

Índice general

1. Introducción	1
1.1. Contexto del proyecto	1
1.2. Antecedentes: Ingeniería Dirigida por Modelos y Desarrollo Dirigido por modelos	3
1.3. Motivación y Objetivos	8
1.4. Estructura del Documento	9
2. Antecedentes y Planificación	11
2.1. Caso de estudio: Generador de código CQL-Cassandra	11
2.2. EMF	12
2.3. Epsilon	13
2.3.1. Epsilon Object Language	13
2.3.2. Epsilon Transformation Language	15
2.3.3. Epsilon Generation Language	15
2.4. Cassandra	16
2.5. Planificación	19
2.6. Sumario	21
3. Transformación modelo a modelo	23
3.1. Introducción	23
3.2. Metamodelo Cassandra	24
3.3. Transformación de Modelo UML a Cassandra	26
3.3.1. Transformación del modelo	26
3.3.2. Transformación de clases	26
3.3.3. Transformación de atributos	27
3.3.4. Transformación de asociaciones	27
3.3.5. Transformación de tipos primitivos	29
3.4. Caso de estudio transformación modelo a modelo	29
3.5. Sumario	31
4. Transformación modelo a texto	33
4.1. Introducción	33
4.2. Generador de código	34

4.3. Caso de estudio Twissandra	34
4.4. Pruebas	36
4.5. Despliegue	36
4.6. Sumario	36
5. Sumario y Trabajos Futuros	39
5.1. Sumario	39
5.2. Experiencia personal	39
5.3. Trabajos futuros	40

Índice de figuras

1.1. Capas del modelado de lenguajes	4
1.2. (a) Meta-modelo, (b) Sintaxis concreta, (c) Sintaxis grafica	5
1.3. Proceso de transformación de un modelo	6
1.4. Meta-modelo de un grafo	7
1.5. Meta-modelo de una red	7
1.6. Resultado transformación HTML	8
2.1. Modelo UML Twissandra	12
2.2. Ejemplo metamodelo casa	14
2.3. Ejemplo código EOL	14
2.4. Ejemplo código ETL	16
2.5. Ejemplo código EGL	17
2.6. Estructura column family	19
2.7. Ejemplo código CQL	20
3.1. Metamodelo Cassandra	25
3.2. Regla de transformación XX	28
4.1. Regla de transformación XX	35
4.2. Modelo UML Twissandra	36
4.3. Código resultante	38

Índice de cuadros

2.1. Ejemplo estructura columna	19
3.1. Equivalencias tipos primitivos UML-Cassandra	29

Capítulo 1

Introducción

Este capítulo sirve de introducción a la presente Memoria de Proyecto de Fin de Carrera. En primer lugar se realiza una breve introducción de los conceptos generales del proyecto. La siguiente sección describe conceptos orientados a la Ingeniería de Lenguajes Dirigida por Modelos y al Desarrollo Dirigido por Modelos. La tercera sección describe las motivaciones y objetivos. La última sección está dedicada a describir la estructura del documento.

Índice

1.1. Contexto del proyecto	1
1.2. Antecedentes: Ingeniería Dirigida por Modelos y Desarrollo Dirigido por modelos	3
1.3. Motivación y Objetivos	8
1.4. Estructura del Documento	9

1.1. Contexto del proyecto

El término conocido como modelado ha sido asociado a las bases de datos y a la gestión de datos durante décadas. [1] El modelo entidad-relación (ER) y el conjunto de reglas para la transformación de un modelo ER en un esquema relacional es un ejemplo bien conocido y utilizado. Recientemente las nuevas tecnologías de gestión de bases datos, también conocidas como tecnologías NoSQL, han surgido como respuesta a los nuevos retos y demandas de las aplicaciones de Internet modernas. Estas tecnologías se han centrado en el nivel de aplicación, al carecer de un soporte de modelado adecuado. Aplicaciones de Internet de uso masivo, como las redes sociales (por ejemplo, Twitter) o tiendas online (por ejemplo, Amazon), están generando nuevos desafíos en materia de almacenamiento y gestión de datos. Por ejemplo, la disponibilidad se está convirtiendo en un aspecto crítico en estos

sistemas, ya que una caída del sistema puede generar pérdidas multimillonarias a estas empresas. Además, estas aplicaciones tienen que mantener su funcionalidad ante las peticiones de un gran número de usuarios, en los que estos usuarios ejecutan operaciones muy similares (por ejemplo, publicar un mensaje en una red social después de un evento popular, como la final de la Super Bowl o unas elecciones presidenciales).

En este contexto las bases de datos relacionales han resultado ser insuficientes de cara a satisfacer estas nuevas exigencias. Las tecnologías NoSQL (Not Only SQL) [2] tienen como objetivo hacer frente a estas nuevas exigencias. NoSQL sacrifica algunas de las ventajas bien conocidas de los sistemas de gestión de bases de datos relacionales, como la integridad o la manipulación de transacciones, con el fin de proporcionar una mejor escalabilidad y un mayor rendimiento. Siguiendo esta idea, varios sistemas NoSQL, como Cassandra [3], HBase [4] o MongoDB [5], han aparecido en los últimos años.

Sin embargo, las tecnologías NoSQL no están aún integradas en los procesos de desarrollo software, su integración puede ayudar a los ingenieros software en la construcción de repositorios NoSQL desde las primeras etapas del ciclo de vida del software hasta la finalización del producto. Este trabajo tiene como objetivo contribuir con una herramienta para superar esta barrera, proporcionando un generador de código que genera bases de datos NoSQL.

Por lo tanto, este trabajo se centrará en la creación de un generador de código que transforma un modelo de datos desarrollado en UML, en código para la creación de un repositorio de datos NoSQL. Para este trabajo, nos centraremos en los sistemas orientados a columnas. Las razones son las siguientes: (1) los sistemas orientados a columnas son sistemas de propósito general, mientras que otros sistemas NoSQL, tales como, las bases de datos documentales que como su propio nombre indica están orientadas a la gestión de documentos, este tipo de bases de datos son más específicos a ese dominio; (2) los estudios cursados en Ingeniería Informática nos han proporcionado experiencia previa en el manejo de estos sistemas. Más concretamente, hemos decidido utilizar Cassandra [3] como sistema NoSQL orientado a columnas debido a su creciente popularidad. Se ha decidido que el lenguaje de modelado utilizado en este proyecto de fin de carrera sea UML por varias razones: (1) es un estándar de la OMG (Object Management Group), (2) es un lenguaje de modelado conocido y utilizado; y, (3) UML tiene un gran número de herramientas que dan soporte de cara al desarrollo.

El modelado de datos de un sistema NoSQL como Cassandra dista del modelado de las bases de datos relacionales. Cassandra por ejemplo, utiliza como unidad básica las Columns cuyo equivalente sería el Campo en el modelo relacional o como unidad de almacenamiento de estas Columns utiliza las ColumnFamily como tablas etc..

Para realizar este generador de código se utilizan una serie de técnicas de transformación entre modelos conocidas como "Desarrollo Dirigido por

Modelos” (MDD). MDD se puede definir como un enfoque de la Ingeniería del software y de la Ingeniería dirigida por modelos (MDE) que utiliza el modelo para crear un producto. ¿Y que es un modelo?. Un modelo se puede entender como la descripción o representación de un sistema en un lenguaje bien definido. Para entender lo que representa un modelo dentro de MDE hay que saber previamente lo que es un meta-modelo. Un meta-modelo es un modelo usado para especificar un lenguaje, básicamente describe las características del lenguaje. Por lo tanto un modelo se puede entender como la instancia de un meta-modelo. Estos conceptos son ampliados en siguientes secciones.

El resultado de la utilización del desarrollo dirigido por modelos es traducido en reducción de costes debido a que el recurso humano requerido es menor, un aumento de la productividad y reutilización de componentes además se puede aumentar el nivel de abstracción a la hora de realizar el diseño de un software.

En resumen la utilización de modelos UML respecto a modelos escritos en Cassandra a la hora de especificar una base de datos no relacional proporciona una abstracción para aquellos desarrolladores que no estén muy familiarizados con el modelado de bases de datos no relacionales. La utilización de modelos diseñados en UML proporciona las siguientes ventajas: (1) UML es un lenguaje de modelado bien conocido por toda la comunidad. (2) La automatización de estos procesos nos permite crear software más rápido, más fiable y de mayor calidad lo que nos lleva a mantener buenas prácticas. Este trabajo pretende contribuir a satisfacer las carencias y virtudes citadas, proporcionando una herramienta que bajo las bases de un proceso de transformación dirigido por modelos transforma modelos UML y genera keyspaces para sistemas NoSQL orientado a columnas. Esperamos que esto permita a los equipos de desarrollo ahorrar esfuerzos y, por lo tanto, reducir costes.

En las siguientes secciones se desarrollan los siguientes aspectos: El apartado 1.2 expande información sobre la Ingeniería dirigida por modelos y el Desarrollo Dirigido por Modelos, este apartado es vital para entender todo lo relacionado con la memoria presente. El apartado 1.3 presenta la motivación y objetivos del proyecto. Finalmente el apartado 1.4 describe la estructura que tendrá el documento presente.

