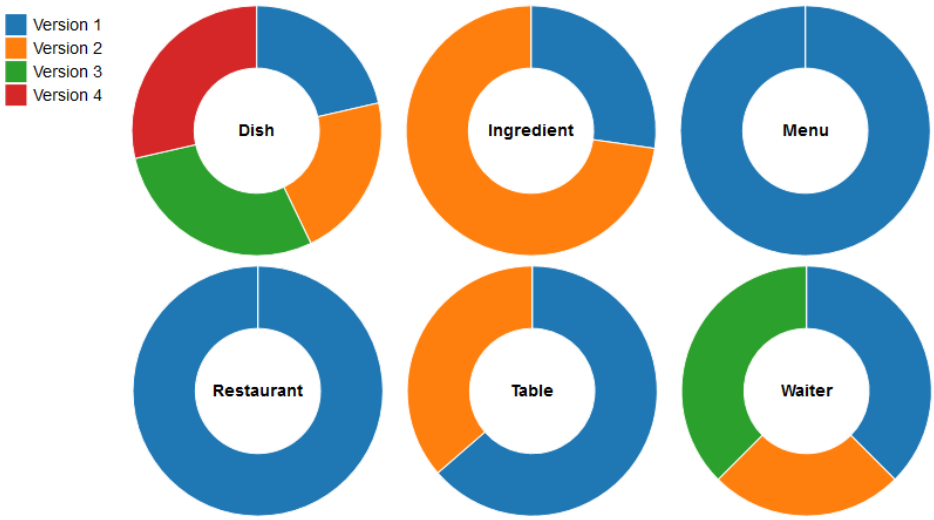


Figura 7.15: Diagrama de dónuts para *Food.nosql_schema*.

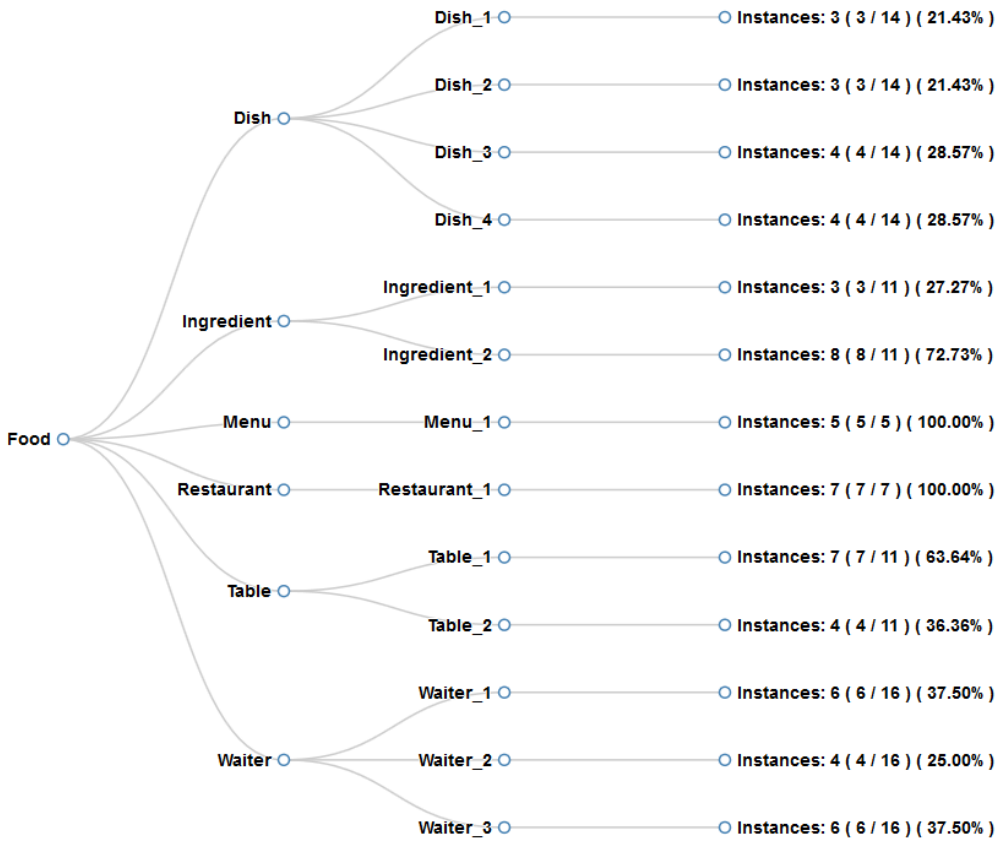


Figura 7.16: Diagrama de árbol para *Food.nosql_schema*.

