UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Un lenguaje de modelado para el diseño de lenguajes embebidos en Java 8

Enrique Albaladejo Barambio
Tutor: Juan de Lara Jaramillo

JUNIO 2016

Un lenguaje de modelado para el diseño de lenguajes embebidos en Java 8

AUTOR: Enrique Albaladejo Barambio
TUTOR: Juan de Lara Jaramillo

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2016

Resumen

Los lenguajes de dominio específico – Domain-Specific Languages (DSL) son lenguajes "pequeños", reducidos, enfocados a resolver una tarea concreta de un sistema software [9]. Ejemplos de DSL incuyen lenguajes de consultas (SQL), para la descripción de interfaces de usuario, o el cálculo matemático, o las máquinas de estados.

Estos LDE han estado poco presentes en proyectos de software en Java, debido en gran medida a que este lenguaje no poseía una sintaxis flexible que permitiera integrar estos lenguajes de manera sencilla. Con la llegada de Java 8 y la introducción de las expresiones lambda (también llamadas funciones anónimas), las cuales permiten pasar como parámetro fragmentos de código, se salvan en mayor o menor medida estas restricciones a la hora de integrar dichos lenguajes en Java.

El propósito de este Trabajo de Fin de Grado consiste en la implementación de un lenguaje de modelado cuyo objetivo principal es ayudar a integrar lenguajes de dominio específico embebidos en Java 8, utilizando tanto métodos con expresiones lambda como otros métodos utilizados antes de la introducción de dichas expresiones en Java 8, como métodos encadenados y anidados. Así mismo se incluye un entorno de modelado con una sintaxis concreta concordante con otros elementos de Java, así como un generador de código destinado a sintetizar las clases e interfaces de Java diseñadas con el propio lenguaje de modelado y de esta manera poder integrarlas fácilmente en proyectos ya creados. También se proporciona un validador de cara a ayudar con el diseño del lenguaje de dominio específico que se quiera realizar, el cual comprueba y valida el propio lenguaje según se va implementando. Todos estos elementos se han desarrollado utilizando *Eclipse Modeling Framework* y se han integrado en un *plugin* de Eclipse.

Para probar el lenguaje de modelado se han diseñado varios lenguajes de dominio específico y se ha comprobado la correcta validación y posterior generación del código Java.

Palabras clave

Java, Lenguaje de Dominio Específico, modelado, generación de código, expresión lambda

Abstract

Domain-Specific Languages (DSL) are “small”, reduced languages focused on resolving specific tasks of a software system [9]. Some examples of DSLs are Structured Query Languages (SQL), languages to describe user interfaces, math calculus or state machines.

Domain-Specific Languages (DSL) have had small presence on Java projects largely due to the lack of a flexible syntax on the Java language to allow the integration of this languages easily. With the release of Java 8 and the introduction of the lambda expressions (also called anonymous functions), which allow passing code fragments as argument, said restrictions are more or less overcome when comes to embed those languages into Java.

The purpose of this Bachelor Thesis is to implement a modeling language whose main objective is to help integrating domain specific languages embedded into Java, using both lambda expressions methods and other methods used before the addition of lambda expressions in Java 8, like chaining and nested methods. It is furthermore included a modeling environment with a specific syntax matching other Java elements as well as a code generator meant to synthesize Java classes and interfaces designed with the modeling language and, thereby, being able to integrate it into projects already created. Also it is provided a validator to help with the design of the desired domain specific language. Said validator verifies and validates the language as it implements. Each and every one of the mentioned elements have been developed using *Eclipse Modeling Framework* and have been integrated on an Eclipse’s *plugin*.

To test the modeling language, several domain-specific languages have been designed and also verified the right validation and subsequent Java code generation.

Keywords

Java, Domain-Specific Language, modeling, code generation, lambda expression

Agradecimientos

Quiero agradecer en primer lugar a mi tutor, Juan de Lara Jaramillo, la inestimable ayuda que me ha ofrecido y todo el apoyo que me ha brindado.

También quiero agradecer a mi familia el estar siempre a mi lado apoyándome y ayudándome en todo lo que pudieron, y sobretodo porque sin ellos no habría llegado tan lejos.

Por último agradezco a mis amigos su apoyo y ánimos siempre que los necesitaba.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

Las expresiones lambda se introdujeron en el lenguaje Java el 18 de marzo de 2014, como elemento fundamental del lanzamiento de la versión 8.0 de Java SE. Se añadieron, tal y como se indica en las notas de lanzamiento de la versión, para permitir expresar interfaces de un solo método (interfaces funcionales) de manera más compacta.

Por ejemplo, interfaces como *Comparator*, que permite entre otras cosas comparar dos objetos de cara a ordenar colecciones, son interfaces funcionales. Si se pasa como argumento un objeto de tipo *Comparator* a un método que acepte este tipo de objetos, como el método *sort*, se le proporciona al método una función que le permite ordenar una colección según el criterio que se haya definido en dicho *Comparator*. Antes de la llegada de Java 8, se tenía que definir previamente una clase de tipo *Comparator* que implementara el método que permite definir el criterio de ordenación (*compare*), sin embargo, gracias a la introducción de las expresiones lambda el criterio de ordenación se puede definir en la propia llamada al método *sort*.

Esta mejora en el uso de interfaces funcionales es muy útil para el tema que se trata en este TFG, la creación de Lenguajes de Dominio Específico en Java.

1.1 Motivación

Los lenguajes de dominio específico – Domain-Specific Languages (DSL) son lenguajes "pequeños", reducidos, enfocados a resolver una tarea concreta de un sistema software [9]. Como se verá en detalle más adelante existen dos tipos de DSL: internos y externos.

Los lenguajes de dominio específico son un apartado de la informática y el desarrollo de software que aún está en descubrimiento y análisis, más aún en cuanto a su inclusión en lenguajes de dominio general, como Java, se refiere.

Con la introducción de las expresiones lambda, esta inclusión es cada vez más fácil y requiere de menor esfuerzo por parte de los programadores, sin embargo no se dispone de muchos ejemplos en los que se haya tratado de utilizar estas expresiones lambda para integrar DSLs en Java, lo cual podría suponer un gran beneficio y ahorro, sobretodo de tiempo, a la hora de desarrollar proyectos en este lenguaje.

Por ejemplo, se podría crear un DSL de una máquina de estados específica en Java, mantenerlo en ejecución e ir probando eventos en dicha máquina para analizar su comportamiento. Sin embargo, esto en Java es complicado de implementar, debido a las restricciones del propio lenguaje y a la falta de herramientas que ayuden en esta tarea (como son los lenguajes de modelado).

Es por ello que se ha decidido desarrollar un lenguaje de modelado que permita hacer más sencilla la tarea de crear nuevos DSL de cara a introducirlos en proyectos software para tareas específicas con las que se puedan beneficiar.

1.2 Objetivos

Los objetivos principales de este proyecto son:

- Desarrollar un lenguaje de modelado fácil de utilizar y que sea lo más parecido posible al lenguaje Java, permitiendo así ser utilizado por programadores ya conocedores del lenguaje.

- Generar el código Java correspondiente al lenguaje de dominio específico creado de cara a facilitar su integración en proyectos Java y así reducir el esfuerzo necesario para integrarlo.

- Permitir la posibilidad de utilizar varios enfoques a la hora de generar dicho código, permitiendo elegir varios estilos para el API generada (encadenado, anidado y mediante expresiones lambda), lo que da al usuario la elección del método que más le convenga.

- Ofrecer una validación del lenguaje para ayudar al usuario a evitar errores de compilación y ejecución.

- Posibilitar el uso de las recientemente introducidas interfaces funcionales.

1.3 Organización de la memoria

La memoria se encuentra organizada en los siguientes capítulos:

- **Capítulo 2:** En este capítulo se exponen las tecnologías utilizadas a lo largo del desarrollo del proyecto así como algunos conceptos previos necesarios para la comprensión del documento.

- **Capítulo 3:** En este capítulo se analiza el diseño utilizado para la realización del proyecto.

- **Capítulo 4:** Aquí se explica la fase de desarrollo del proyecto.

- **Capítulo 5:** En este capítulo se presenta un caso de prueba y se resuelve utilizando el proyecto.

- **Capítulo 6:** Por último, en este capítulo se realiza un análisis de los objetivos cumplidos y una serie de posibles mejoras para el futuro.

2 Tecnologías a utilizar y conceptos previos

En este capítulo se estudiará cada una de las tecnologías utilizadas en el desarrollo de este proyecto así como una serie de conceptos que se mencionan a lo largo de este documento. En concreto se analizarán los siguientes aspectos:

- **Lenguajes de Dominio Específico – *Domain-Specific Languages (DSL)***
- **Desarrollo de software dirigido por modelos – *Model-Driven Software Development (MDSM)***
- ***Eclipse Modeling Framework (EMF)*** y la serie de herramientas utilizadas que incluye.

Se puede omitir este capítulo si ya se dispone de la suficiente información sobre los aspectos mencionados.

2.1 Lenguajes de Dominio Específico

Los Lenguajes de Dominio Específico – *Domain-Specific Languages (DSL)* son pequeños lenguajes centrados en realizar tareas muy específicas de software [9]. Debido a esto y a diferencia de los lenguajes de propósito general (como es Java), los DSL disponen de un alcance y capacidades limitadas, dado que están diseñados para un "dominio específico", lo que permite que sean simples y concisos.

Existen dos formas generales para clasificar a los DSL: externos o internos. Un DSL externo es un lenguaje que no depende de ningún lenguaje "anfitrión" y en el que el propio creador debe decidir su sintaxis y gramática [20]. Esto permite flexibilidad, ya que no está restringido por la gramática del lenguaje "anfitrión", pero por otro lado, obliga al desarrollador a definir la gramática, con todo el consumo de tiempo y recursos que eso conlleva.