1.2. Antecedentes: Ingeniería Dirigida por Modelos y Desarrollo Dirigido por modelos

Según [6] la Ingeniería Dirigida por Modelos (MDE) puede ser definida como la *“técnica que hace uso, de forma sistemática y reiterada, de modelos como elementos primordiales a largo de todo el proceso de desarrollo. MDE trabaja con modelos como entradas al proceso y produce modelos como salidas del mismo”*. El concepto de Desarrollo Dirigido por Modelos (MDD) es como

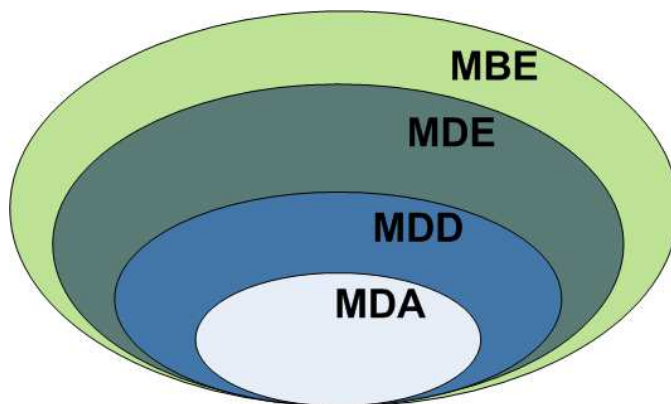


Figura 1.1: Capas del modelado de lenguajes

se puede observar en la Figura 1.1, un subconjunto de la Ingeniería Dirigida por Modelos (MDE). A diferencia de MDD,[7] MDE va más allá de las actividades de desarrollo puros y abarca otras tareas basadas en el proceso de modelado por ejemplo, aplicar ingeniería inversa a un modelo.

Tanto MDD como MDE aportan varias ventajas respecto a métodos de desarrollo tradicionales, podemos encontrar las siguientes ventajas [8], [9]:

1. Tiempo de desarrollo menor.
2. Reutilización de componentes en distintos sistemas.
3. Alto nivel de abstracción para escribir aplicaciones y artefactos de software a través de la arquitectura de niveles del meta-modelado y las capas de modelado de MDD. Este beneficio favorece diseñar una aplicación partiendo de lo mas general a lo más concreto, es decir, son independientes de la tecnología.
4. Un menor número de líneas de código escritas, ya que los niveles de abstracción que aporta MDD a través de los modelos y meta-modelos diseñados fomentan el reúso del código y de los modelos.
5. Si se centran los esfuerzos en el generador de código se puede mejorar la calidad del software, además reducimos la cantidad de errores así como el tiempo incurrido en pruebas e incurrimos en buenas practicas de desarrollo.

Una vez entendidos los conceptos anteriores y los beneficios que nos proporciona este enfoque de desarrollo se presentan los conceptos básicos que nos ayudaran a comprender como funciona MDD.

El enfoque MDD parte del meta-modelo [10], un meta-modelo es la representación del lenguaje que se desea implementar en sintaxis abstracta. Un lenguaje de modelado está formado de sintaxis y semántica. La sintaxis

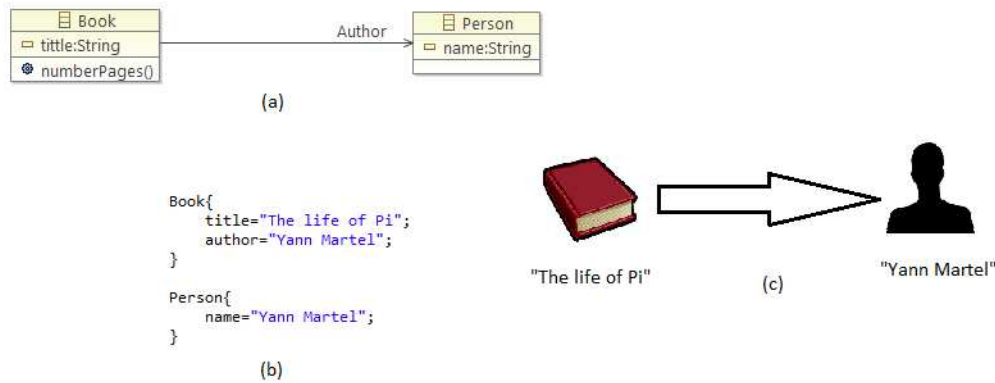


Figura 1.2: (a) Meta-modelo, (b) Sintaxis concreta, (c) Sintaxis grafica

son el conjunto de normas o reglas de escritura del lenguaje. La semántica refleja el significado de la sintaxis. La definición de la sintaxis abstracta forma parte del primer paso en el proceso de diseño de un lenguaje, esta sintaxis abstracta especifica la forma del meta-modelo. Clases y relaciones de dominio son los principales elementos que forman parte de un meta-modelo. A su vez los meta-modelos son considerados instancias de estos lenguajes de modelado. Un ejemplo de meta-modelo es la definición de UML [11]. En UML por ejemplo elementos como las clases, paquetes o atributos se pueden usar dentro del propio UML.

El siguiente paso consiste en la definición de la sintaxis concreta, este paso consiste en la representación de los elementos bien definidos en la sintaxis abstracta. Véase la creación de un modelo. Un modelo es la representación concreta de los elementos descritos en la sintaxis abstracta o meta-modelo. Por lo tanto un modelo siempre posee las propiedades y cumple las restricciones de su meta-modelo. En la figura 1.2 se pueden observar representaciones de los elementos descritos.

Según [10] y formalizando los conceptos anteriores estos son los pasos que hay que realizar para la creación de un nuevo lenguaje de modelado:

1. En primer lugar hay que establecer cuál va a ser la sintaxis de nuestro lenguaje de modelado. Para ello hay que crear un meta-modelo el cual como se comentaba anteriormente establece cuales son las reglas que han de seguir los modelos. El meta-modelo ha de ser construido utilizando un lenguaje de meta-modelado, el lenguaje de modelado utilizado en el presente proyecto es ECore [8]. Además de ECore hay otros lenguajes de modelado como puede ser MOF [9].
2. El siguiente paso consiste en la definición de la sintaxis concreta de nuestro lenguaje de modelado. La creación de un modelo bien definido a partir del meta-modelo creado en el paso anterior.

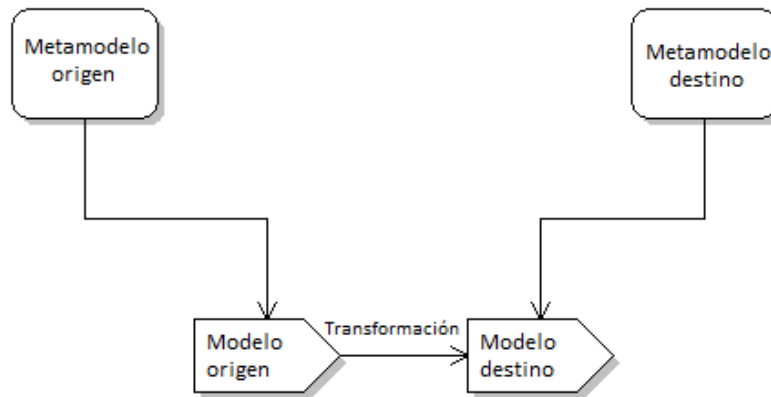


Figura 1.3: Proceso de transformación de un modelo

3. La definición de la semántica del lenguaje de modelado es el último paso, por ejemplo crear un generador que transforme los elementos del modelo en elementos de un lenguaje distinto bien definido.

La herramienta con la que se va a trabajar es EMF un plugin de eclipse que nos permite crear y trabajar con lenguajes de modelado. MDD proporciona las transformaciones de modelos como principal herramienta para trabajar con modelos en el proceso de desarrollo software. Existen dos tipos de transformaciones:

1. *Model to model (M2M)*. Dado un modelo de entrada esta transformación produce otro modelo como salida. Este tipo de transformación se puede utilizar de varias formas, la primera transformar un modelo dándole propiedades que no posee el original. La segunda es la que se utilizara en el presente proyecto, dado un modelo origen definido en UML realizar una transformación a un modelo destino definido en Cassandra (Figura 1.3). Esta información es ampliada en el siguiente capítulo.
2. *Model to text (M2T)*. Es la ultima transformación que se realiza en la etapa de desarrollo para la creación del generador de código de la aplicación. Estas transformaciones no generan un modelo de salida sino una representación textual del modelo que se desea transformar. Se puede utilizar para generar por ejemplo una representación del modelo transformado en formato HTML.

Para explicar todo este proceso se presenta un sencillo ejemplo donde se explican cómo se utilizan brevemente las técnicas de desarrollo dirigido por modelos, Model to Model (M2M) y Model to Text (M2T). Para aplicar

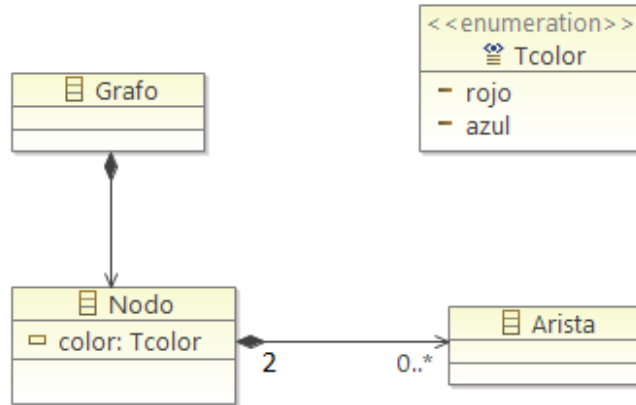


Figura 1.4: Meta-modelo de un grafo

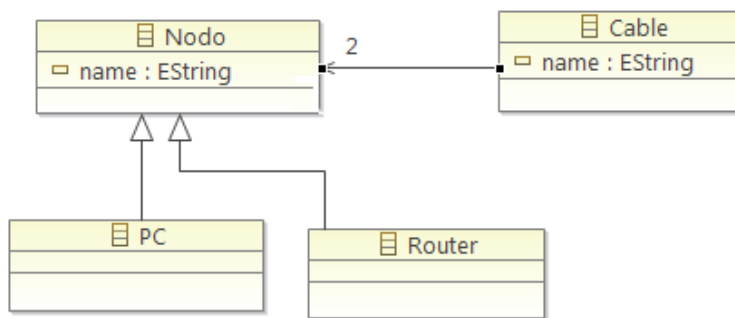


Figura 1.5: Meta-modelo de una red

M2M nos harán falta dos meta-modelos (uno origen y uno destino) y para aplicar M2T se ha diseñado un sencillo generador de código HTML.