8 Conclusiones

Este trabajo ha presentado una de las primeras propuestas para mostrar esquemas de bases de datos NoSQL teniendo en cuenta la existencia de versiones de las distintas entidades almacenadas. Además, también ha abordado la visualización de datos NoSQL que son clasificados de acuerdo a la versión a la que pertenecen.

Para la visualización de esquemas se han propuesto distintos diagramas y árboles, cada uno de los cuales responde a diferentes tipos de información del esquema en los que pueda estar interesado el usuario, desde un esquema global con todas las entidades y relaciones entre ellas a otros que muestran versiones de esquemas. Esto ha supuesto proponer algunas definiciones tales como *entidad raíz*, *esquema* y *versión de esquema*. Los diagramas y árboles diseñados han sido definidos en Sirius. La funcionalidad y prestaciones de esta herramienta han permitido llevar a cabo las visualizaciones previamente diseñadas.

En cuanto a la visualización de datos se han mostrado diferentes gráficos basados en *D3.js* que ofrecen información sobre la distribución de objetos almacenados en versiones y las propiedades que caracterizan a cada versión y las diferencias entre ellas. Se han definido distintas vistas de datos que han sido implementadas mediante representaciones gráficas estáticas con la librería D3.js en Javascript. El proyecto ha detallado la manera de mostrar visualmente análisis de datos NoSQL planteados en otros trabajos.

Por otra parte, el uso de técnicas y herramientas MDE en la implementación ha facilitado el desarrollo de las dos soluciones. Por ejemplo, (i) el metamodelo *NoSQL_Schema* facilita la independencia de sistemas NoSQL concretos al tiempo que ofrece una representación de alto nivel; (ii) el uso de Sirius ha reducido considerablemente el esfuerzo requerido para implementar la herramienta de visualización de esquemas desde cero; (iii) las transformaciones modelo a texto han permitido automatizar la generación del código que realiza las clasificaciones de los datos en sus correspondientes versiones.

Cabe observar algunas limitaciones en el uso de las dos herramientas. La visualización de esquemas NoSQL puede verse dificultada si existe una *versión de esquema* que englobe a una cantidad arbitrariamente grande de versiones y entidades (centenares), ya que la herramienta de visualización no conseguirá mostrar de forma clara todas las entidades involucradas en dicho esquema. En el caso de la visualización de datos, la limitación consiste en el procesamiento y clasificación de los objetos JSON proporcionados como entrada. Un conjunto de objetos JSON arbitrariamente grande (miles) podría llevar a un tiempo de clasificación alto que provocaría una experiencia de usuario negativa.

8.1 Vías futuras

Para finalizar el capítulo se proponen distintas vías futuras y guías para posteriores desarrollos a partir del trabajo presentado, con el fin de extender o mejorar algún aspecto:

- En primer lugar es posible aprovechar las posibilidades de *Sirius* para desarrollar una herramienta de manipulación de modelos completa. Esta herramienta serviría para crear, visualizar y modificar modelos *NoSQL_Schema* con el fin de representar, rápidamente y de forma gráfica, esquemas internos de bases de datos NoSQL que posteriormente podrían ser creados o migrados.
- Puede ser interesante exportar la visualización realizada en este proyecto mediante *Sirius* a un entorno independiente de Eclipse usando otra herramienta de visualización. De este modo se dispondría de una aplicación *Java* a la que suministrar un modelo de entrada *NoSQL_Schema* que realice las transformaciones y generaciones necesarias para obtener la visualización de estos esquemas, por ejemplo utilizando *D3.js* tal y como se recoge en el capítulo 7.
- Respecto a la segunda parte, se podría llevar la funcionalidad de visualización de datos a un servicio web. Este servicio web, montado por ejemplo sobre *Node.js* [10], permitiría al usuario enviar un modelo siguiendo el metamodelo (distribuible como *plugin* para Eclipse) y un fichero JSON de entrada, y realizar la evaluación remota y visualización de resultados. Este podría ser un primer paso a abrir el proyecto a los usuarios y daría pie a futuras mejoras.
- Otro punto de mejora en la segunda parte podría ser realizar un procesamiento en dos pasos. Cuando el usuario abre el fichero *index.html* con un navegador se procede a la clasificación de instancias y visualización de las mismas. Esto es permisible para ejemplos pequeños y medianos pero resulta inviable para casos de entrada muy grandes. La utilización de *Node.js* podría ayudar a clasificar las instancias de versiones como un paso previo, antes de que el usuario visualice los resultados.
- Otra propuesta futura es la de implementar adaptadores de bases de datos. En los casos de uso hemos partido de ficheros JSON con un formato determinado. Este formato es muy similar al generado por bases de datos NoSQL como *MongoDB*, pero no se puede garantizar que toda base de datos ofrezca el mismo formato de texto plano similar al JSON esperado. La creación de adaptadores de bases de datos NoSQL particulares ayudaría en este proceso.
- Para acabar quizá sea interesante explotar el metamodelo *Version_Diff*. Este metamodelo es utilizado en el proyecto para identificar de forma única instancias de versiones, pero dado que se contiene información sobre cada atributo, podría utilizarse para realizar visualizaciones basadas en atributos o para destacar progresiones de versiones de una misma entidad.