En contraposición, los DSL internos son diseñados e implementados dentro de un lenguaje "anfitrión" (como puede ser Java) [9]. Esto implica que el creador del lenguaje no tiene que preocuparse de crear la gramática y dispone de las herramientas del lenguaje anfitrión, lo que permite una reducción del tiempo y recursos empleados. Sin embargo, también implica que el DSL está limitado por el propio lenguaje anfitrión, por lo que es, quizás, un esfuerzo extra el hacer que dicho DSL sea simple y conciso dadas las limitaciones del lenguaje anfitrión.

Como se puede comprobar, ambos tipos de DSL tienen sus ventajas e inconvenientes. Mientras que un DSL externo da libertad a la hora de diseñar la sintaxis como se desee a costa de definir la gramática, un DSL interno ahorra ese esfuerzo a costa de ser limitado por la gramática del lenguaje anfitrión.

2.2 Desarrollo de software dirigido por modelos

El Desarrollo de Software Dirigido por Modelos – *Model-Driven Software Development (MDSD)* es la metodología de desarrollo de software que se ha seguido para realizar este proyecto. Se trata de una metodología que se centra en crear y utilizar "modelos de dominio", los cuales son modelos conceptuales de todos los temas relacionados con un problema específico [16].

2.2.1 Objetivos

Los objetivos principales del MDSD son:

- **Incrementar la velocidad de desarrollo en un proyecto de software.** Esto se consigue gracias a la automatización: generando código a partir de los modelos creados.

- **Mejorar la calidad del software,** mediante el uso de transformaciones automatizadas y lenguajes de modelado definidos.

- **Permitir la reutilización,** ya que una vez que se ha definido la arquitectura, los lenguajes de modelado y las transformaciones se puede utilizar múltiples veces y en distintas aplicaciones, programas…

- **Mejorar la "gestión de complejidad a través de la abstracción"**, es decir, los lenguajes de modelado permiten "programar" en un nivel más abstracto.

Todos estos objetivos no son nuevos en el mundo del desarrollo de software, sin embargo el MDSD es un gran paso de cara a conseguir estos objetivos. A continuación se expondrá, de manera general, cómo el MDSD intenta llegar a estos objetivos.

2.2.2 Modelos

Un modelo es una representación abstracta de la función, comportamiento o estructura de un sistema. Que un modelo sea una representación quiere decir que es una imagen o idea de a aquello que representa; y que sea abstracta significa que simplifica esta representación haciéndola más general y centrándose sólo en los aspectos más importantes o más relevantes.

Dado que, como hemos visto, un modelo es una simplificación de una idea, permite el paso de análisis a implementación en el ciclo de vida de un proyecto de software de una manera más sencilla y eficaz.

2.2.3 Lenguajes de modelado

Tal y como el nombre indica, un lenguaje de modelado es un lenguaje utilizado para definir modelos, es decir, un lenguaje que permite representar una realidad de una manera simplificada y centrada en los aspectos de interés. Todo lenguaje de modelado se encuentra dividido en los siguientes componentes: sintaxis abstracta, sintaxis concreta y semántica.

La sintaxis abstracta de un lenguaje especifica, a grandes rasgos, la estructura del lenguaje; mientras que la sintaxis concreta de un lenguaje define la información acerca de la propia representación del lenguaje. Se podría decir que la sintaxis concreta es la "materialización" de una sintaxis abstracta. Sin embargo, una sintaxis abstracta puede poseer varias sintaxis concretas, o lo que es lo mismo, varias sintaxis concretas pueden tener en común la misma sintaxis abstracta. Por ejemplo, una suma de dos enteros (Figura 2-1) puede ser representada de diversas maneras (Figura 2-2).

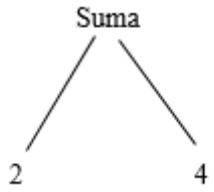

Figura 2-1: *Sintaxis abstracta de una suma*

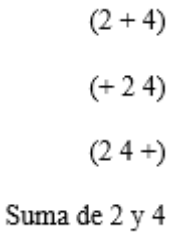

Figura 2-2: *Sintaxis concretas de una suma*

Por otro lado, la semántica de un lenguaje determina el criterio para decidir si el lenguaje está bien formado. Un ejemplo en los lenguajes de programación sería la regla que determina que una variable debe ser declarada. Esto no puede ser determinado por la sintaxis del lenguaje; un analizador sintáctico no puede detectar si se ha declarado una variable o no, sin embargo el compilador fallará en caso de que no se haya declarado.

2.2.4 Meta-modelos

Dado que ya se ha definido lo que son los modelos y los lenguajes de modelado, pasemos ahora a definir los meta-modelos. Un meta-modelo es, básicamente, el modelo de un modelo, es decir, define conceptos que se utilizaran para la creación del modelo (instancia del meta-modelo) así como la posible estructura del mismo. Un meta-modelo define la sintaxis abstracta y la semántica de un lenguaje de modelado.

Ya que un modelo es una instancia de un meta-modelo, se necesita, para definir un meta-modelo, un lenguaje de meta-modelado, lo que a su vez, obliga a que sea definido por un meta-meta-modelo. Se podría pensar, lógicamente, que esto llevaría a un bucle infinito de definición de meta-modelos, sin embargo, en la práctica esto no es así, ya que se ha comprobado que un meta-modelo puede definirse a sí mismo, por lo que no es necesario ir más allá del primer meta-modelo.

En la Figura 2-3 se puede observar un ejemplo de una serie de niveles de meta-modelado. Así como las relaciones entre clases e instancias de cada uno.

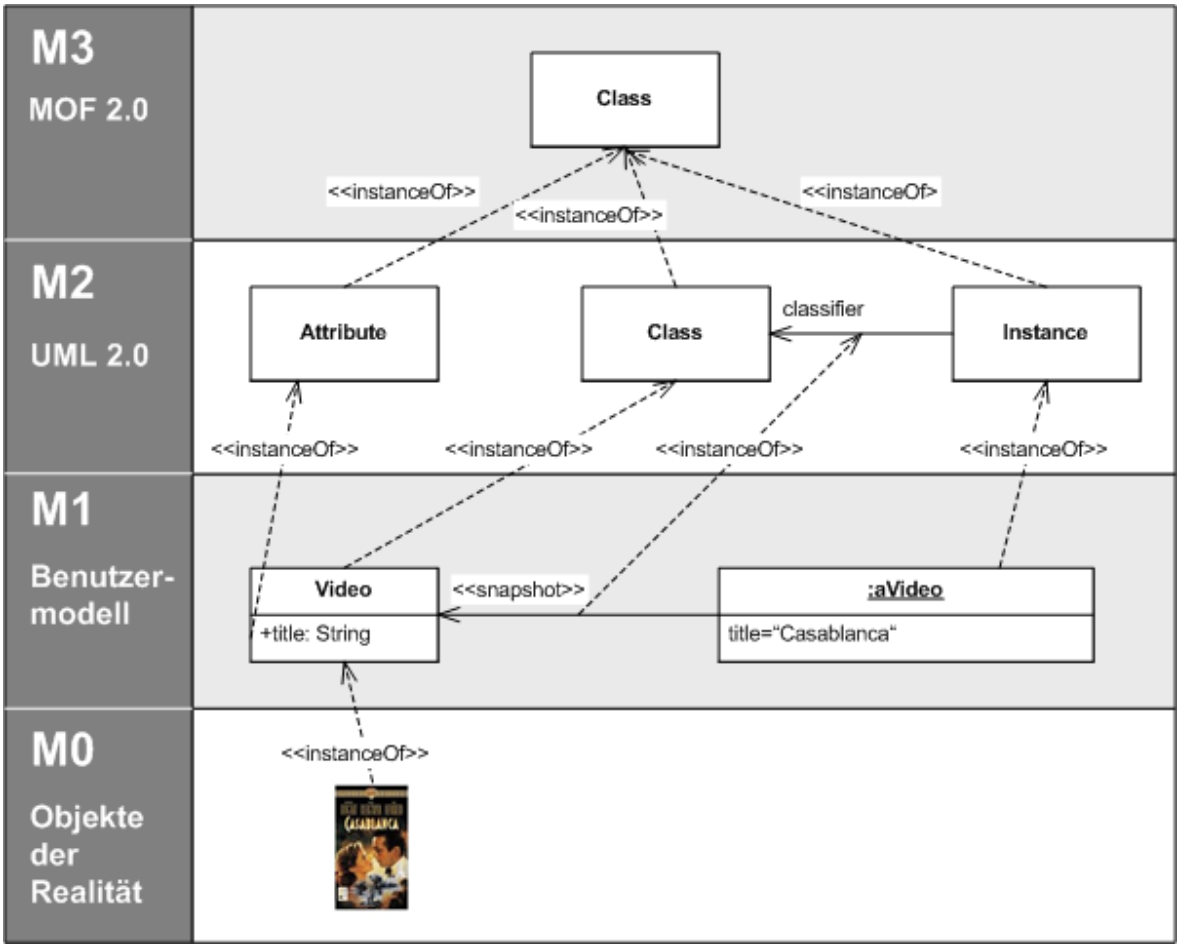

Figura 2-3: *Ejemplo de meta-modelos [10]*

Además dentro del meta-modelo se han definido una serie de restricciones OCL (Object Constraint Languaje). Estas restricciones no son más que reglas para expresar condiciones no expresables mediante los diagramas y cardinalidades. Por ejemplo, la necesidad de que los nombres de las clases sean únicos.