La figura 1.4 muestra la representación de la sintaxis abstracta o meta-modelo del lenguaje. El meta-modelo consiste en un sencillo grafo el cual está compuesto de un número indefinido de nodos los cuales están conectados entre sí mediante aristas. Estos nodos como se ve en la imagen tienen un atributo de tipo "Tcolor" por lo que un Nodo puede ser de color rojo o de color azul. Por lo tanto un modelo bien definido de este meta-modelo será un modelo que contenga un grafo con un número cualquiera de Nodos conectados dos a dos entre sí mediante aristas.

A continuación se presenta otro meta-modelo como se ve en la figura 1.5. Este meta-modelo consiste en la representación de una red de computadores que está compuesta de PC's y Routers estos análogamente al anterior meta-modelo están conectados indistintamente entre sí mediante cables. Tanto los nodos como los cables tienen un atributo de tipo "String" para definir el nombre.

Conexiones

A1	PC1	Router1
A2	PC2	PC3
A3	Router2	PC4

Figura 1.6: Resultado transformación HTML

Nuestro objetivo es la transformación de un modelo de tipo Grafo a un modelo de tipo Red, dependiendo del color del Nodo, el Nodo será transformado en un PC o en un Router (Rojo-PC, Azul-Router) para realizar esta transformación se utilizan técnicas M2M.

Una vez obtenido el modelo de tipo Red deseamos representar textualmente el modelo generado para ello utilizamos técnicas M2T. En este caso la representación del modelo es realizada en formato HTML. La representación del código generado utilizando un modelo de tipo grafo y aplicando la transformación entre modelos es el que se puede observar en la figura 1.6.

El proceso que se ha seguido es explicado más detalladamente en capítulos posteriores, tanto las técnicas de transformación de modelos como la realización del generador de código así como la creación de los modelos.

1.3. Motivación y Objetivos

Como hemos visto a lo largo de la introducción el enfoque que proporciona el Desarrollo Dirigido por Modelos así como la Ingeniería Dirigida por Modelos nos proporciona múltiples ventajas respecto al desarrollo tradicional por lo que la utilización de este enfoque es apropiada para la realización del generador de código así como para las transformaciones entre modelos.

El objetivo de este proyecto de fin de carrera es la implementación de un generador de código que transforme modelos UML en su correspondiente código ejecutable en Cassandra. Para ello, previa implementación del generador de código habrá que transformar el modelo UML a un modelo escrito en Cassandra. Por lo tanto necesitaremos desarrollar un meta-modelo que describa el lenguaje de Cassandra y el meta-modelo de UML. Dicho generador y dicha transformación se desarrollaran utilizando el enfoque y las técnicas que proporcionan el Desarrollo Dirigido por Modelos. Esta implementación será realizada con el framework que proporciona eclipse de modelado llamado EMF.

Como resultado del proyecto se genera un código escrito en el lenguaje de Cassandra (CQL-Cassandra Query Language) que puede ser ejecutado en cualquier herramienta que soporte dicho lenguaje.

1.4. Estructura del Documento

Tras este capítulo de introducción, la presente memoria del Proyecto Fin de Carrera se estructura tal y como se describe a continuación: El Capítulo 2 describe en términos generales todos los conceptos y tecnologías que son necesarios para comprender el contenido de esta memoria conceptos tales como Cassandra o EMF por ejemplo. El Capítulo 3 describe como se ha realizado la transformación de modelos así como la metodología de transformación de modelos (M2M). El Capítulo 4 describe como se ha realizado el generador de código así como las técnicas y tecnologías utilizadas. Finalmente el Capítulo 5 presenta mis conclusiones tras la realización del proyecto así como posibles mejoras de la herramienta.

Capítulo 2

Antecedentes y Planificación

Este capítulo describe las tecnologías y técnicas utilizadas en el desarrollo del presente Proyecto de Fin de Carrera. La primera sección está dedicada a introducir el caso de estudio que se analizará a lo largo del proyecto presente, el cual consiste en la realización de un generador de código Cassandra. La siguiente sección está dedicada a describir en qué consiste Eclipse Modeling Framework (EMF) la herramienta utilizada para desarrollar lenguajes de modelado. A continuación se describe la herramienta Epsilon que se ha utilizado para desarrollar el generador de código. La siguiente sección está dedicada a explicar de manera breve algunos conceptos de Cassandra. Finalmente se expone la planificación que ha seguido el proyecto para su realización, desde formación hasta desarrollo.

Índice

2.1. Caso de estudio: Generador de código CQL-Cassandra	11
2.2. EMF	12
2.3. Epsilon	13
2.4. Cassandra	16
2.5. Planificación	19
2.6. Sumario	21

2.1. Caso de estudio: Generador de código CQL-Cassandra

En esta sección se presenta el caso de estudio que se analizará lo largo de este proyecto. El caso de estudio que se analizara trata sobre Twissandra, Twissandra es un proyecto creado para aprender como utilizar Cassandra. El modelo UML correspondiente a Twissandra se puede ver en la figura 2.1.

Twissandra es una versión simplificada de Twitter, que es a menudo utilizada para demostrar las capacidades de Cassandra. Twitter es una red

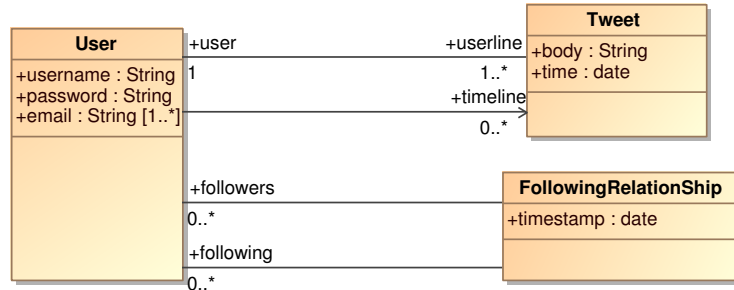


Figura 2.1: Modelo UML Twissandra

social muy conocida que permite escribir a los usuarios pequeños mensajes de texto, funciona de la siguiente manera: los usuarios registrados pueden publicar tweets. Un tweet es simplemente un texto con un límite de 140 caracteres publicado a una hora determinada. La colección de todos los tweets publicados por un usuario cronológicamente ordenados, están asignados a su Userline. Cada usuario registrado en Twissandra puede seguir a otros usuarios registrados. Cuando decimos que Pedro sigue a María significa que Pedro está interesado en saber qué publica María en su tablón, por lo que Pedro recibirá todos los mensajes que publique María en su tablón. Así, cada usuario registrado tiene también un timeline que vendría a ser la colección de todos los tweets de las personas a las que sigue ordenadas cronológicamente.

Los principales casos de uso de un sistema de este tipo son: (1) obtener el timeline de un usuario determinado; (2) obtener el Userline de un usuario, (3) obtener la lista de usuarios que un usuario está siguiendo; y (4) obtener la lista de usuarios que están siguiendo a un usuario específico. Estas dos últimas listas deben ser ordenadas cronológicamente.

Una vez entendido en los siguientes capítulos se procede a detallar el proceso de creación de un generador de código de este caso de estudio.

2.2. EMF

Eclipse Modeling Framework (EMF) [2.2] es un framework de modelado que nos proporciona la base para la elaboración de lenguajes de modelado. Para la creación de meta-modelos EMF [2.3] utiliza dos modelos de metadatos: Ecore y Genmodel. Ecore contiene la información sobre las clases que se han definido. Genmodel contiene información adicional para la generación del código, por ejemplo la ruta y la información del archivo. Genmodel también contiene atributos que sirven de control a la hora de generar el código, encontramos los siguientes parámetros de control:

1. EClass: representa una clase, con cero o más atributos y cero o más referencias.

2. EAttribute: representa un atributo que tiene un nombre y un tipo.
3. EReference: representa un extremo de una asociación entre dos clases.
4. EDataType: representa el tipo de un atributo, por ejemplo, int, float.

Estos atributos nos servirán a la hora de crear las reglas de transformación entre modelos UML y modelos Cassandra.

Utilizando EMF se pueden crear meta-modelos de forma gráfica muy similar a los diagramas de clases en UML. EMF permite crear un meta-modelo a través de diferentes medios, por ejemplo, XMI, anotaciones Java, XML o UML. Además EMF proporciona un framework para almacenar la información del modelo.

Un ejemplo sencillo de la utilidad de EMF es el siguiente: Imaginemos que deseamos construir una aplicación para manipular mensajes escritos en XML. El primer paso que daríamos sería empezar definiendo el schema del mensaje sin embargo con EMF se puede trabajar ignorando este nivel. Con EMF podemos crear plugins que generen por ejemplo un diagrama de clases UML a partir de este mensaje XML o directamente generar el código Java que implemente las clases del mensaje XML.