Bibliografía

[1] 50 Javascript libraries for Charts and Graphs. http://techslides.com/50-javascript-charting-and-graphics-libraries. Último acceso: 22-05-2016.

[2] Eclipse Software Foundation. https://eclipse.org/org/. Último acceso: 26-05-2016.

[3] JavaScript JSON. http://www.w3schools.com/js/js_json.asp. Último acceso: 22-05-2016.

[4] Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/. Último acceso: 15-06-2016.

[5] Página de instalación de Sirius. https://eclipse.org/sirius/download.html. Último acceso: 26-05-2016.

[6] Página del OMG de MOF. http://www.omg.org/spec/MOF/2.0/. Último acceso: 08-06-2016.

[7] Página principal de Acceleo. http://www.eclipse.org/acceleo/. Último acceso: 26-05-2016.

[8] Página principal de D3.js. https://d3js.org/. Último acceso: 22-05-2016.

[9] Página principal de jQuery. https://jquery.com/. Último acceso: 22-05-2016.

[10] Página principal de NodeJS. https://nodejs.org/en/. Último acceso: 31-05-2016.

[11] Página principal de Obeo. http://www.obeodesigner.com/. Último acceso: 26-05-2016.

[12] Página principal de Sirius. https://eclipse.org/sirius/. Último acceso: 25-05-2016.

[13] Página principal de Thales. https://www.thalesgroup.com/en. Último acceso: 26-05-2016.

[14] Página principal de UML. http://www.uml.org/. Último acceso: 29-05-2016.

[15] Página principal de Xtend. http://www.eclipse.org/xtend/. Último acceso: 22-05-2016.

[16] Página principal del Eclipse Marketplace. `https://marketplace.eclipse.org/`. Último acceso: 26-05-2016.

[17] Página principal del proyecto Kiama. `https://bitbucket.org/inkytonik/kiama`. Último acceso: 12-06-2016.

[18] Update Site de Sirius. `http://download.eclipse.org/sirius/updates/releases/3.1.1/mars`. Último acceso: 26-05-2016.

[19] Why build Data Visualizations with D3.js. `https://www.dashingd3js.com/why-build-with-d3js`. Último acceso: 22-05-2016.

[20] *Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores*, 2015.

[21] S. Abiteboul. Querying Semi-Structured Data. Technical Report 1996–19, Stanford InfoLab, 1996.

[22] Antonio García Alba. Guía rápida de Sirius. `https://www.gitbook.com/book/galba/desarrollo-de-dsl-visuales-con-sirius/details`. Último acceso: 26-05-2016.

[23] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[24] P Buneman. Semistructured data. In *Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121. ACM, 1997.

[25] DataDiversity. Insights into NoSQL Modeling Report, 2015.

[26] M. Fowler. *Domain-specific languages*. Addison-Wesley, 2010.

[27] M. Fowler. Schemaless Data Structures, January 2013.

[28] J. García-Molina, F.O. García, V. Pelechano, A. Vallecillo, J.M. Vara, and C. Vicente-Chicote. Desarrollo de software dirigido por modelos: Conceptos, métodos y herramientas. 2013.

[29] Object Management Group. Página principal de OCL. `http://www.omg.org/spec/OCL/`. Último acceso: 26-05-2016.

[30] J.L.C. Izquierdo, J.S. Cuadrado, and J.G. Molina. Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization. In *Workshop on Model-Driven Software Evolution*, 2008.