2.2.5 Generación de código

La generación de código consiste en, una vez definidas ambas sintaxis (abstracta y concreta), plasmar dichas sintaxis en la semántica del lenguaje con código específico y generado automáticamente. Dicha generación de código presenta ventajas frente al uso de otros acercamientos más genéricos, algunas de estas ventajas son:

- **Rendimiento:** La generación de código permite alcanzar alto rendimiento manteniendo cierta flexibilidad.

- **Volumen de código:** Generar código permite mantener una estructura de código compacta, ya que el generador de código sólo generará el código imprescindible para ejecutar aquello que se necesite en cada momento.

- **Restricciones del lenguaje de programación:** Muchos lenguajes de programación tienen expresiones restrictivas que se pueden atajar mediante la generación de código.

2.3 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) es una infraestructura o *framework* de modelado y generación de código basado en Eclipse, un Entorno de Desarrollo Integrado – *Integrated Development Environment (IDE)*.

Dicha infraestructura es la que se ha utilizado para desarrollar este proyecto y más concretamente las herramientas que se expondrán a continuación.

2.3.1 Ecore

Ecore es un lenguaje de meta-modelado para la creación de meta-modelos en EMF. Es, también, su propio meta-modelo.

Ecore dispone, además, de una serie de componentes que se relacionan de acuerdo a la siguiente jerarquía (Figura 2-4):

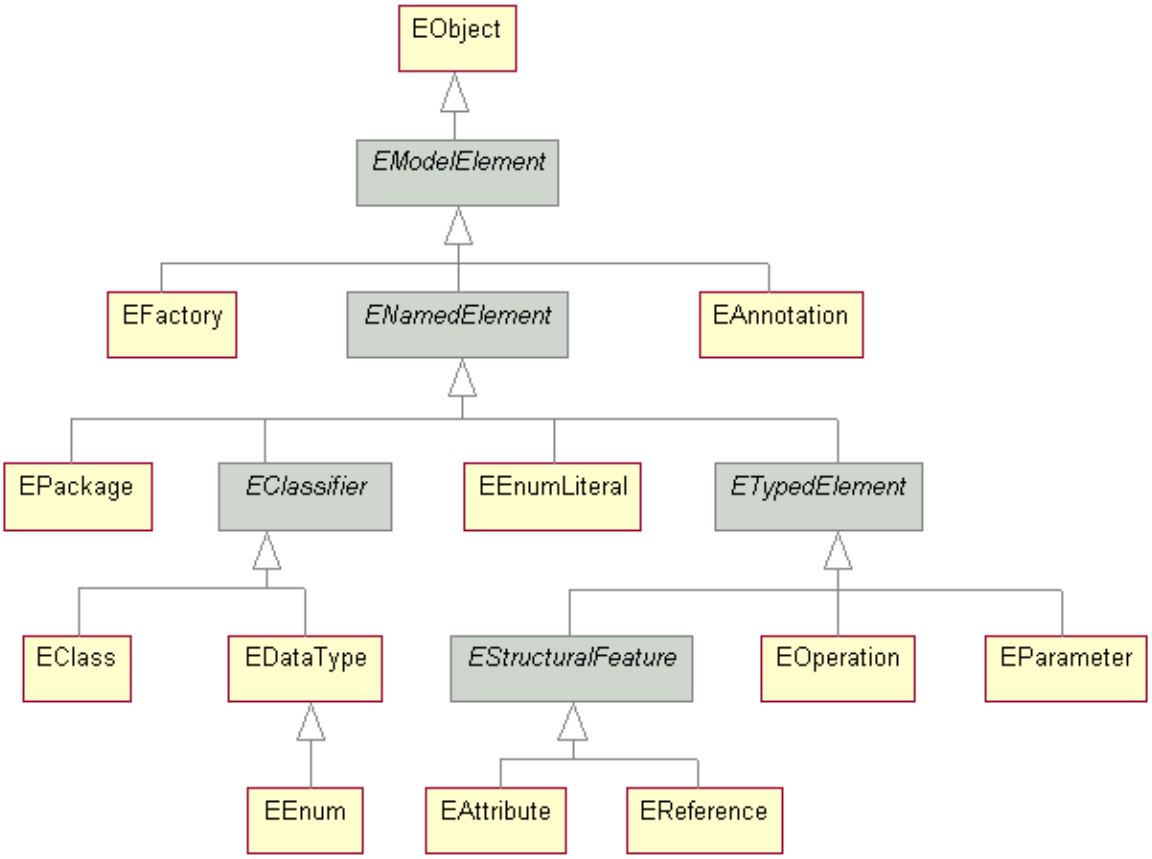

Figura 2-4: *Jerarquía de componentes de Ecore [8]*

También tiene asociados tipos de datos Ecore a los propios del lenguaje Java, como se puede comprobar en la Figura 2-5, además de otros tipos de datos externos.

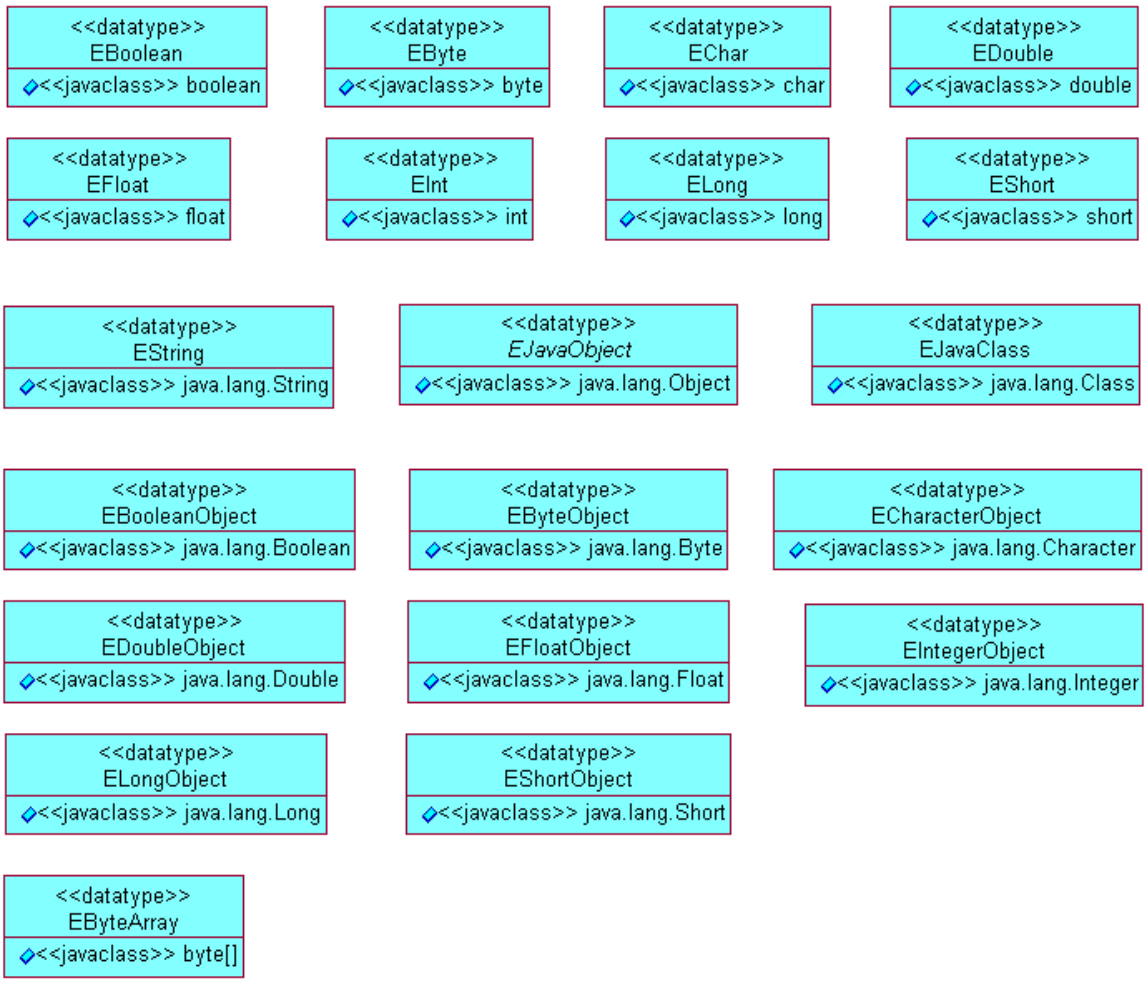

Figura 2-5: *Tipos de datos Java relacionados con Ecore [8]*

A su vez, de cara a definir los meta-modelos en Ecore, se dispone de una serie de editores, tanto textuales como gráficos. En este proyecto se ha utilizado un editor en forma de árbol.

2.3.2 Xtext

Xtext es un entorno de código abierto para el desarrollo de tanto lenguajes de programación como DSL [1] [24] y es el que se ha utilizado en este proyecto para la definición de la sintaxis concreta, como se verá en el Capítulo 4.

Xtext además genera un analizador sintáctico, un modelo de clases para la sintaxis abstracta y un entorno de desarrollo basado en Eclipse.

2.3.3 Xtend

Xtend es un lenguaje de programación basado en el lenguaje Java pero que lo mejora en varios aspectos tales como añadir funcionalidad a tipos ya definidos o proporcionar diversas características de interfaces funcionales.

En este proyecto se ha utilizado Xtend tanto para la generación de código como la validación.

3 Diseño

Este capítulo está destinado a especificar el diseño utilizado para el desarrollo del Lenguaje de Modelado así como la generación de código.

3.1 Arquitectura

El lenguaje desarrollado en este proyecto está dividido principalmente en:

- **Meta-modelo** que define la sintaxis abstracta del lenguaje.
- **Sintaxis concreta** textual.
- **Generador de código** Java.