2.3. Epsilon

Epsilon es una familia de lenguajes y herramientas para el desarrollo de software dirigido por modelos, entre estas podemos encontrar: la transformación de modelos, validación de modelos, generación de código entre otras funcionalidades. Epsilon es distribuido a través de la plataforma de modelado de lenguajes de Eclipse. Epsilon proporciona multitud de lenguajes y herramientas para trabajar con modelos. Los lenguajes utilizados en este proyecto son EOL (Epsilon Object Language), ETL (Epsilon Transformation Language) y EGL (Epsilon Generation Language) también ha sido utilizado EUnit como herramienta para validar el código escrito por el generador de código Cassandra-CQL. Estos lenguajes y herramientas son descritos a continuación.

2.3.1. Epsilon Object Language

[2.1] Epsilon Object Language (EOL) es un lenguaje de programación imperativo utilizado para crear, consultar y modificar los modelos EMF. Según [2.1], EOL se puede considerar un lenguaje mezcla de Javascript y OCL, que combina lo mejor de ambos lenguajes. Como tal, proporciona todas las características habituales imperativas que se encuentran en Javascript (por ejemplo, la secuencia de la declaración, las variables, bucles for y while, etc) y todas las características interesantes de OCL como las operaciones sobre las colecciones (por ejemplo `Sequence1..5.select (x — x!3)`).

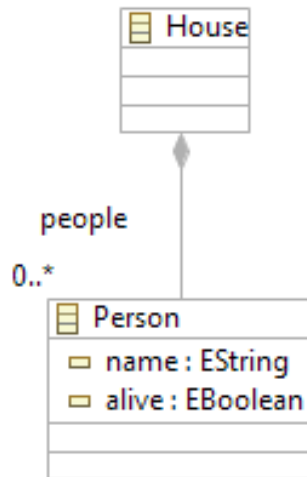


Figura 2.2: Ejemplo metamodelo casa

```

-----
for (person in Person.all){
    if (person.alive == true) {
        person.name.println();
    }
}

//Podemos realizar lo mismo de la siguiente manera:

Person.all.select(r|r.alive==true).name.println();
-----

```

Figura 2.3: Ejemplo código EOL

Para entender mejor el funcionamiento de EOL se expone el siguiente ejemplo. Se ha definido el meta-modelo mostrado en la figura 2.2 el cual consiste en la representación de una casa y las personas que viven en ella. Las personas tienen un nombre y un atributo booleano que representa si una persona está viva o no. Existe una relación de agregación para reflejar que la casa contiene personas. Una vez creado el meta-modelo podemos crear un modelo como instancia de ese meta-modelo. Un ejemplo de cómo funciona EOL puede ser el siguiente: deseamos saber que personas habitan en la casa y están vivas. La sintaxis correspondiente sería la siguiente (figura 2.3).

Como vemos la sintaxis es muy similar a cualquier lenguaje orientado a objetos, podemos manipular y consultar los objetos del modelo, sin embargo

EOL no nos permite la definición de clases.

2.3.2. Epsilon Transformation Language

[2.1] Epsilon Transformation Language (ETL) es un lenguaje de transformación modelo a modelo basado en reglas (Model to Model-M2M). ETL proporciona las características estándar de un lenguaje de transformación, también nos permite manipular los modelos de entrada y salida así como su código fuente. ETL tiene su propia sintaxis sin embargo utiliza el lenguaje EOL como base.

Recordando el ejemplo de la Red de computadores y el Grafo detallado en el capítulo anterior (sección 1.2). Deseamos realizar la transformación de un modelo de tipo grafo a un modelo de tipo red para ello definiremos una serie de reglas de transformación utilizando para ello ETL. En primer lugar necesitamos definir el modelo del grafo para poder realizar la transformación a un modelo de tipo Red. La figura 2.4 muestra el código que realiza el proceso de transformación de un Grafo a una Red.

La sección de código inicial define algunas variables previa ejecución de las reglas. En ella se inicializan variables que se utilizarán posteriormente. En este código encontramos solo una regla. Esta regla transforma aristas del grafo a cables de la red. La primera instrucción copia el nombre de la arista al cable. A continuación la primera condicional cuestiona si esa arista tiene un padre definido en caso afirmativo asigna el cable a la red. A continuación por cada nodo se crea o bien un PC o un router dependiendo del color del nodo que se esté analizando (rojo-PC, azul-Router). En siguiente lugar se asigna el nodo creado al cable correspondiente y finalmente se añade a la red. Este proceso se repite por cada arista del grafo. Una vez ejecutado este código dado un modelo de entrada de tipo Grafo obtenemos un modelo de tipo Red que cumple las reglas definidas en el meta-modelo.

2.3.3. Epsilon Generation Language

[2.1] [2.4] Epsilon Generation Language (EGL) es un lenguaje utilizado para la generación de código basado en la transformación de modelos (Model to Text-M2T). EGL puede ser utilizado para transformar los modelos en varios tipos de lenguajes, por ejemplo código ejecutable Java, código HTML o incluso aplicaciones completas que comprenden el código en varios lenguajes (por ejemplo, HTML, Javascript y CSS). En este proyecto se utilizará EGL para la generación de código Cassandra Query Language (CQL) a partir de modelos UML.

Cada plantilla de EGL contiene varias secciones. Cada sección puede ser estática o bien dinámica. Una sección estática contiene texto que aparecerá en la salida generada por la plantilla. Una sección dinámica comienza con la secuencia '[%' y termina con la secuencia '%]'. La sección dinámica

```

pre {
    "Running ETL".println();
    var Red : new Red!Red;
    Red.nameRed="Red1";
    var iPC:new Integer=1;
    var iRouter:new Integer=1;
}

rule Arista2Cable
transform a : Grafo!Arista
to r : Red!Cable {
    r.nameCable = a.nombreArista;
    if (a.parent.isDefined()) {
        r.parent=Red;
        for (nodoArista in a.children) {
            if(nodoArista.color=TColor#R){
                var PC : new Red!PC;
                PC.nameNodo="PC"+iPC;
                r.children.add(PC);
                iPC=iPC+1;
            }
        }
    }
    else{
        var Router : new Red!Router;
        Router.nameNodo="Router"+iRouter;
        r.children.add(Router);
        iRouter=iRouter+1;
    }
}
}

```

Figura 2.4: Ejemplo código ETL

contiene el lenguaje (EOL, Epsilon Object Language).

La figura 2.5 muestra como se realiza la generación de código HTML utilizando para ello el modelo generado de una red a partir de un grafo (ver sección anterior).

Como vemos en el código la integración del código EGL junto con HTML es total, en este sencillo código se genera una página HTML con una tabla que muestra varias filas, una fila por cada conexión entre dos componentes de la red.

2.4. Cassandra

Esta sección trata sobre Cassandra, desde su historia, arquitectura, sintaxis y otras características que explican como es el sistema de bases de datos escogido para este proyecto de fin de carrera.

```

-----
[%
  var red: Red := Red.allInstances().at(0);
%]

<html>
  <head>
    <title> Red </title>
  </head>
  <body>
    <h1>Conexiones</h1>
    <table border="1">
      <col style="width: 200px" />
      <col style="width: 100px" span="3" />
      [% for (conexiones in red.conexiones){%]
      <tr>
        <th scope="row">[%=conexiones.nameCable%]</th>
        [% for (nodos in conexiones.children){%]
          <td>[%=nodos.nameNodo%]</td>
        [% }%]
      </tr>
      [% }%]
    </table>
  </body>
</html>
-----

```

Figura 2.5: Ejemplo código EGL

Desde que nació SQL en el año 1974, SQL ha sido el modelo de base de datos relacionales utilizado por excelencia. En los últimos años ha surgido otra vertiente denominada NoSQL la cual surge por la necesidad de manejo de grandes volúmenes de información no estructurada, distribuida y accedida con la mayor rapidez posible. NoSQL es literalmente la combinación de dos palabras: No y SQL, este término es utilizado en la actualidad como "Not Only SQL", hoy en día se utiliza para denominar todas las bases de datos que no siguen los principios de los sistema de gestión de bases de datos relacionales tradicionales. Es usado en plataformas como Facebook o Twitter. Podemos encontrar varios tipos de bases de datos no relacionales: Bases de datos documentales, bases de datos clave-valor, orientadas a grafos entre otros.

Existen varias diferencias entre estos sistemas NoSQL y sistemas SQL, las principales características de NoSQL que difieren de SQL son:

1. NoSQL no garantiza las propiedades ACID (atomicidad, coherencia, aislamiento y durabilidad).
2. No utiliza SQL como lenguaje de consultas. Algunas bases de datos

no relacionales utilizan SQL como lenguaje de apoyo sin embargo la mayoría utilizan su propio lenguaje de consultas como por ejemplo Cassandra que utiliza CQL.