[31] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[32] Peffers K., Tuunanen T., Rothenberger MA., and Chatterjee S. A design science research methodology for information systems research. In *Journal of Management Information Systems*, volume 24, pages 45–77, September 2008.

[33] S. Kelly and R. Pohjonen. Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26(4):22–29, 2009.

[34] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.

[35] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained*. Addison-Wesley, 2003.

[36] Syed Fazle Rahman. Javascript chart libraries. `https://www.sitepoint.com/15-best-javascript-charting-libraries/`. Último acceso: 22-05-2016.

[37] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. Inferring Versioned Schemas from NoSQL Databases and its Applications. In *ER*, pages 467–480, September 2015.

[38] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. Model Driven NoSQL Data Engineering. In *JISBD*, Santander, September 2015.

[39] T. Rückstieß. `mongodb-schema` npm package. `https://www.npmjs.com/package/mongodb-schema`. Visited April 2016.

[40] P.J. Sadalage and M. Fowler. *NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[41] Drew Skau. Why D3.js is so great for Data Visualization. `http://www.scribblelive.com/blog/2013/01/29/why-d3-js-is-so-great-for-data-visualization/`. Último acceso: 22-05-2016.

[42] D. Steinberg, F. Budinsky, M. Paternostro, and E Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2009.

[43] J. Sánchez, J. García-Molina, and M. Menárguez. RubyTL: A practical, extensible transformation language. *2nd European Conference on Model-Driven Architecture*, 4066:158–172, 2006.

[44] Dan Tao. With which programming language does JSON pair best? `https://www.quora.com/With-which-programming-language-does-JSON-pair-best`. Último acceso: 22-05-2016.

[45] V. Vaishnavi and W. Kuechler. Design science research in information systems. `http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf`. Último acceso: 21-06-2016.

[46] M. Völter. MD* best practices. *Journal of Object Technology*, 8(6):79–102, 2009.

[47] Allen Wang. Unified Data Modeling for Relational and NoSQL Databases. `https://www.infoq.com/articles/unified-data-modeling-for-relational-and-nosql-databases`, February 2016.

[48] L. Wang, O. Hassanzadeh, S. Zhang, J. Shi, L. Jiao, J. Zou, and C. Wang. Schema Management for Document Stores. In *VLDB Endowment*, volume 8, 2015.

[49] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

[50] Matei Zaharia, Mosharaf Chowdhury, et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, April 2012.

A Detalle de los proyectos Java

Este anexo presenta los proyectos Java que implementan las herramientas de visualización presentadas en la tesis de máster. Se divide en dos secciones y cada sección se dedica a una herramienta de un capítulo concreto. Todo el código se puede encontrar en un repositorio *GitHub* con url `https://github.com/Soltari/NoSQLVisualizationTools` donde puede ser consultado.

A.1 Visualización de esquemas NoSQL

La herramienta desarrollada para visualizar esquemas NoSQL del capítulo 6 se compone de los siguientes proyectos Eclipse implementados en Java:

- *NoSQLSchema*: Este proyecto, mostrado en la figura A.1, contiene las definiciones de los metamodelos con los que se trabaja en la visualización de esquemas de versiones y del código Java encargado de inferir estos esquemas de versiones y manipular modelos. Las clases implementadas más interesantes son las siguientes:
  - *metamodel/NoSQLSchema.ecore*: Definición del metamodelo *NoSQL_Schema*.
  - *metamodel/ExNoSQLSchema.ecore*: Definición del metamodelo *Extended_NoSQL_Schema*.
  - *src/NoSQL_Schema/*: Paquete con clases Java usadas para manejar objetos del metamodelo *NoSQL_Schema*.
  - *src/Extended_NoSQL_Schema/*: Paquete con clases Java usadas para manejar objetos del metamodelo *Extended_NoSQL_Schema*.
  - *src/schema/utils/*: Paquete con clases Java auxiliares utilizadas para abrir y guardar modelos de los metamodelos definidos. También se han definido clases Java que permiten serializar modelos de los dos metamodelos en una cadena de texto.

- *src/transform*: Paquete con una clase Java que lleva a cabo la transformación *m2m* entre un modelo *NoSQL_Schema* y uno *Extended_NoSQL_Schema*.

- *src/analyzer/*: Paquete con una clase Java definida para obtener, a partir de un objeto *NoSQLSchema* (raíz del modelo), todas sus versiones de esquemas.