Cada una de las partes en las que se divide la arquitectura se explican en detalle en las secciones posteriores de este capítulo.

3.2 Meta-modelo

Con el objetivo de definir la sintaxis abstracta del lenguaje se ha creado un meta-modelo en el que se incluyen los distintos componentes del propio lenguaje. Los componentes principales son:

- **Modelo semántico:** Componente que engloba al resto y que está definido por un nombre (elegido por el usuario) y el método en el que el usuario quiere que su DSL se genere: encadenado, anidado o con expresiones lambda. El método encadenado se basa en el encadenamiento de métodos Java (<objeto>.metodo.metodo…). El método anidado está basado en la creación de métodos dentro de otros métodos y que son referenciados entre ellos (<objeto>.metodo(método(método…))). Por último, el método con expresiones lambda se basa en el uso de interfaces funcionales (como se vió en la Introducción de este documento).

- **Clase:** De las cuales se compone el modelo semántico y son equivalentes a las clases Java generadas. Una clase puede ser heredada de otra, puede ser *top* (Clase que engloba a otras) y puede poseer una Variable (elemento explicado más adelante) que defina a la Clase (id) o no. En caso de que la Clase esté definida como tipo de dato (*ClassType*) en cualquier otra Clase del modelo semántico, dicha Clase debe poseer una Variable id, a no ser que la clase sea *top*. Sólo puede existir una Clase *top* en el Modelo semántico.

- **Elemento:** Cada clase se compone de elementos que equivaldrían tanto a los objetos Java que se generan como a fragmentos de código (expresiones lambda).

- **Atributo:** Un elemento puede componerse de atributos los cuales dan funciones extra a dichos elementos. Por el momento sólo se dispone de un tipo de atributo (*add*) el cual se explicará más adelante.

En la Figura 3-1 mostrada a continuación se pueden distinguir los componentes explicados.

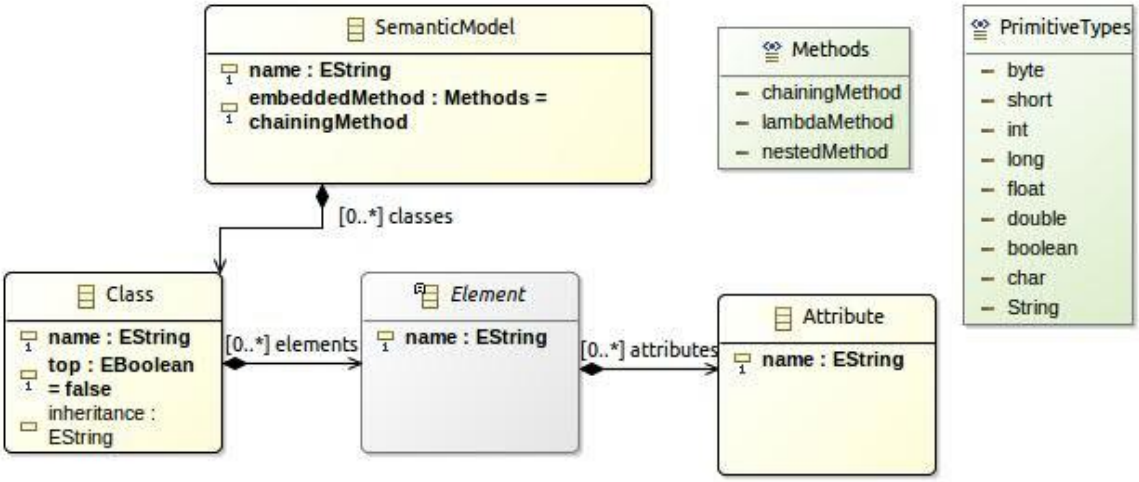

Figura 3-1: Meta-modelo del TFG simplificado

3.2.1 Elementos

Los elementos son una de las claves que definen este proyecto. Se encuentran divididos en:

- **Objeto:** Equivale a los objetos del propio lenguaje Java. Cada objeto posee su propio tipo de dato, bien sea definido por el usuario (Clase) o primitivo (*PrimitiveTypes* Figura 3-1). Un objeto está a su vez dividido en:
    - **Colección:** Engloba a elementos que poseen más de un valor, como son las listas, los conjuntos y los mapas. Estos elementos no pueden ser los que definan a una clase (id).
        - **Lista:** Se corresponden a los *ArrayList* [2] definidos en Java.
        - **Conjunto:** Se corresponden con los *TreeSet* [4] del lenguaje Java y se encuentran ordenados. Si su tipo de dato está definido por el usuario (Clase) y se indica una Variable por la que ordenar, el conjunto se ordenará utilizando dicha Variable, si no se ordenará utilizando la Variable id de la propia Clase.
        - **Mapa:** Se corresponde al objeto Java *LinkedHashMap* [3]. Además de disponer de un tipo de valor (como todos los Objetos), dispone de un tipo de clave, el cual sólo puede ser de tipo primitivo.
    - **Variable:** Contiene objetos que sólo disponen de un valor. Puede definir a la Clase en la que se encuentra (id) o no.

- **Código:** Corresponde a la interfaz Java *Consumer* [12] o bien *BiConsumer* [11], siendo ambas interfaces funcionales (ya explicadas en la introducción de esta memoria) y cuya función es la de recibir uno (en el caso de *Consumer*) o dos (en el caso de *BiConsumer*) argumentos y realizar una operación definida por el usuario con dichos argumentos. Dispone de un "contexto" que determina el tipo de dato del argumento de *Consumer* o del primer argumento de *BiConsumer*. También posee un componente opcional "onEvents" que determina el segundo argumento de *BiConsumer*.

Además un Elemento se puede componer de atributos, como ya se ha mencionado anteriormente, actualmente sólo se dispone de un tipo de atributo (*add)*. Dicho atributo debe encontrarse en un Objeto cuyo tipo de dato sea definido por el usuario (*ClassType*) y su función es la de permitir que la Variable o Colección en que se encuentre no necesite que su valor o valores (en caso de una colección) sean creados por el usuario antes de utilizarlos.

Por ejemplo, si se define una Variable "var" de tipo "Camion" con el atributo *add*, siendo "Camion" una Clase definida por el usuario, dicha Variable "var" a la hora de ser utilizada no necesitará que se haya creado previamente una instancia de la Clase "Camion", simplemente se creará una nueva según se utiliza.

En la Figura 3-2 mostrada a continuación se puede comprobar el diagrama de clases correspondiente al meta-modelo de Elemento.

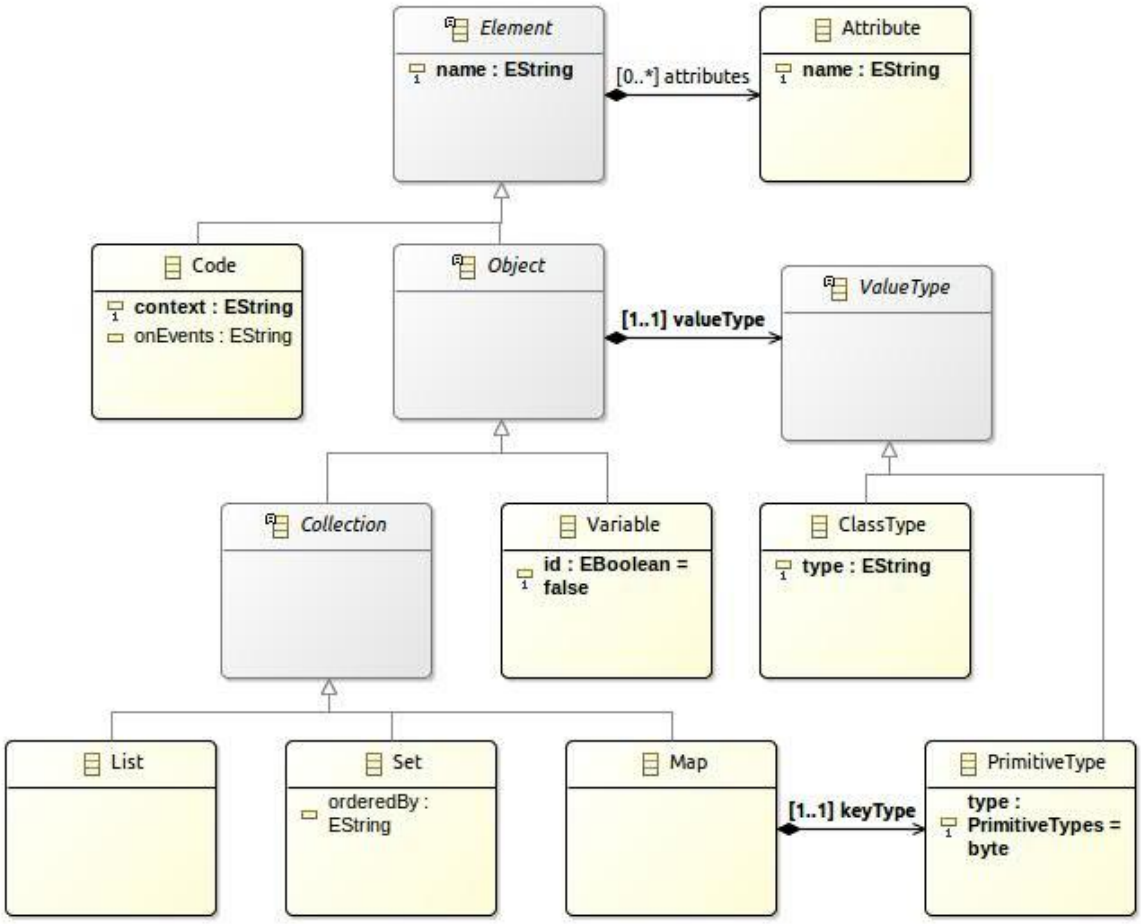

Figura 3-2: Meta-modelo de elementos

3.3 Sintaxis concreta

La sintaxis concreta explicada a continuación se trata de una sintaxis textual. Se ha decidido que sea textual en lugar de gráfica ya que el proyecto está destinado a ser utilizado por programadores en Java y a ser integrado en proyectos Java, por lo que no tendría sentido que la sintaxis fuera de otra forma no textual.