3. No está permitido el uso de joins ya que al manejar grandes volúmenes de información una consulta con un join puede llegar a sobrecargar el sistema.
4. Escalan horizontalmente y trabajan de manera distribuida por lo que la información puede estar en distintas maquinas y el añadir nodos mejora el rendimiento.
5. Resuelven problemas de altos volúmenes de información

En este proyecto de fin de carrera se utiliza como sistema de bases de datos Cassandra, Cassandra es una distribución de bases de datos no relaciones (NoSQL). Cassandra fue diseñado por Avinash Lakshman (uno de los creadores de Amazon's Dynamo) y Prashant Malik (Ingeniero de Facebook), está en producción para Facebook sin embargo aun está en fase de desarrollo. Dentro del mundo de las bases de datos no relaciones Cassandra pertenece a la familia de bases de datos denominada clave-valor. Las bases de datos clave-valor son aquellas que asocian los valores a una determinada clave esto permite la recuperación y escritura de información de forma muy rápida y eficiente. Cassandra reúne las tecnologías de sistemas distribuidos de Amazon Dynamo y el modelo de datos BigTable de Google. Al igual que Dynamo, Cassandra es consistente. Como BigTable, Cassandra proporciona un modelo de datos basado en column families siendo considerado el sistema basado en clave-valor más popular. Para la manipulación y administración de los datos se utiliza OpsCenter que ofrece una interfaz de gestión y administración para los cluster Cassandra.

Las bases de datos basadas en Cassandra son soluciones utilizadas cuando es necesaria escalabilidad y alta disponibilidad sin comprometer el rendimiento. Escalabilidad lineal y tolerancia a fallos o infraestructura en la nube lo convierten en la plataforma idónea para datos de misión crítica. Cassandra proporciona gran estabilidad en cuanto a la replicación de datos a través de múltiples datacenters, consiguiendo una menor latencia para sus usuarios y la tranquilidad de que no existan pérdidas de datos ante caídas. Cassandra utiliza un lenguaje de consultas llamado CQL (Cassandra Query Language) con una sintaxis muy similar a SQL aunque con menos funcionalidades.

A continuación se explican brevemente algunos términos que hay que tener en cuenta a la hora de trabajar con Cassandra. En Cassandra un keyspace es el equivalente a un schema en los sistemas de bases de datos relacionales. El conocido término de tabla en las bases de datos relacionales tiene su equivalente en Cassandra llamado column family, las column families se guardan en ficheros separados y son ordenadas por su key. Una

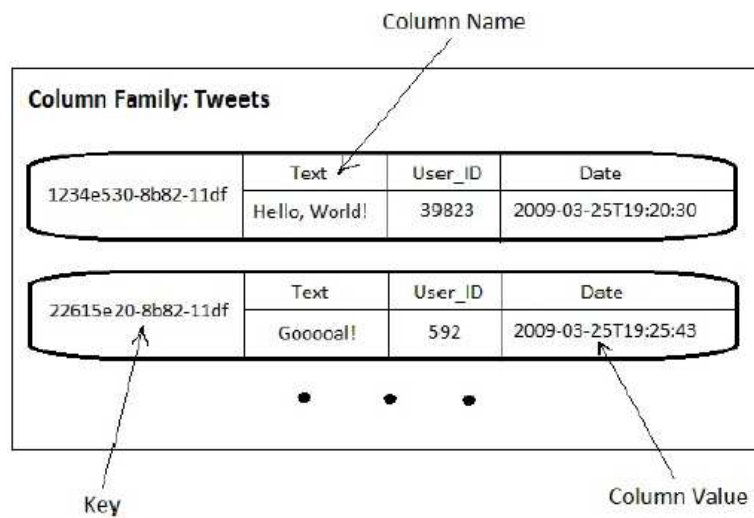


Figura 2.6: Estructura column family

columna es la unidad de almacenamiento básica, está formada de tres campos: Nombre, valor y timestamp. El nombre y el valor se almacena como una matriz de bytes sin procesar y pueden ser de cualquier tamaño. Los tres valores anteriores son introducidos por el cliente, incluido el timestamp. Un ejemplo de la estructura de una columna: Por lo tanto un keyspace puede

Nombre de la columna	Username
Valor de "Username"	Ignacio
Timestamp	123456789

Cuadro 2.1: Ejemplo estructura columna

contener varias column families y una column family a su vez contiene varias columnas. La estructura de una column family queda como se puede observar en la figura ??.

En cuanto a la sintaxis del lenguaje CQL es muy similar a SQL, CQL contiene sintaxis ya conocida de SQL como INSERT, DELETE, UPDATE, INSERT, ... La Figura 2.7 refleja un ejemplo breve de código CQL ejecutable.

Conceptos relacionados con Cassandra son ampliados en el siguiente capítulo a la hora de definir el meta-modelo de Cassandra.

2.5. Planificación

Como se ha comentado en el presente documento, el objetivo de este proyecto de fin de carrera es la implementación de un generador de código

```
DROP KEYSPACE twitter;

CREATE KEYSPACE twitter
WITH replication = {'class':'SimpleStrategy', 'replication_factor':2};

USE twitter;

CREATE TABLE Tweet(
    UUID text,
    usernameTw text,
    body text,
    PRIMARY KEY(UUID)
);

CREATE TABLE User(
    username text,
    password text,
    followers set<text>,
    followings set<text>,
    tweets_written list<text>,
    PRIMARY KEY(username)
);
```

Figura 2.7: Ejemplo código CQL

Cassandra a partir de modelos UML. El proceso de desarrollo así como el de aprendizaje que se ha seguido para la realización del proyecto queda reflejado en la figura [figuraProceso].

La primera tarea como es evidente consistió en adquirir los conocimientos necesarios para el desarrollo del proyecto, en primer lugar todo lo relacionado con el proceso de modelado de un lenguaje y transformación de lenguajes [kleppe]. También fueron necesarios conocimientos sobre la Ingeniería y el Desarrollo Dirigido por Modelos, también sintaxis y arquitectura de Cassandra.

A continuación se comenzó a trabajar con la herramienta Epsilon, sus lenguajes EOL, EGL, ETL y EUnit como herramienta para las pruebas de los modelos y código generados. Así como con el lenguaje para la definición de lenguajes de modelado Eclipse Modeling Framework (EMF). Para conocer cómo funcionaban estos lenguajes se desarrollaron una serie de casos prácticos para familiarizarse con los métodos de transformación así como con la herramienta y creación de casos de prueba, estos casos de prueba son los que se exponen en la sección de Epsilon.

Una vez conocido estos conceptos se estudiaron las reglas de transformación a aplicar para transformar un modelo UML a un modelo Cassandra, estas reglas fueron propuestas por [pabloCassandra].

Tras estas tareas de adquisición de conocimientos se empezó a trabajar

en el generador de código Cassandra empezando por la transformación de modelos UML a modelos Cassandra. Estas transformaciones son expuestas en el siguiente capítulo. Una vez finalizada la transformación se comenzó a trabajar en el generador de código Cassandra, esta tarea es descrita en el capítulo 5. Tras realizar dicha tarea se realizaron una serie de casos de prueba para verificar si el resultado del generador de código era el esperado utilizando la herramienta EUnit. Una vez desarrollado el generador de código se realizaron distintos casos de prueba sobre Cassandra para probar el correcto funcionamiento del generador de código.

Finalmente y tras generar varios casos de ejemplo se instaló la plataforma de Cassandra BLABLA

2.6. Sumario

Durante este capítulo se han descrito los conceptos necesarios para lograr comprender el ámbito y el alcance de este proyecto, se ha descrito el caso de estudio planteado en el proyecto. También se ha hablado sobre tecnologías implicadas en el desarrollo del generador de código, así como de Cassandra y su arquitectura, Epsilon los lenguajes utilizados y herramientas utilizadas. El siguiente capítulo describe los primeros pasos para la realización del generador de código, el primer paso consiste en la definición de reglas de transformación entre modelos llamadas "Model to Model" (M2M). Además se continúa el caso de estudio planteado en este capítulo, se define el meta-modelo de Cassandra y se definen las reglas de transformación entre modelos UML y Cassandra.

Capítulo 3

Transformación modelo a modelo

Este capítulo describe el primer paso del proceso de transformación de acuerdo al proceso de desarrollo dirigido por modelos. El principal objetivo de esta fase es la transformación de un modelo definido en UML a un modelo en Cassandra. La primera fase consiste en la definición de un meta-modelo que defina el modelado en Cassandra. En segundo lugar debemos definir las reglas de transformación para convertir un modelo UML a un modelo en Cassandra utilizando para ello el lenguaje Epsilon Transformation Language (ETL). A continuación se presenta un caso de estudio en el cual se van a aplicar las reglas de transformación ETL que se han definido. Finalmente el sumario con las conclusiones de este capítulo.

Índice

3.1. Introducción	23
3.2. Metamodelo Cassandra	24
3.3. Transformación de Modelo UML a Cassandra	26
3.4. Caso de estudio transformación modelo a modelo	29
3.5. Sumario	31

3.1. Introducción

A la hora de desarrollar el generador de código necesitamos establecer en primer lugar cuales van a ser los modelos de origen y de salida, en este proyecto de fin de carrera y como se especifica en capítulos anteriores nuestro objetivo consiste en transformar un modelo UML en modelos Cassandra para ello es necesario contar con un meta-modelo definido en Cassandra el cual es definido en la siguiente sección, en cuanto al meta-modelo origen basado en UML 2.0 utilizaremos el que nos proporciona la propia plataforma Epsilon.

El siguiente paso consiste en establecer una serie de reglas de correspondencia entre el modelo de entrada y el modelo de salida. Estas reglas han de contemplar los distintos tipos de elementos que pueden aparecer en ambos lenguajes, así como elementos que no aparecen en uno de los dos lenguajes y tiene importancia en el otro. En este caso, consiste en establecer reglas de correspondencia entre elementos del modelado UML 2.0 y elementos del modelado Cassandra, para así crear el generador de código Cassandra Query Language (CQL). Para la definición de estas reglas se ha utilizado el lenguaje Epsilon Transformation Language (ETL) y el lenguaje Epsilon Object Language (EOL) los cuales nos proporciona la plataforma Epsilon. Tras implementar estas reglas y comprobar que son funcionales se puede implementar el generador de código pero este tema se trata en el siguiente capítulo.