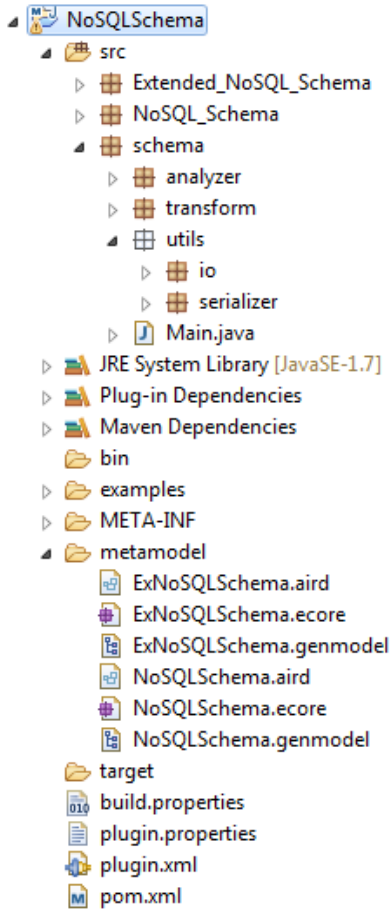


Figura A.1: Estructura del proyecto *NoSQLSchema*.

- *NoSQLSchema.edit* y *NoSQLSchema.editor*: Proyectos autogenerados a partir de los ficheros *NoSQLSchema.genmodel* y *ExNoSQLSchema.genmodel*. Estos proyectos contienen el código que permite editar modelos de los metamodelos definidos, así como proporcionar la interfaz necesaria para manipular modelos desde el editor de Eclipse.

- *NoSQLSchema.FeatureProject*: Un *feature project* es la unidad utilizada por Eclipse para almacenar plugins para una posterior distribución. Este proyecto se compone de un fichero XML que almacena los plugins generados a partir de los proyectos *NoSQLSchema*, *NoSQLSchema.edit* y *NoSQLSchema.editor*.

- *NoSQLSchema.design*: Este proyecto implementa la herramienta de visualización en Sirius, y el fichero más interesante del mismo es *NoSQLSchema.odesign*, que almacena los *viewpoints* o puntos de vista definidos en la herramienta.
- *NoSQLSchemaVisualization.FeatureProject*: Este *feature project* agrupa el plugin de la herramienta de visualización desarrollada en Sirius y separada del otro *feature project* porque han seguido líneas de desarrollo distintas.
- *NoSQLSchema.UpdateSite*: Un *update site* agrupa los distintos *feature project* creados y genera una estructura distribuible a otros usuarios para que estos puedan instalar los proyectos descritos en su instancia de Eclipse. En la figura A.2 se puede observar el contenido del *update site* desarrollado.

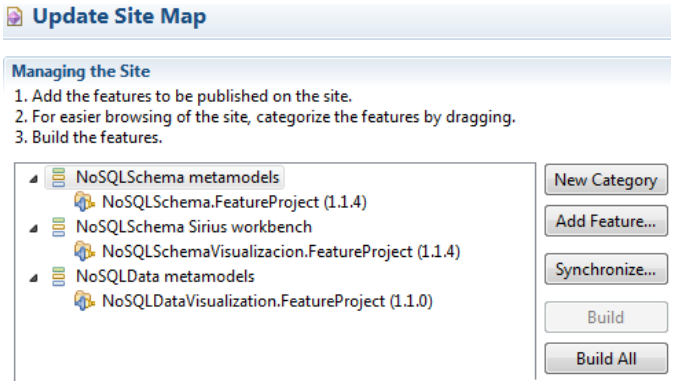


Figura A.2: Contenido del *Update Site*.

Para hacer uso de la herramienta de visualización basta con instalar este *update site*, que se puede hacer público en una URL o bien como un fichero zip. Desde la opción *Install new software* de Eclipse se instala el *update site*, y además se debe instalar el resto de dependencias necesarias: Sirius, Xtend, Maven, etc. A partir de este punto ya es posible definir modelos *NoSQL_Schema*, modelos *Extended_NoSQL_Schema* y visualizar los mismos. Es necesario importar todos los proyectos Java si el usuario desea ejecutar las transformaciones *m2m* entre los dos metamodelos. La ejecución se inicia con la clase *NoSQLSchema/src/schema/Main.java*.