Toda la sintaxis concreta en este apartado está definida con la *Notación de Backus-Naur Extendido – Extended Backus-Naur Form (EBNF)* y esta notación es la que se muestra en los fragmentos de la sintaxis mostrados en esta sección.

En el Anexo A se puede encontrar la sintaxis concreta completa.

3.3.1 Modelo semántico

En la siguiente Figura 3-3 se puede observar la sintaxis general del Modelo semántico.

⟨*SemanticModel*⟩ ::= 'SemanticModel' ⟨*ID*⟩ '['⟨*Methods*⟩']' '{' (⟨*Class*⟩)* '}';

⟨*Class*⟩ ::= ('top')? 'Class' ⟨*ID*⟩ ('extends' ⟨*STRING*⟩)? '{' (⟨*Element*⟩)* '}';

⟨*Methods*⟩ ::= 'chainingMethod'
| 'nestedMethod'
| 'lambdaMethod';

Figura 3-3: Sintaxis concreta general del Modelo semántico

Como se puede comprobar, definir un Modelo semántico es sencillo. Se indica antes de nada el identificador del modelo (nombre) y el tipo de método a utilizar por el generador de código y, a continuación, se definen las clases deseadas.

Por ejemplo, si queremos definir un modelo semántico que contenga una única clase vacía podríamos hacerlo de la siguiente manera:

SemanticModel “MiModelo” [chainingMethod] {

Class “MiClase” {}

}

3.3.2 Elementos

En la Figura 3-4 que se puede ver a continuación, se puede comprobar cómo la definición de elementos es simple.

⟨*Code*⟩ ::= ⟨*ID*⟩ ':' 'Code' '[' 'context' ⟨*STRING*⟩ (',' ⟨*STRING*⟩ 'on events')? ']' ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*Variable*⟩ ::= ('id')? ⟨*ID*⟩ ':' ⟨*ValueType*⟩ ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*List*⟩ ::= ⟨*ID*⟩ ':' ⟨*ValueType*⟩ '[*]' ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*Set*⟩ ::= ⟨*ID*⟩ ':' ⟨*ValueType*⟩ 'ordered' ('by' ⟨*STRING*⟩)? ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*Map*⟩ ::= ⟨*ID*⟩ ':' ⟨*PrimitiveType*⟩ '-¿' ⟨*ValueType*⟩ ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

Figura 3-4: Sintaxis concreta de los elementos

Como se puede ver, para definir un elemento se indica primero el identificador del elemento (nombre), salvo en el caso de una variable, en la que antes se indica si se trata del ID de la clase; a continuación, separado por ':' se indica el tipo de dato del elemento o 'Code' en caso de código; por último se añaden los componentes que definen a cada tipo de elemento, como ya se indicaron en el apartado 3.1.

Por ejemplo, para definir una variable de tipo *String* y una lista de *Strings* se haría de esta manera:

"miVariable" : String;

"miConjunto" : String [*];

3.4 Generador de código

Como ya se ha indicado anteriormente, el generador de código permite obtener el código Java correspondiente a la sintaxis descrita por el usuario de cara a integrar el DSL deseado en proyectos Java. La estructura del generador de código se puede ver en la Figura 3-5 a continuación.

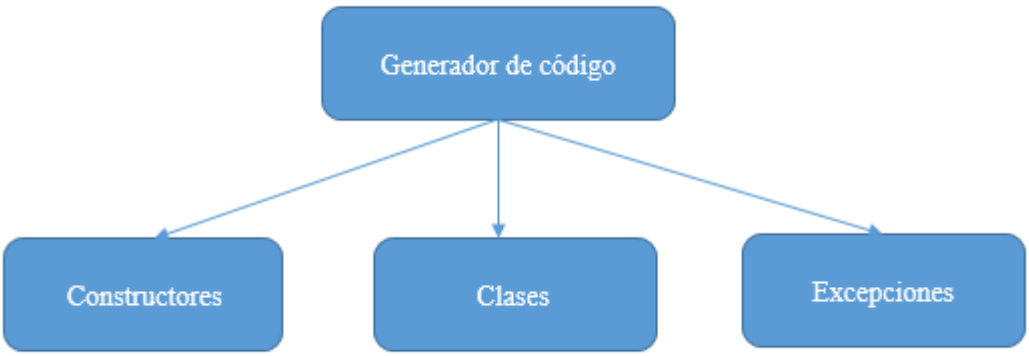

Figura 3-5: Estructura del generador de código

Como se puede comprobar el generador de código genera tres tipos de código Java:

- **Clases:** Código Java correspondiente a las clases definidas por el usuario.
- **Constructores:** Código necesario para poblar las clases definidas y específico para cada método indicado en el modelo semántico. Es decir, para el método encadenado se generan constructores distintos a los que se generan para el método anidado, e igual para el método con expresiones lambda.
- **Excepciones:** Generadas en caso de que sean necesarias; por ejemplo, para indicar que se está utilizando un objeto no creado anteriormente.

Por ejemplo, si definimos un DSL con un modelo semántico (“MiModelo”) y tres clases (“Clase1”, “Clase2”, “Clase3”) se generarían los siguientes ficheros:

- Tres ficheros para cada una de las clases (“Clase1.java”, “Clase2.java”, “Clase3.java”)
- Tres ficheros para cada constructor de las clases (“Clase1Builder.java”, “Clase2Builder.java”, “Clase3Builder.java”)

4 Desarrollo

En este capítulo, se analiza el desarrollo del proyecto según el diseño ya indicado en el capítulo anterior.

Todo el desarrollo del proyecto se ha realizado utilizando las diferentes herramientas de la infraestructura de modelado *Eclipse Modeling Framework (EMF)*. Estas herramientas son:

- Para el desarrollo del meta-modelo se ha utilizado el editor en forma de árbol *Sample Ecore Model Editor* para Ecore.

- Se ha usado la herramienta *OCLinEcore Editor* para implementar restricciones OCL (*Object Constraint Language*), ya mencionadas en el capítulo 2 (2.2.4).

- Para implementar la sintaxis concreta del proyecto se ha utilizado la infraestructura o *framework* Xtext

- Por último, la generación de código y validación se han desarrollado utilizando el lenguaje Xtend.

El desarrollo del proyecto se ha realizado siguiendo el diseño ya explicado en el capítulo 3. A continuación se analizan los distintos componentes del proyecto en cuanto al desarrollo se refiere, al igual que se hizo en el capítulo anterior con el diseño, mostrando fragmentos del código para ilustrar mejor la explicación.

4.1 Meta-modelo

Como ya se ha indicado al inicio de este capítulo, el meta-modelo ha sido implementado utilizando *Sample Ecore Model Editor*. Se trata de un editor que permite gran facilidad a la hora de definir el meta-modelo y a su vez una buena precisión, ya que el modo de utilizarlo es muy similar a como se haría un diagrama de clases (definiendo clases, atributos de las mismas, relaciones entre ellas, etc.). En la Figura 4-1 se puede ver un fragmento del meta-modelo en este editor.

Tal y como se puede comprobar, se trata de una representación de los diagramas de clases que se han visto en el capítulo 3. El modelo semántico, por ejemplo, dispone de un nombre, un atributo que define el tipo de método del mismo y está compuesto de clases. También se puede comprobar como Clase o Elemento disponen de una serie de restricciones (*constraints*) las cuales se verán en detalle a continuación cuando se hable de las restricciones OCL ya mencionadas anteriormente. Así mismo, los tipos de datos de los atributos son Ecore (EString, EBoolean…) pero se corresponden a los tipos de datos en Java (String, Boolean).