3.2. Metamodelo Cassandra

Previo paso a la definición de las reglas de transformación entre modelos necesitamos definir un meta-modelo de origen y un meta-modelo de destino. Como habíamos descrito en el capítulo anterior la base del proceso de transformación entre modelos parte del meta-modelo. El meta-modelo de UML utilizado como origen en este proyecto es el que nos proporciona Epsilon, dicho meta-modelo sigue el estándar de UML 2.0. A parte del meta-modelo de UML nos hace falta un meta-modelo que defina el lenguaje de Cassandra. Dicho meta-modelo fue proporcionado por Pablo Sánchez Barreiro. Este meta-modelo se construyó por medio de la abstracción de las características de Cassandra. El meta-modelo sufrió algunos cambios respecto al inicial debido a exigencias y variantes que sufrió el proyecto. La Figura 3.1 muestra un diagrama de clase UML que representa el modelo de datos. Este tipo de modelo es el origen del proceso de transformaciones entre modelos que queremos crear. A continuación se detallan los aspectos mas importantes del meta-modelo de Cassandra.

De acuerdo con la estructura de Cassandra, el elemento raíz de cualquier esquema orientado a columnas es el `keyspace`, el `keyspace` es el equivalente a la base de datos en el modelo relacional, la meta-clase `keyspace` cuenta con los meta-atributos: `nombre` utilizado para denominar el `keyspace`, `[cassandra] replicationFactor` que representa el número de servidores de Cassandra de los que se debe guardar un registro u obtener una respuesta al recuperar algún registro y `replicaPlacementStrategy` es la estrategia de replicación que se va a tomar, como estrategias tenemos tres tipos:

1. `SimpleStrategy`: Es la estrategia de replicación utilizada por defecto al crear un `KeySpace` utilizando Cassandra. Es utilizada para clústeres de datacenters simples.

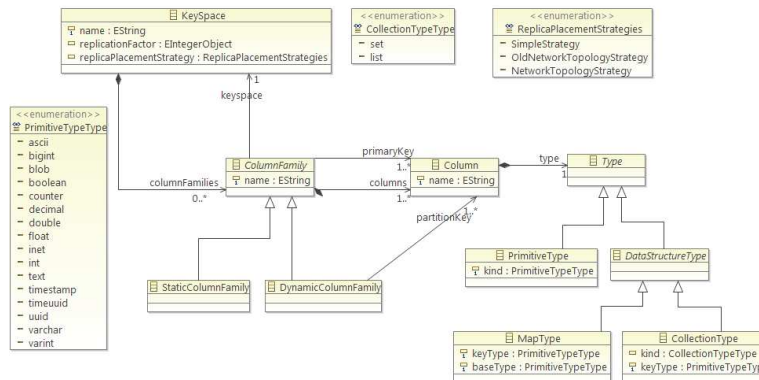


Figura 3.1: Metamodelo Cassandra

2. NetworkTopologyStrategy: Utilizada cuando se tiene (o se va a tener) el clúster desplegado a través de múltiples data centers. Esta estrategia especifica cuántas réplicas se desean en cada data center.
3. OldNetworkTopologyStrategy: Se utiliza para proporcionar retro compatibilidad con instalaciones Cassandra antiguas.

A continuación tenemos la meta-clase ColumnFamily equivalente a las tablas en el modelo relacional, dentro de esta meta-clase encontramos dos tipos: las StaticColumnFamily y las DynamicColumnFamily. Recordamos que las static column family son el equivalente a las tablas en el modelo relacional, sin embargo las dynamic column family son utilizadas para la recuperación de datos eficiente, algo similar a las vistas del modelo relacional.

A la hora de definir las claves de las column families hay que tener en cuenta el orden, en CQL el orden de definición de las claves importa, en la primera columna se define la partition key, esta tiene la propiedad de que todas las filas que comparten la misma partition key se almacenan en el mismo nodo físico. Además, la inserción, actualización o eliminación de filas que comparten la partition key para una column family determinada se realizan de forma atómica. Es posible tener una partition key compuesta, es decir, una partition key formada por varias columnas, en CQL esto se define utilizando paréntesis para delimitar el conjunto de partición. En la siguiente columna se define la clustering key utilizada para la recuperación de filas de manera eficiente. Aunque en siguientes secciones esta información se amplía, el resumen de la definición de column families dinámicas es la siguiente: PRIMARY KEY((partitioning key_1, ... partitioning key_n), clustering key_1 ... clustering key_n)

La siguiente meta-clase es Column, esta meta-clase contiene los valores que se almacenan en las Column Families, estas Columns tienen como meta-atributos un nombre y un tipo de dato. Dentro de los tipos de datos que

pueden darse en una columna encontramos dos tipos, `PrimitiveType` y `DataStructureType`, el primero define los tipos primitivos, por ejemplo entero, texto, uuid, etc.. `DataStructureType` define las colecciones. Estas pueden ser de dos tipos: `MapType` o bien `CollectionType`, ambas colecciones tienen un meta-atributo llamado `KeyType` para definir el tipo primitivo que utilizan. `MapType` cuenta con un meta-atributo llamado `baseType` de tipo primitivo que define el segundo tipo de dato del mapa. Dentro de `CollectionType` encontramos un meta-atributo llamado `kind` que define el tipo de colección que vamos a utilizar, esta puede ser o bien tipo `set` o tipo `list`. Más adelante se explica que características reúne cada colección y mapa y cuando se utilizan.

3.3. Transformación de Modelo UML a Cassandra

Una vez definido el meta-modelo de Cassandra utilizamos Epsilon Transformation Language (ETL) para establecer las reglas de transformación entre modelos UML y modelos Cassandra. Recordamos que el lenguaje ETL es el lenguaje que utiliza Epsilon para la transformación entre modelos basado en reglas. A continuación se presenta una explicación de las reglas junto con el código correspondiente. Las reglas de transformación son definidas en [docPablo]. A continuación se detalla cómo se han implementado dichas reglas en el lenguaje de definición de reglas de transformación ETL.

3.3.1. Transformación del modelo

El modelo UML se considera el elemento raíz que contiene los elementos del modelo, este se transforma en el keyspace para el repositorio de Cassandra. El nombre del keyspace corresponderá al que se haya escogido para el modelo de datos UML. Los Key Spaces permiten agrupar entidades tales como Column Families, Columns.. De esta forma, se pueden tener varios Key Spaces en el mismo proyecto independientes entre si.

3.3.2. Transformación de clases

Se han definido dos reglas ETL que transforman una clase definida en UML a una Column Family estática de Cassandra (el equivalente a una tabla en el sistema de bases de datos relacionales). La primera regla es definida para las clases que tengan un atributo clave definido, para ello se utiliza la propiedad `isID` de UML la cual define que ese atributo identifica a la clase de forma única. La segunda regla es para las clases sin atributo clave definido. El proceso de transformación es similar en ambas reglas, en primer lugar se transforma la clase UML a una column family de Cassandra, se asigna la column family al keyspace que le corresponde, se copia el nombre de la clase a la column family y se añade al conjunto de column families del

keyspace. En la segunda regla utilizada para las clases sin atributo clave, se crea un atributo que va a ser la clave de esa column family ya que esta no tiene, dicha clave tendrá de nombre, el nombre de la clase más el distintivo `_ID` y será de tipo `uuid`. Dichas reglas están definidas en el punto 5.3 del documento [docPablo].

3.3.3. Transformación de atributos

La siguiente regla define como se realiza la transformación de un atributo del modelo UML a una columna del modelo Cassandra. La definición básica es que un atributo UML corresponde a una columna Cassandra. Para ello en primer lugar se define una guarda ETL para que la transformación se haga solo de los atributos de las clases y no de los atributos de la relación. A continuación se realiza un filtro para evitar la adición de columnas que no pertenezcan al modelo de datos. Una vez hecho esto hay que diferenciar dos tipos de atributos, aquellos cuya multiplicidad sea igual a 1 y aquellos atributos con multiplicidad mayor de 1. En cuanto a los atributos de multiplicidad igual a 1 se realiza la transformación del atributo a columna copiando su tipo de dato. Respecto a los atributos de tamaño mayor de 1 se aplica la siguiente regla: Aquellos atributos que se hayan definido en el modelo UML como únicos y no ordenados son transformados como tipo set de Cassandra. En caso contrario se definen como tipo list. Una vez transformado en set o list, en ambos casos se realiza la transformación correspondiente del tipo primitivo UML a Cassandra (viene descrita en la sub-sección inferior), se realiza una copia del nombre y se añade la columna ya transformada a la column family correspondiente. Finalmente en caso de que el atributo sea clave de la clase se añade a la column family como primary key. El código correspondiente a esta regla es el que se puede observar en la figura 3.2

3.3.4. Transformación de asociaciones

La siguiente regla define como se realiza la transformación de los atributos de las asociaciones entre clases UML. En primer lugar hay que diferenciar como se tratan las asociaciones, existen dos tipos: aquellas asociaciones cuyo extremo tiene una cardinalidad igual a uno y aquellas con una cardinalidad mayor de uno.