A.2 Visualización de datos NoSQL

La segunda herramienta desarrollada para visualizar datos NoSQL, detallada en el capítulo 7, se compone de los siguientes proyectos Eclipse implementados en Java:

- *NoSQLDataVisualization*: Como se puede observar en la figura A.1 este proyecto contiene la estructura de ficheros para implementar la herramienta de visualización de datos NoSQL, y la definición del metamodelo *Version_Diff*. Este proyecto utiliza

las definiciones del metamodelo *NoSQL_Schema*, por lo que referencia al proyecto Java *NoSQLSchema* anteriormente comentado. Los aspectos más interesantes del proyecto son los siguientes:

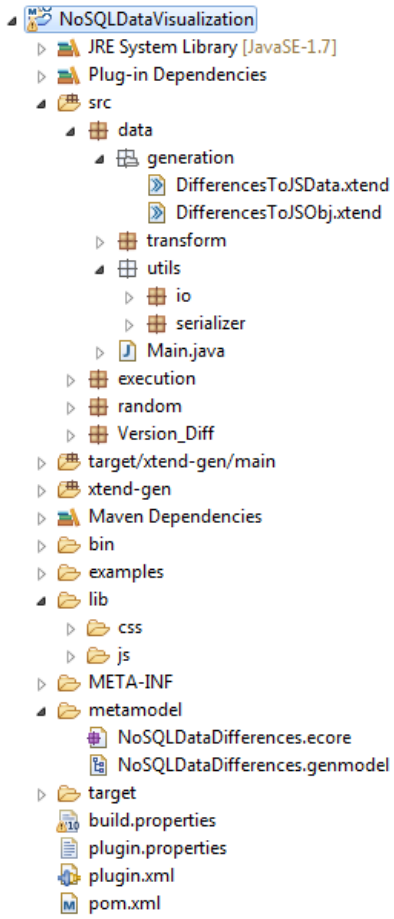


Figura A.3: Estructura del proyecto *NoSQLDataVisualization*.

- *metamodel/NoSQLDataDifferences.ecore*: Definición del metamodelo *Version_Diff*.
- *src/Version_Diff*: Paquete con clases Java usadas para manejar objetos del metamodelo *Version_Diff*.
- *src/data/utils*: Paquete con clases Java auxiliares utilizadas para abrir y guardar modelos del metamodelo definido en este proyecto, así como para serializar modelos de dicho metamodelo en cadenas de texto.
- *src/transform*: Paquete con una clase Java que lleva a cabo la transformación *m2m* entre un modelo *NoSQL_Schema* y uno *Version_Diff*.

  - *src/generation/*: Paquete con dos clases Xtend para la transformación *m2t*. Estas dos clases generan los ficheros de código HTML y Javascript necesario para visualizar datos.
  - *src/random*: Paquete con clases Java empleadas para efectuar pruebas sobre la herramienta de visualización al generar modelos de entrada aleatorios y ficheros de datos JSON aleatorios para dichos modelos.
  - *src/execution*: Paquete con una clase Java utilizado para automatizar la tarea de generar datos aleatorios para modelos de entrada y efectuar la cadena de transformaciones implementadas para finalmente generar la estructura de visualización.

- *NoSQLDataVisualization.edit* y *NoSQLDataVisualization.editor*: Estos proyectos son generados automáticamente a partir del fichero *NoSQLDataDifferences.genmodel* cuando se genera la API Java para manejar modelos conformes al metamodelo *Version_Diff*. Lo proyectos son necesarios para editar estos modelos y contienen los componentes necesarios para que Eclipse proporcione un editor manipulador de modelos muy simple.
- *NoSQLDataVisualization.FeatureProject*: Este *feature project* encapsula la funcionalidad desarrollada en el resto de proyectos de esta sección en un fichero XML, y a su vez se encuentra agregado al *update site* descrito en la sección anterior: *NoSQLSchema.UpdateSite*.
- *NoSQLDataIndex*: Aunque este no es un proyecto Eclipse implementado en Java, se genera como resultado de la ejecución para recrear la herramienta de visualización de datos desarrollada. En su interior se muestra una serie de carpetas, una por cada modelo de entrada, con un fichero *index.html* para iniciar la visualización, entre otras cosas.

La distribución de esta herramienta se realiza conjuntamente a la herramienta anterior a partir del mismo *Update site*. Instalando el plugin necesario es posible crear modelos conformes al metamodelo *Version_Diff*. Si se desea realizar las transformaciones *m2m* y *m2t* para generar la infraestructura de visualización en Javascript y D3.js es necesario importar los proyectos. La ejecución se inicia con la clase *NoSQLDataVisualization/src/data/Main.java*.