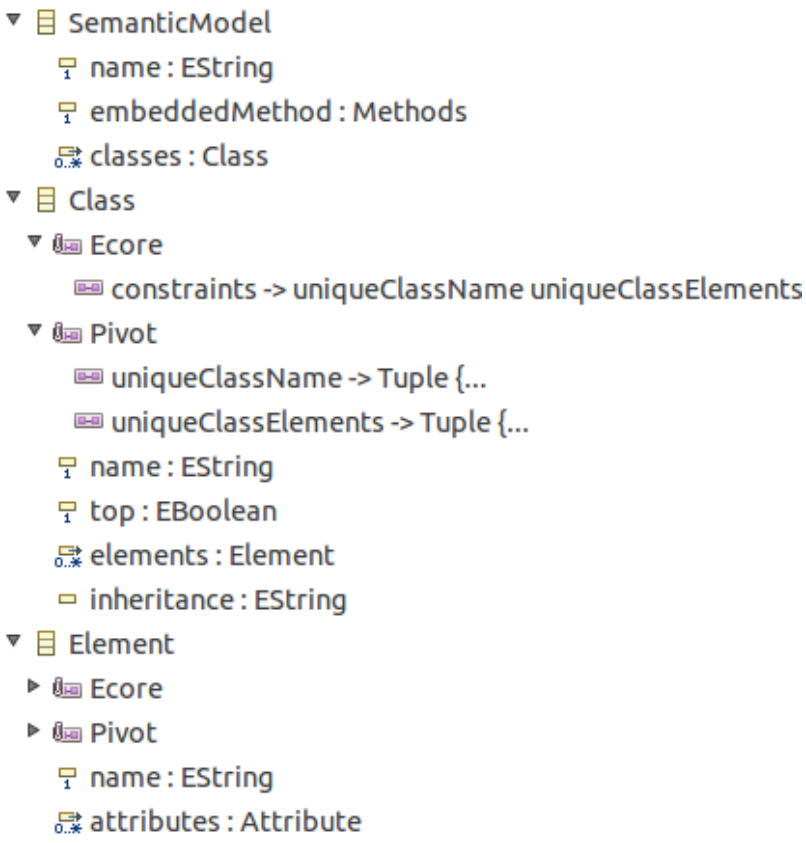

Figura 4-1: Fragmento del meta-modelo Ecore

4.1.1 Restricciones OCL

Dentro del meta-modelo Ecore es necesario exponer una serie de restricciones OCL implementadas. Dichas restricciones sirven para expresar condiciones no expresables mediante los diagramas y cardinalidades definidos en el propio meta-modelo.

```
class Class
{
    attribute name : String;
    attribute top : Boolean = 'false';
    property elements : Element[*] { ordered composes };
    attribute inheritance : String[?];
    invariant uniqueClassName('The name of the class must be unique'):
        Class.allInstances()->isUnique(name);
    invariant uniqueClassElements('The name of the elements inside a class must be unique'):
        elements->forAll(e1, e2 |
            e1 <> e2 implies e1.name <> e2.name
        );
}
abstract class Element
{
    attribute name : String;
    property attributes : Attribute[*] { ordered composes };
    invariant uniqueElementAttributes('The name of the attributes inside an element must be unique'):
        attributes->forAll(a1, a2 |
            a1 <> a2 implies a1.name <> a2.name
        );
}
```

Figura 4-2: Fragmento de código de OCL

Tal y como se ve en la Figura 4-2, las restricciones OCL definidas en este fragmento de código limitan la creación de componentes con el mismo nombre, ya sean las clases, los elementos, los atributos dentro de estos elementos…

4.2 Sintaxis concreta

El siguiente fragmento (Figura 4-3) contiene el código general en Xtext del modelo semántico y la clase.

```
SemanticModel returns SemanticModel:
    'SemanticModel' name=EString '['embeddedMethod=Methods']' '{'
        (classes+=Class)*
    '}';

Class returns Class:
    (top?='top')? 'Class' name=EString ('extends' inheritance=EString)? '{'
        (elements+=Element)*
    '}';

enum Methods returns Methods:
    chainingMethod = 'chainingMethod' | lambdaMethod = 'lambdaMethod' | nestedMethod = 'nestedMethod';

EBoolean returns ecore::EBoolean:
    'true' | 'false';

EString returns ecore::EString:
    STRING | ID;
```

Figura 4-3: Fragmento de la sintaxis concreta en Xtext

Ya que se trata de la sintaxis concreta, la sintaxis reflejada en Xtext es la que utilizará el usuario para definir su DSL. Primero hay que indicar que se va a definir un modelo semántico introduciendo "SemanticModel" seguido del nombre del modelo semántico, a continuación introducir entre corchetes el tipo de método que se va a seguir para implementar el lenguaje y por último, como si de una clase Java se tratase, abrir una llave para definir las clases dentro del modelo y cerrarla cuando se haya terminado.

Cada clase dentro del modelo se define de la siguiente manera: Primero se indica si la clase va a ser *top* o no, es decir, si va a contener a otras clases; luego se introduce la palabra "Class" seguida del nombre de la propia clase; en caso de que se quiera que la clase sea heredada de otra, se indica mediante la palabra "extends" (al igual que se hace en Java para indicar que una clase hereda de otra) seguido del nombre de la clase de la que hereda (dicha clase sólo puede ser una creada por el usuario); por último, al igual que en el modelo semántico, se abren llaves para introducir cada uno de los elementos de los que se compone la clase.

Además, dado que se utiliza la notación de *Xtext*, hay que definir los tipos de métodos para el modelo semántico así como los valores de tipo EBoolean y EString, siendo ambos tipos de datos Ecore correspondientes a los de Java.

4.3 Generación de código

El desarrollo del generador de código se ha realizado siguiendo el diseño establecido en el capítulo 3.

Como ya se indicó en el capítulo anterior, existen tres variantes de código generado: clases, constructores y excepciones. Cada una de estas variantes está contenida en su propio paquete Java. En el paquete de las clases se encuentran los ficheros .java correspondientes a cada una de las clases que haya definido el usuario a la hora de crear su DSL; en el paquete de los constructores se encuentran los ficheros .java necesarios para poblar cada una de dichas clases; por último, en el paquete de excepciones se encuentran las excepciones pertinentes en caso de que se requieran.

Todo el generador de código está escrito en Xtend, como ya se indicó anteriormente. A su vez está dividido en métodos que generan cada uno de los ficheros .java descritos: uno para generar cada una de las clases, otro para las excepciones y tres más, cada uno para generar los ficheros correspondientes a los constructores del método del modelo semántico elegido por el usuario (en cadena, anidado o con expresiones lambda).

```
def compileObjectNotCreatedException(){
    '''
    package «sm.name»Exceptions;

    @SuppressWarnings("serial")
    public class ObjectNotCreated extends Exception {
        public ObjectNotCreated(String cl, String name){
            super(cl+" \""+name+"\" not created.");
        }
    }
    '''
}
```

Figura 4-4: Fragmento de generador de código

La Figura 4-4 es un ejemplo del generador de código, específicamente de la excepción “Objeto no creado”.

4.4 Validación de código

La validación del código, como se ha dicho, está implementada con Xtend, al igual que el generador. La validación permite definir reglas al usuario a la hora de desarrollar su DSL.

```
@Check
def classNameStartsWithCapital(Class cl){
    if(!Character.isUpperCase(cl.name.charAt(0)))
        warning("Class name should start with a capital letter",
            EmbeddedLanguagePackage.Literals.CLASS__NAME
        )
}

@Check
def hasIdIfNeeded (EmbeddedLanguage.Object obj){
    var SemanticModel sm = (obj.eContainer.eContainer as SemanticModel)
    if(obj.valueType instanceof ClassType){
        for(cl : sm.classes)
            if(cl.name.equals((obj.valueType as ClassType).type) && cl.hasId) return;
        error('Class '+(obj.valueType as ClassType).type+' needs an id element',
            EmbeddedLanguagePackage.Literals.OBJECT__VALUE_TYPE,
                'noId');
    }
}

def boolean hasId (Class cl){
    for(e : cl.elements){
        if(e instanceof Variable && (e as Variable).id) return true;
    }
    return false;
}
```

Figura 4-5: Fragmento del validador

En la Figura 4-5 se puede ver un fragmento del validador, concretamente se valida con un *warning* si el usuario ha introducido alguna clase con un nombre sin la primera letra mayúscula (para concordar con el código Java) o un error si una clase ha de tener un elemento id pero no lo tiene. Además, se puede comprobar que hay dos métodos que están precedidos por la anotación *@Check* esto indica que serán comprobados por el validador cada vez que se introduzca código por parte del usuario.

5 Caso de prueba

En este capítulo se analizará un caso de prueba para comprobar el comportamiento del proyecto. El caso de prueba se trata de un DSL que permita la creación y población de grafos como el de la Figura 6-1.

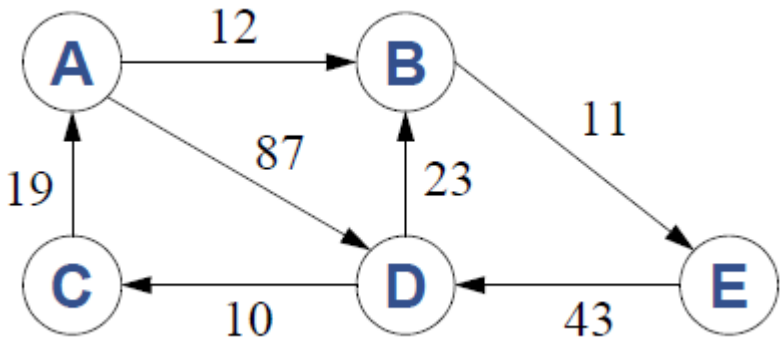

Figura 5-1: *Grafo dirigido*

Como es sabido, un grafo dispone de nodos y aristas. Cada nodo dispone de una etiqueta y cada arista de un peso. Además, como se trata de un grafo dirigido, cada nodo apunta a otro mediante una arista.

Este es el caso que se va a probar en este capítulo.

5.1 Definición del DSL

Para empezar, se crea un proyecto Java vacío y se le añade un fichero .embeL (extensión DSL del proyecto) en el cual se escribirá el DSL (Figura 6-2).

```
SemanticModel MyGraphs [chainingMethod]{
    top Class Graph {
        id name : String;
        nodes : Node [*];
        edges : Edge ordered;
    }

    Class Node {
        id label : String;
    }

    Class Edge {
        id weight : double;
        from : Node {add};
        to : Node;
    }
}
```

Figura 5-2: *DSL de grafo*

Como se puede comprobar, se indica que se quiere crear un modelo semántico llamado "MyGraphs" y el método que se utilizará para su generación es el encadenado (*chainingMethod*). A continuación se define la clase *top* "Graph" que tiene un nombre (que además es el id de la clase) y al ser *top* tiene que contener al resto de clases, nodes siendo una lista y edges un conjunto ordenado por el id de la clase "Edge". Después está definida la clase "Node" que tiene una etiqueta y la clase "Edge" que posee un peso y va de un nodo origen a un nodo destino, al nodo de origen se le ha añadido el atributo "add" para mostrar su utilidad más adelante.