Para los extremos con cardinalidad igual a uno se crea una columna que será añadida en la column family del otro extremo de la relación, esta columna nueva es una copia de la primary key de la column family fuente, el nombre de la nueva columna es la composición del nombre de la column family y del nombre de la primary key, el tipo de la nueva columna es el mismo del de la primary key de la column family fuente. Esto se comprende mejor con el caso de estudio propuesto en la sección siguiente.

Para los extremos con cardinalidad mayor de uno, se crea una column

```

-----
//5.5 Attribute with primitive type transformation
rule Attribute2Column
transform attribute : UML!Property
to column : nosql!Column {
    guard: (("+"attribute.qualifiedName).contains("Data::") and attribute.type.isKindOf(UML!PrimitiveType))

    //filtramos para evitar añadir columnas ajenas al modelo de datos
    for(cfamil in kspace.columnFamilies){

        if(attribute.qualifiedName=="Data::"+cfamil.name+"::"+attribute.name){

            //5.5 Attribute with primitive type transformation->Attributes with upper bound = 1
            if(attribute.upper=1){
                //transformacion del atributo en una columna basica
                var type : new nosql!PrimitiveType;
                type.kind=umlType2modelType(attribute.type.name);
                column.type=type;
            }
            //5.5 Attribute with primitive type transformation->Attributes with upper bound > 1
            else if(attribute.upper<>0){
                //transformacion del atributo en un set o list
                var ctype : new nosql!CollectionType;

                if(attribute.isUnique and not attribute.isOrdered)//set
                    ctype.kind=nosql!CollectionType#set;
                else//list
                    ctype.kind=nosql!CollectionType#list;

                ctype.keyType=umlType2modelType(attribute.type.name);
                column.type=ctype;
            }

            column.name=attribute.name;
            cfamil.columns.add(column);

            //5.1 Assignment of keys to classes
            if(attribute.isID)
                cfamil.primaryKey.add(column);
        }
    }
}
-----

```

Figura 3.2: Regla de transformación XX

family dinámica. Recordamos que la primary key de este tipo de column family esta compuesta de dos tipos de columna: La partition key y la cluster key. La transformación utilizada para este tipo de column family es la siguiente: Esta column family dinámica está compuesta de dos columnas, la primera columna y segunda columna juegan el papel de partition key de la

column family, la primera columna se construye de la misma manera que el caso anterior, se copia el nombre y el tipo de la primary key de la column family extremo de la asociación fuente. La segunda columna de la misma manera toma los datos nombre y tipo de la primary key del otro extremo de la asociación. El rol de cluster key lo desempeña la primera columna definida. El nombre de la column family será la concatenación del nombre de las column family de ambos extremos de la asociación (primero fuente, segundo extremo fin de la asociación). En resumen la transformación de la primary key en las column family dinámicas es la siguiente:

1. Partition Key: Primera columna (fuente de la asociación), segunda columna (extremo de la asociación).
2. Clustering Key: Primera columna (fuente de la asociación).

3.3.5. Transformación de tipos primitivos

En cuanto a la transformación de variables de tipo primitivo se define una operación básica de correspondencia de esta manera podemos convertir un tipo primitivo UML a su equivalente en Cassandra. Las correspondencias básicas serían las siguientes:

UML	Cassandra
string	text
int	int
date	timestamp
uuid	uuid
float	float
double	double
boolean	boolean
char	varchar

Cuadro 3.1: Equivalencias tipos primitivos UML-Cassandra

3.4. Caso de estudio transformación modelo a modelo

Como se explicaba en el capítulo 2 en la sección Caso de estudio.^{el} objetivo de este caso de estudio es la creación de un generador de código de una versión simplificada de Twitter. En esta sección se reproducirán los procesos M2M y M2T en el siguiente capítulo, todo esto bajo el proceso de desarrollo dirigido por modelos. Esta sección esta dedicada a describir la transformación del modelo UML de Twissandra a un modelo Cassandra. Partiendo del

modelo UML de la figura 2.1 y una vez establecidas las reglas de transformación entre modelos, esta sección explica el proceso de transformación del modelo UML al modelo Cassandra de este caso de estudio.

Por cada paquete estereotipado como «dataModel», se crea un nuevo keyspace. El nombre del keyspace será el nombre utilizado por el modelo de datos. Los atributos restantes de las meta-clases del keyspace se establecen en sus valores definidos por defecto. A continuación, todos los elementos correspondientes de ese paquete se procesan y se colocan dentro de su keyspace correspondiente.

Marcamos el atributo username de la clase User como clave ya que el parámetro isID del modelo UML esta marcado como true. En el caso de las clases FollowingRelationship y la clase Tweet al no tener un atributo marcado como clave generamos dos columnas clave automáticamente para cada clase, llamadas FollowingRelationship_id y tweet_id respectivamente. La clase User del modelo UML es transformada en una column family llamada User. Una vez procesadas las clases UML y transformadas a su correspondiente column family se proceden a transformar los atributos y asociaciones. De manera similar para aquellos atributos del modelo UML cuya multiplicidad sea igual a uno se realiza una transformación simple, por ejemplo el atributo username y password se transforman en dos columnas, ambos del tipo text. Estas columnas están contenidas en la column family User. De la misma manera se transforman los atributos body y time de la clase Tweet y el atributo timestamp de la clase FollowingRelationship. En el caso de el atributo del modelo UML email cuya multiplicidad es mayor de uno y tiene las propiedades isUnique establecida en false y la propiedad isOrdered establecida en false (en el modelo no se puede apreciar pero esta configurado así), se transforma este atributo en un set llamado email de tipo text dentro de la column family User.

En cuanto a las asociaciones de multiplicidad igual a uno la transformación que se realiza por ejemplo, para la asociación de la clase user una nueva columna llamada user_username (recordemos que username es la clave de la column family user) es creada y añadida a la column family Tweet. Para las asociaciones cuya multiplicidad es mayor de uno, por ejemplo la asociación llamada userline se crea una dynamic column family llamada User_userline. A continuación una columna llamada user_username de tipo text es añadida a esta column family. Después una columna llamada tweet_id de tipo uuid es añadida (el atributo tweet_id fue creado en la column family tweet al no tener clave). Las columnas user_username y tweet_id son designadas como primary key, la columna user_username será la partition key.

3.5. Sumario

Durante este capítulo se ha descrito la metodología "Model to Model" (M2M) utilizada. En primer lugar se ha realizado una descripción de la construcción del meta-modelo de Cassandra. A continuación se han presentado cuales son las reglas que se han seguido a la hora de transformar cada uno de los elementos de un modelo UML a un modelo Cassandra. Para ello se han definido una serie de reglas escritas en el lenguaje Epsilon Transformation Language (ETL). La siguiente sección está dedicada a continuar el caso de estudio desarrollado a lo largo del proyecto relacionado con Twissandra, en esta sección se finaliza el generador de código. Finalmente se definen como se han realizado los casos de prueba así como los resultados.

Capítulo 4

Transformación modelo a texto

El capítulo anterior describió el funcionamiento y desarrollo del primer paso del proceso de transformación dirigido por modelos, 'Model To model'(M2M). En este capítulo se presenta el siguiente paso del proceso de transformación dirigido por modelos, dicho paso consiste en la transformación del modelo a texto, este paso es mas conocido como 'Model To Text'(M2T). En este capitulo se explica como se ha realizado la implementación del generador de código utilizando para ello el lenguaje Epsilon Generation Language (EGL). Este capítulo describe el funcionamiento y desarrollo de los generadores de código, los cuales tienen como objetivo adaptar la implementación de referencia a las necesidades de cada cliente.

Índice

4.1. Introducción	33
4.2. Generador de código	34
4.3. Caso de estudio Twissandra	34
4.4. Pruebas	36
4.5. Despliegue	36
4.6. Sumario	36

4.1. Introducción

Una vez establecidas las reglas de correspondencia entre ambos modelos empezamos a desarrollar el generador de código. El código generado es código Cassandra Query Language (CQL) que puede ser ejecutado en herramientas que soporten dicho lenguaje. A continuación tras implementar el generador de código el siguiente paso consiste en la implementación de una serie de casos de prueba para comprobar el correcto funcionamiento del generador

de código. Para la implementación de este generador de código se ha utilizado el lenguaje Epsilon Generation Language (EGL).

Contar algo que tienen las siguientes secciones.

4.2. Generador de código

Esta sección analiza el proceso de creación del generador de código. Para crear este generador de código se utiliza el lenguaje Epsilon Generation Language (EGL), como se comentaba en el capítulo 2 es un lenguaje utilizado para la generación de código basado en la transformación de modelos, el lenguaje a generar es Cassandra Query Language (CQL) este lenguaje de consultas es muy similar a SQL, sin embargo existen pequeñas diferencias que se han citado a lo largo del proyecto por ejemplo: la definición de claves, tipos de dato etc. En la figura 4.1 se muestra parte del código del generador de código. Esta sección de código muestra código EGL. En primer lugar se realiza la definición del keyspace para ello hay que definir el nombre del keyspace que parte del modelo transformado Cassandra y la estrategia de replicación (ver capítulo 3, sección 2). A continuación se realiza la creación de una column family en CQL. En primer lugar se definen las columnas de la tabla. Por cada columna de la column family se obtiene el nombre de la columna y se define el tipo de dato (primitivo, map, set/list) junto con su nombre. El resto de código se ha omitido por razones de espacio sin embargo se detalla a continuación. Tras la definición de la columna se define la primary key. Para la definición de la primary key hay que diferenciar si la column family es dinámica o estática. En caso de ser estática la definición sería la clásica: `PRIMARY KEY(ColumnaClave)`. En el caso de tratarse de una column family dinámica la construcción modelo-código es distinta, la más frecuente es la siguiente: `PRIMARY KEY(partitioning key, clustering key_1 ... clustering key_n)` Sin embargo hay que tener en cuenta que en la construcción también es posible tener una partition key compuesta, es decir, una partition key formada por varias columnas, para ello se utilizan paréntesis y así delimitamos el conjunto de partición. Quedando una posible definición de la primary key de la siguiente manera: `PRIMARY KEY((partitioning key_1, ... partitioning key_n), clustering key_1 ... clustering key_n)`. Este proceso se repite por cada column family del modelo.