Si probamos, por ejemplo, a quitar el id del elemento "label" vemos (Figura 6-3) que el validador muestra un error indicando que la clase "Node" necesita un id ya que está siendo accedida desde otra clase. Algo parecido ocurre si cambiamos el tipo de dato del elemento "nodes" de la clase "Graph" de "Node" a "Nodo"; puesto que "Nodo" no es una clase definida por el usuario ni un tipo de dato primitivo de Java, aparece un error indicando que la clase "Nodo" no existe (Figura 6-4), dicho error definido por una restricción OCL.

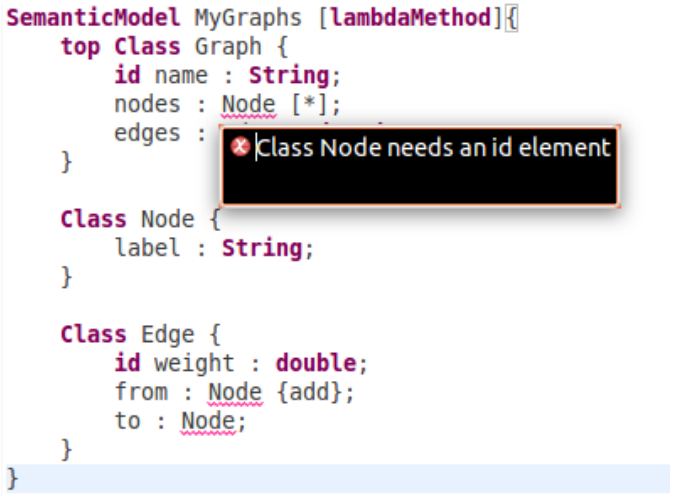

Figura 5-3: *Error id necesario*

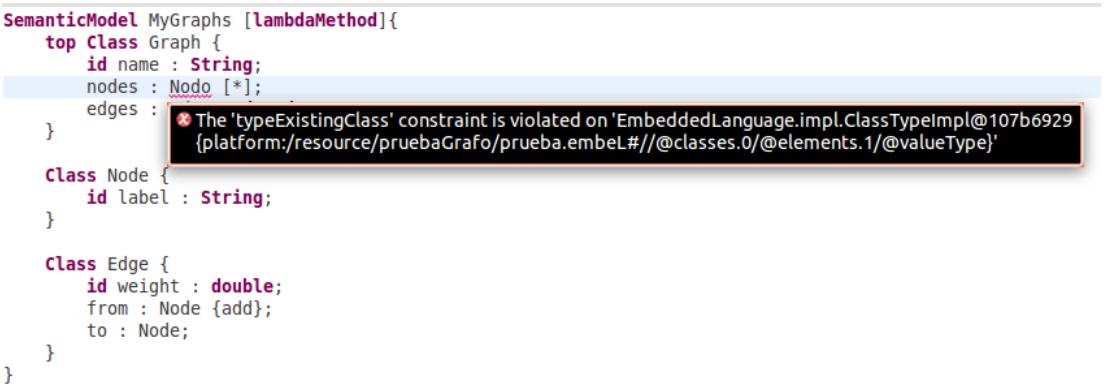

Figura 5-4: *Error clase inexistente*

Una vez se guarda el fichero .embeL, el generador de código genera las clases necesarias para su ejecución (exceptuando la clase main que ha de ser creada por el usuario). En la Figura 6-5 se pueden ver las clases generadas.

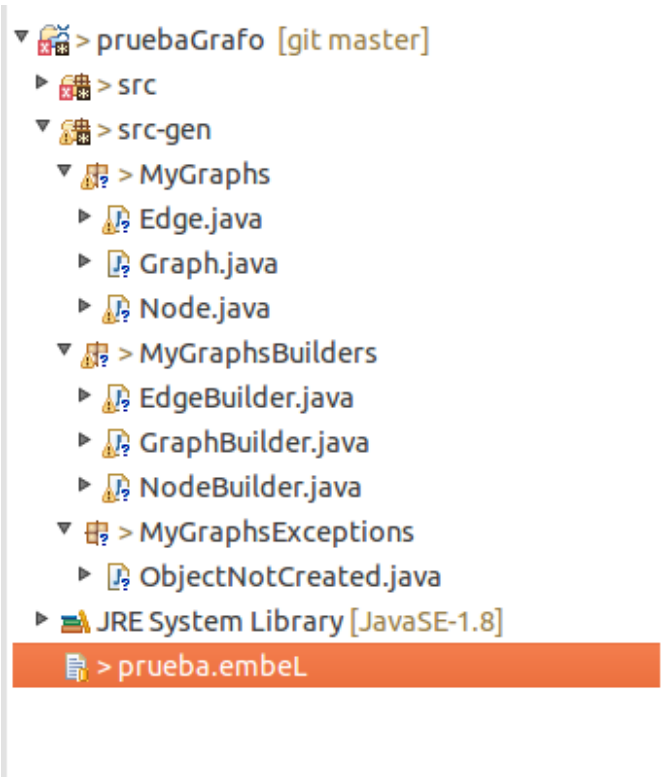

Figura 5-5: *Clases Java generadas*

Tal y como se puede ver, se han generado tres clases en el paquete "MyGraphs" (cada una correspondiente a la clase definida en el DSL) y otras tres clases en el paquete "MyGraphsBuilders" (correspondientes a los constructores), además de una clase de excepción.

Dado que el código de las clases es muy similar se ha decidido mostrar sólo un fragmento del código de la clase *Graph* (Figura 6-6). En dicho código se puede observar un constructor de la clase con el id (nombre), los *getters* y *setters* del nombre y el *getter* de la lista de nodos.

En la Figura 6-7 se ve un fragmento del código de la clase GraphBuilder, la cual dispone de métodos para crear nodos y aristas.

```java
public Graph(String name) {
    this.name = name;
    nodes = new ArrayList<>();
    edges = new TreeSet<>(new Comparator<Edge>() {

        @Override
        public int compare(Edge e1, Edge e2) {
            return (int) (e1.getWeight() - e2.getWeight());
        }
    });
}


public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}


public Node getNodesElement(String label){
    for(Node node : nodes)
        if(node.getLabel().equals(label))
            return node;
    return null;
}
```

Figura 5-6: *Fragmento clase Graph*

```java
  public GraphBuilder(String name) {
      graph = new Graph(name);
  }

  public static GraphBuilder Graph(String name) {
      return new GraphBuilder(name);
  }


  public Graph getGraph() {
      return graph;
  }

  public GraphBuilder node(String label) {
      NodeBuilder builder = new NodeBuilder(this, label);

      getGraph().addNodesElement(builder.node);

      return this;
  }
  public EdgeBuilder edge(double weight) {
      EdgeBuilder builder = new EdgeBuilder(this, weight);

      getGraph().addEdgesElement(builder.edge);

      return builder;
  }
```

Figura 5-7: *Fragmento clase GraphBuilder*

A continuación tendremos que crear una clase main para poder usar estos constructores y poder poblar nuestro grafo (Figura 6-8).

```
package pruebaGrafo;

import java.util.*;

import MyGraphs.*;
import static MyGraphsBuilders.GraphBuilder.*;

public class ChainingMain {
    public static void main(String[] args) {
        try{
            Graph("graph 1")
                .edge(12.0).from("a").to("b")
                .edge(87.0).from("a").to("d")
                .edge(19.0).from("c").to("a")
                .edge(10.0).from("d").to("c")
                .edge(23.0).from("d").to("b")
                .edge(11.0).from("b").to("e")
                .edge(43.0).from("e").to("d");
        } catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Figura 5-8: *Clase Main encadenada*

Como se puede ver, dado que hemos elegido el método encadenado, el grafo ha de poblarse encadenando los métodos para añadir nodos y aristas al grafo. Si ejecutamos este Main lanza un error (Figura 6-9); lo cual es debido a que no se ha creado el nodo "b" y esto es necesario ya que a la hora de definir el DSL no añadimos el atributo "add" al elemento "to" lo que obliga a tener que crear el nodo antes de poder utilizarlo, sin embargo esto no ocurre con los nodos "from".

```
MyGraphsExceptions.ObjectNotCreated: Node "b" not created.
        at MyGraphsBuilders.EdgeBuilder.to(EdgeBuilder.java:29)
        at pruebaGrafo.ChainingMain.main(ChainingMain.java:12)
```

Figura 5-9: *Captura de excepción lanzada*

A continuación veamos el mismo ejemplo utilizando el método con expresiones lambda y el método anidado. Las clases generadas en "MyGraphs" son las mismas, por lo que no tiene sentido centrarse en ellas, sin embargo, los "Builders" cambian, como podemos comprobar en las Figura 6-10 y 6-12.

Ahora la clase generada con el método con expresiones lambda utiliza la interfaz funcional *Consumer* (Capítulo 4) para permitir crear nodos y aristas mediante expresiones lambda, tal y como se puede observar en la Figura 6-11.

Por otro lado, el constructor generado por el método anidado, posee únicamente un constructor de "Graph" con el que crea todos los nodos y aristas introducidos, como se puede ver en la Figura 6-13.

```java
public static Graph Graph(Consumer<GraphBuilder> graphConsumer) {
    GraphBuilder graphBuilder = new GraphBuilder();
    graphConsumer.accept(graphBuilder);
    return graphBuilder.graph;
}
public void name(String name){
    graph.setName(name);
}
public void node(Consumer<NodeBuilder> nodeConsumer) {
    NodeBuilder nodeBuilder = new NodeBuilder();
    nodeConsumer.accept(nodeBuilder);
    Node node = nodeBuilder.node();

    graph.addNodesElement(node);
}
public void edge(Consumer<EdgeBuilder> edgeConsumer) throws ObjectNotCreated {
    EdgeBuilder edgeBuilder = new EdgeBuilder();
    edgeConsumer.accept(edgeBuilder);
    Edge edge = edgeBuilder.edge();

    graph.addEdgesElement(edge);
    int flag = 0;
    for(Node node : graph.getNodes())
        if(edge.getTo().getLabel().equals(node.getLabel()))
            flag = 1;
    if(flag == 0)
        throw new ObjectNotCreated("Node", edge.getTo().getLabel());
}
```

Figura 5-10: *Código GraphBuilder expresiones lambda*

```java
public static void main(String[] args)
      Graph g1 = Graph( graph -> {
          graph.name("grafo 1");
          try {
              graph.edge( e -> {
                  e.weight(12.0);
                  e.from("a");
                  e.to("b");
              });
              graph.edge( e -> {
                  e.weight(87.0);
                  e.from("a");
                  e.to("d");
              });
              graph.edge( e -> {
                  e.weight(19.0);
                  e.from("c");
                  e.to("a");
              });
              graph.edge( e -> {
                  e.weight(23.0);
                  e.from("d");
                  e.to("b");
              });
          } catch (Exception e1) {
              e1.printStackTrace();
          }
```

Figura 5-11: *Main expresiones lambda*

```java
package MyGraphsBuilders;

import MyGraphs.*;

public class GraphBuilder {
    public static Graph Graph(String name, Object... classes) throws ObjectNotCreated {
        Graph graph = new Graph(name);


        for(Object c : classes){
            if(c instanceof Node){
                graph.addNodesElement((Node) c);
            }
            if(c instanceof Edge){
                graph.addEdgesElement((Edge) c);
                int flag = 0;
                for(Node node : graph.getNodes())
                    if(((Edge) c).getTo().getLabel().equals(node.getLabel()))
                        flag = 1;
                if(flag == 0)
                    throw new ObjectNotCreated("Node", ((Edge) c).getTo().getLabel());
            }
        }
        return graph;
    }
}
```

Figura 5-12: *GraphBuilder método anidado*

```java
package pruebaGrafo;

import static MyGraphsBuilders.GraphBuilder.*;
import static MyGraphsBuilders.NodeBuilder.*;
import static MyGraphsBuilders.EdgeBuilder.*;
import MyGraphsExceptions.ObjectNotCreated;

public class NestedMain {
    public static void main(String[] args) {

        try {
            Graph("grafo 1",
                    edge(12.0, from("a"), to("b")),
                    edge(87.0, from("a"), to("d")),
                    edge(19.0, from("c"), to("a")),
                    edge(23.0, from("d"), to("b"))
                  );
        } catch (ObjectNotCreated e) {
            e.printStackTrace();
        }

    }
}
```

Figura 5-13: *Main método anidado*

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Los principales objetivos de este proyecto eran: la creación de un lenguaje de modelado sencillo y similar al lenguaje Java, que permitiera integrar o ser integrado en proyectos Java con facilidad, que diera la opción de utilizar diversos métodos de integración y permitir generar APIs con distintos estilos a partir de un único modelo semántico.

Se puede decir que se ha conseguido alcanzar la mayoría de estos objetivos, si no todos. Bien sea por la sencillez y compactación del código generado, por la facilidad de diseño del DSL a generar, por la variedad de métodos a utilizar para generar dicho código o por que se han introducido en mayor o menor medida las interfaces funcionales.

6.2 Trabajo futuro

Se podrían realizar diversas mejoras de cara al futuro, como por ejemplo:

- Añadir un mayor número de tipos de dato y colecciones.
- Introducir la posibilidad de herencia múltiple en las clases.
- Permitir la creación de interfaces.
- Añadir más atributos que modifiquen las funcionalidades de los elementos.

7 Referencias

[1] Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing.

[2] *Class ArrayList<E>*. (s.f.). Obtenido de https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

[3] *Class LinkedHashMap<K,V>*. (s.f.). Obtenido de https://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html

[4] *Class TreeSet<E>*. (s.f.). Obtenido de https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html

[5] *Creating Internal DSLs in Java, Java 8- Adopting Martin Fowler's approach*. (s.f.). Obtenido de https://sanaulla.info/2013/05/30/creating-internal-dsls-in-java-java-8-adopting-martin-fowlers-approach/

[6] *Eclipse Modeling Framework (EMF)*. (s.f.). Obtenido de https://eclipse.org/modeling/emf/

[7] *Eclipse Modeling Framework, Wikipedia*. (s.f.). Obtenido de https://en.wikipedia.org/wiki/Eclipse_Modeling_Framework

[8] *Ecore*. (s.f.). Obtenido de http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#package_description

[9] Fowler, M., & Parsons, R. (2011). *Domain-Specific Languages.* Addison Wesley.

[10] Gubaer - Own work based on UML 2.0 Infrastructure Specification, S. 3. (s.f.). *Wikipedia*. Obtenido de https://commons.wikimedia.org/w/index.php?curid=4754540

[11] *Interface BiConsumer<T,U>*. (s.f.). Obtenido de https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html

[12] *Interface Consumer<T>*. (s.f.). Obtenido de https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

[13] *Java™ Platform, Standard Edition 8*. (s.f.). Obtenido de http://docs.oracle.com/javase/8/docs/api/

[14] *La programación no es un arte*. (s.f.). Obtenido de http://laprogramacionnoesunarte.blogspot.com.es/p/interfacesfuncionales-index-e-c-z-1.html

[15] ORACLE. (s.f.). *What's New in JDK 8*. Obtenido de http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

[16] Stahl, T., & Völter, M. (2006). *Model-Driven Software Development.* John Wiley & Sons.

[17] Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse Modeling Framework.*

[18] Tratt, L. (2008). Domain Specific Language Implementation via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. Obtenido de http://dl.acm.org/citation.cfm?id=1391958

[19] *UML*. (s.f.). Obtenido de http://www.uml.org/what-is-uml.htm

[20] Vasudevan, N., & Tratt, L. (s.f.). *Comparative Study of DSL Tools.* Obtenido de http://www.sciencedirect.com/science/article/pii/S1571066111000788

[21] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C., . . . Wachsmuth, G. (s.f.). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.*

[22] *What is a Domain-Specific Language?* (s.f.). Obtenido de http://www.javaworld.com/article/2077865/core-java/core-java-creating-dsls-in-java-part-1-what-is-a-domain-specific-language.html

[23] *Xtend*. (s.f.). Obtenido de https://eclipse.org/xtend/documentation/

[24] *Xtext*. (s.f.). Obtenido de https://eclipse.org/Xtext/

Glosario

API	Application Programming Interface
DSL	Domain-Specific Language
MDSD	Model-Driven Software Development
EMF	Eclipse Modeling Framework
EBNF	Extended Backus-Naur Form
OCL	Object Constraint Language
IDE	Integrated Development Environment

Anexos

A Sintaxis concreta

En este anexo se muestra la sintaxis concreta completa del proyecto.

⟨*SemanticModel*⟩ ::= 'SemanticModel' ⟨*ID*⟩ '['⟨*Methods*⟩']' '{' (⟨*Class*⟩)* '}';

⟨*Methods*⟩ ::= 'chainingMethod'
| 'nestedMethod'
| 'lambdaMethod';

⟨*Class*⟩ ::= ('top')? 'Class' ⟨*ID*⟩ ('extends' ⟨*STRING*⟩)? '{' (⟨*Element*⟩)* '}';

⟨*Element*⟩ ::= ⟨*Object*⟩
| ⟨*Code*⟩;

⟨*Object*⟩ ::= ⟨*Collection*⟩
| ⟨*Variable*⟩;

⟨*Collection*⟩ ::= ⟨*List*⟩
| ⟨*Set*⟩
| ⟨*Map*⟩;

⟨*Code*⟩ ::= ⟨*ID*⟩ ':' 'Code' '[' 'context' ⟨*STRING*⟩ (',' ⟨*STRING*⟩ 'on events')? ']' ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*Variable*⟩ ::= ('id')? ⟨*ID*⟩ ':' ⟨*ValueType*⟩ ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*List*⟩ ::= ⟨*ID*⟩ ':' ⟨*ValueType*⟩ '[*]' ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*Set*⟩ ::= ⟨*ID*⟩ ':' ⟨*ValueType*⟩ 'ordered' ('by' ⟨*STRING*⟩)? ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*Map*⟩ ::= ⟨*ID*⟩ ':' ⟨*PrimitiveType*⟩ '-¿' ⟨*ValueType*⟩ ('{' ⟨*Attribute*⟩ (',' ⟨*Attribute*⟩)* '}')? ';';

⟨*Attribute*⟩ ::= ⟨*ID*⟩;

$$
\begin{array}{lll}
\langle \mathit{ValueType} \rangle & ::= & \langle \mathit{PrimitveType} \rangle \\
& | & \langle \mathit{ClassType} \rangle; \\
\\
\langle \mathit{ClassType} \rangle & ::= & \langle \mathit{STRING} \rangle; \\
\\
\langle \mathit{PrimitiveType} \rangle & ::= & \langle \mathit{PrimitiveTypes} \rangle; \\
\\
\langle \mathit{PrimitiveTypes} \rangle & ::= & \text{'byte'} \mid \text{'Byte'} \\
& | & \text{'short'} \mid \text{'Short'} \\
& | & \text{'int'} \mid \text{'Integer'} \\
& | & \text{'long'} \mid \text{'Long'} \\
& | & \text{'float'} \mid \text{'Float'} \\
& | & \text{'double'} \mid \text{'Double'} \\
& | & \text{'boolean'} \mid \text{'Boolean'} \\
& | & \text{'char'} \mid \text{'Character'} \\
& | & \text{'String'};
\end{array}
$$