4.3. Caso de estudio Twissandra

Una vez definido el funcionamiento del generador de código podemos continuar con el caso de estudio planteado en el segundo capítulo sobre Twissandra. Como se presentó anteriormente el objetivo de este caso de estudio es la generación de código CQL a partir de un modelo UML 2.0. Para ello definimos una serie de reglas de transformación entre modelos


```

-----
DROP KEYSPACE [%=keyspace.name%];
CREATE KEYSPACE [%=keyspace.name%]
WITH replication = {'class': '%=keyspace.replicaPlacementStrategy%', 'replication_factor': [%=keyspace.replicaFactor%]};

USE [%=keyspace.name%];

[% for (cf in keyspace.columnFamilies){ tableKey="";%]
CREATE TABLE [%=cf.name%](
[% for (cols in cf.columns){ //este for define cada grupo de columnas
s=cols.name.toString();
for (c in cols.type){

if (c.isTypeOf(PrimitiveType)) //definicion del tipo primitivo
s=s+" "+c.kind.toString();

if (c.isTypeOf(MapType)) //definicion del tipo map
s=s+" Map"+"<" +c.keyType.toString()+" "+c.baseType.toString()+">";

if(c.isTypeOf(CollectionType)) //definicion del tipo set o list
s=s+" "+c.kind.toString()+"<" +c.keyType.toString()+">";

}
};
[%}%]
-----

```

Figura 4.1: Regla de transformación XX

y obtuvimos un modelo de Cassandra a partir de un modelo UML que representaba una versión simplificada de Twitter llamada Twissandra. En esta sección se presenta el resultado de la transformación del modelo de Twissandra a código.

Tras obtener el modelo Cassandra de Twissandra (figura ??) y tras la definición del generador de código se pone en funcionamiento la transformación modelo-código. El resultado de la transformación se detalla a continuación.

El código resultante tras ejecutar el generador de código es el siguiente (figura 4.3). Como vemos en el código las tres primeras líneas son para borrar el keyspace en caso de que existiera, crearlo y configurar la arquitectura según estaba especificada en el modelo Cassandra. A continuación se conecta la sesión con el keyspace correspondiente. Por cada column family (sea estática o dinámica) se genera un bloque `CREATE TABLE`. Cada columna de la column family es transformado en una columna de la tabla. La transformación como se contaba en la sección anterior es trivial, sin embargo el problema del generador de código surge a la hora de definir la estructura de la primary key. Como se comentó en secciones anteriores las column family estáticas siguen una definición de claves tradicional, sin embargo las column families dinámicas tienen que tener en cuenta que las claves de partición

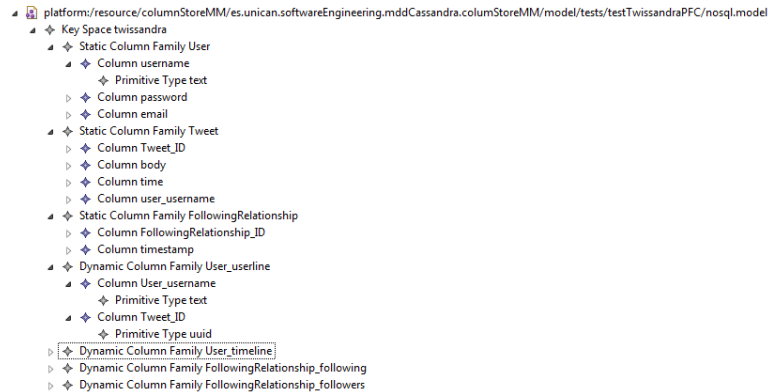


Figura 4.2: Modelo UML Twissandra

(partition keys) pueden ser compuestas (aunque en este ejemplo no existen). Por ejemplo el caso de user_userline las columnas user_name y tweet_id son designadas como primary key, user_username será la clave de partición.

4.4. Pruebas

Una vez implementado el generador de código la siguiente tarea consiste en comprobar que el código generado funciona correctamente. Para ello creamos, una serie de pruebas unitarias que permitan comprobar que el funcionamiento de los generadores de código es correcto para un conjunto de modelos de entrada.

Estas pruebas unitarias se han implementado en EUnit, el lenguaje de definición de pruebas de la suite Epsilon. EUnit funciona de una manera muy similar a JUnit, pero aplicado a los lenguajes de la suite Epsilon, como EGL. Utilizamos EUnit para comprobar que el funcionamiento del generador de código es correcto, para ello se diseñan una serie de casos de prueba y se crea la salida esperada de cada uno de esos casos de prueba de forma manual. A continuación, se ejecuta el caso de prueba creado en EUnit y se comprueba que la salida generada coincide con la esperada, que es la creada manualmente.

4.5. Despliegue

4.6. Sumario

Durante este capítulo se ha descrito el proceso de desarrollo del generador de código. En primer lugar se ha presentado el código del generador de código y parte de la sintaxis utilizada para construirlo así como la descripción de su realización. A continuación se ha detallado el proceso de transformación del

caso de estudio introducido en el capítulo dos. Finalmente se han presentado las pruebas realizadas así como la herramienta utilizada.

```
DROP KEYSPACE twissandra;
CREATE KEYSPACE twissandra
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};

USE twissandra;

CREATE TABLE User(
    username text,
    password text,
    email set<text>,
    PRIMARY KEY(username)
);

CREATE TABLE Tweet(
    Tweet_ID uuid,
    body text,
    time timestamp,
    user_username text,
    PRIMARY KEY(Tweet_ID)
);

CREATE TABLE FollowingRelationship(
    FollowingRelationship_ID uuid,
    timestamp timestamp,
    PRIMARY KEY(FollowingRelationship_ID)
);

CREATE TABLE User_userline(
    User_username text,
    Tweet_ID uuid,
    PRIMARY KEY((User_username), Tweet_ID)
);

CREATE TABLE User_timeline(
    User_username text,
    Tweet_ID uuid,
    PRIMARY KEY((User_username), Tweet_ID)
);

CREATE TABLE FollowingRelationship_following(
    FollowingRelationship_FollowingRelationship_ID uuid,
    username text,
    PRIMARY KEY((FollowingRelationship_FollowingRelationship_ID), username)
);

CREATE TABLE FollowingRelationship_followers(
    FollowingRelationship_FollowingRelationship_ID uuid,
    username text,
    PRIMARY KEY((FollowingRelationship_FollowingRelationship_ID), username)
);
```

Figura 4.3: Código resultante

Capítulo 5

Sumario y Trabajos Futuros

Este capítulo resume el trabajo realizado durante la elaboración del presente Proyecto Fin de Carrera, se describen brevemente las lecciones y experiencias personales aprendidas y la descripción breve de posibles trabajos futuros.

Índice

5.1. Sumario	39
5.2. Experiencia personal	39
5.3. Trabajos futuros	40

5.1. Sumario

Esta memoria de Proyecto Fin de Carrera ha descrito el proceso de desarrollo de un generador de código Cassandra bajo el desarrollo dirigido por modelos.

5.2. Experiencia personal

Esta sección describe las experiencias personales acontecidas a lo largo del proyecto.

Las tareas de aprendizaje de lenguajes como EOL, ETL o EGL no resultaron muy costosas sin embargo siempre cuesta adentrarse en materias desconocidas ya que a pesar de que EOL por ejemplo es similar a OCL (Object Constraint Language) son conceptos que no se ven mucho durante la carrera.

En cuanto a la tarea de desarrollo del generador de código se utilizó la herramienta Epsilon que ofrece funcionalidades que facilitan el trabajo de crear generadores de código, sin embargo el mayor problema de estas herramientas es que al ser de reciente creación el numero de errores que surgen

durante el desarrollo son molestos y numerosos, por ejemplo existen errores sin identificar durante la ejecución de los casos de prueba. También surgen dificultades a la hora de poner en marcha el generador de código ya que algunos parámetros de configuración de Epsilon no son descritos en los manuales, a pesar de que Epsilon proporciona a los usuarios manuales y tutoriales completos y didácticos pero a nivel básico. A pesar de algunos infortunios o problemas a lo largo del desarrollo el resultado del trabajo es satisfactorio.

El paradigma utilizado de desarrollo dirigido por modelos en mi opinión será una materia básica a la hora de estudiar tecnologías y herramientas que pueden facilitar la vida a las empresas. Personalmente creo que estas tecnologías utilizadas para la automatización de construcción software son parte del futuro y serán materias troncales para futuros alumnos que estudien Ingeniería Informática, ya que permiten a las empresas reducción de costes, ahorro de tiempo y reutilización de componentes tres pilares clave en materia de optimización de los recursos de una empresa software.

5.3. Trabajos futuros

En concreto, una cuestión obvia que se debe resolver, en cuanto los desarrolladores de Epsilon solucionen los problemas planteados, es completar la integración con Eclipse, mediante un plug-in que pueda ejecutar los generadores de código sin que estos generen ningún tipo de excepción.