UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Un lenguaje de dominio específico para la definición de métricas para lenguajes de modelado

Manuel Cobos Fernández
Tutor: Esther Guerra Sánchez

Mayo 2017

Un lenguaje de dominio específico para la definición de métricas para lenguajes de modelado

AUTOR: Manuel Cobos Fernández
TUTOR: Esther Guerra Sánchez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2017

Resumen (castellano)

La calidad del software se ha vuelto fundamental para la Ingeniería de Software, de forma que la calidad en los procesos de desarrollo tiene cada vez más recursos para su aseguramiento.

Una forma de determinar la calidad del software utilizada con frecuencia es con el uso de métricas. Aplicado a los recursos software durante el ciclo de vida, pueden producir evaluaciones como base para la posterior evaluación de las propiedades de calidad.

En este proyecto se desarrolla un DSL (*Domain-Specific Language)* que proporciona operaciones para la definición de métricas sobre modelos.

Los lenguajes de dominio específico son lenguajes más pequeños y enfocados a resolver una tarea específica de un sistema software.

Este lenguaje, denominado MyMtr, ofrece la posibilidad de definir las métricas deseadas sobre distintos meta-modelos. Desde contar el número de cierto tipo objetos o el número de atributos de ese tipo de objetos (con una profundidad ilimitada en la recursión sobre los atributos), hasta condicionar las cuentas mencionadas. También se pueden definir métricas que sean resultado de operaciones de otras métricas, ya sean unarias (media, calcular los elementos diferentes, suma de elementos) o binarias (suma, resta, división…), y se pueden definir umbrales para cada una de ellas, lo cual hace más expresivo el resultado obtenido al poder marcar las métricas que no se cumplen con los limites definidos.

Aparte de definir las métricas, MyMtr genera un plugin que proporciona una vista para Eclipse cuya funcionalidad es realizar el cálculo de las métricas definidas sobre los modelos deseados, y además la posibilidad de exportar los resultados a un fichero *.csv*.

Palabras clave (castellano)

Lenguaje de Dominio Específico, modelo, meta-modelo, generación de código, métrica.

Abstract (English)

Software quality has become essential in Software Engineering so that development processes have increasingly more resources to ensure it.

A frequently used procedure to determine software quality is by using metrics. Applied to software resources throughout their life cycle, they can produce evaluations as a starting point to further evaluation procedures of quality properties.

In this project, we develop a DSL (Domain Specific Language) to provide operations to define metrics on models.

Domain Specific Languages are smaller languages focused on solving a specific task in a software system.

This language, called MyMtr offers the option to define desirable metrics on different meta-models. From telling the number of times a specific kind of object occurs or the number of attributes of that kind of object (with an unlimited depth in attributes recursion parameters) to conditionate the operation mentioned above. It is also possible to define metrics from the result of other unary metric operations (average, to calculate different elements) or binary metric operation (addition, subtraction, division…), and it is also possible to define threshold values for each one of them that is essential to obtain a more expressive and meaningful result as we can mark those metrics which do not verify predefined limits.

Apart from defining metrics, MyMtr generates a plugin that provides a view for Eclipse whose functionality is to calculate defined metrics over desirable models and also provide the possibility to export the results in a .csv. format file.

Keywords (inglés)

Domain Specific Language, model, meta-model, code generation, metric.

INDICE DE CONTENIDOS

1 Introducción ... 1
1.1 Motivación ... 1
1.2 Objetivos ... 1
1.3 Organización de la memoria ... 2
2 Tecnologías a utilizar y conceptos previos ... 3
2.1 Desarrollo Dirigido por Modelos ... 3
    2.1.1 Modelo ... 4
    2.1.2 Lenguajes de Modelado ... 4
    2.1.3 Meta-modelo ... 5
    2.1.4 Generación de código ... 5
2.2 Eclipse Modeling Framework ... 5
    2.2.1 Ecore y Genmodel ... 5
    2.2.2 Xtext ... 6
    2.2.3 Xtend ... 7
2.3 Desarrollo de plugins para Eclipse ... 7
    2.3.4 Estructura del Plugin ... 8
3 Diseño ... 9
3.1 Arquitectura MyMtr ... 9
    3.1.1 Meta-modelo ... 9
    3.1.2 Lenguaje ... 10
    3.1.3 Generador de Código ... 13
3.2 Arquitectura MetricAnalyzer ... 13
    3.2.1 Modelo ... 14
    3.2.2 Vista ... 14
    3.2.3 Controlador ... 14
4 Desarrollo ... 15
4.1 Funcionalidades ... 15
4.2 Desarrollo MyMtr ... 17
    4.2.1 Ecore ... 17
    4.2.2 Xtext ... 18
    4.2.3 Xtend ... 19
5 Pruebas y resultados ... 23
6 Conclusiones y trabajo futuro ... 29
6.1 Conclusiones ... 29
6.2 Trabajo futuro ... 29
Referencias ... 31
Glosario ... 33
Anexos ... I
A Sintaxis Concreta MyMtr ... I
B Métricas definidas con MyMtr para diagramas de clases ... III

INDICE DE FIGURAS

FIGURA 3-1. DEFINICIÓN SINTAXIS DEL LENGUAJE 1 11

FIGURA 3-2. DEFINICIÓN SINTAXIS DEL LENGUAJE 2 11

FIGURA 3-3. DEFINICIÓN SINTAXIS DEL LENGUAJE 3 12

FIGURA 3-4. DEFINICIÓN SINTAXIS DEL LENGUAJE 4 12

FIGURA 3-5. RESULTADO DE APLICAR COUNT Y COUNTBYFIRST 13

FIGURA 4-1. LENGUAJE MYMTR EJEMPLO 1 15

FIGURA 4-2. LENGUAJE MYMTR EJEMPLO 2 16

FIGURA 4-3. LENGUAJE MYMTR EJEMPLO 3 16

FIGURA 4-4. META-MODELO DEL PROYECTO MYMTR 17

FIGURA 4-5. DEFINICIÓN DE LA SINTAXIS EN XTEXT 1 18

FIGURA 4-6. DEFINICIÓN DE LA SINTAXIS EN XTEXT 1 18

FIGURA 4-7. DEFINICIÓN DE LA SINTAXIS EN XTEXT 3 19

FIGURA 4-8. GENERADOR DE CÓDIGO 19

FIGURA 4-9. GENERADOR DEL CÓDIGO DEL PLUGIN 20

FIGURA 4-10. PROVEEDOR DE CONTENIDO PARA LA MÉTRICA COUNT 21

FIGURA 4-11. VALIDADOR DE LOS NOMBRES DE LAS MÉTRICAS 22

FIGURA 5-1. META-MODELO PARA LA DEFINICIÓN DE DIAGRAMAS DE CLASES 23

FIGURA 5-2. DIAGRAMA DE CLASES DE EJEMPLO 24

FIGURA 5-3. RESULTADO DEL CÁLCULO DE MÉTRICAS 1 25

FIGURA 5-4. LENGUAJE MYMTR EJEMPLO 4 26

FIGURA 5-5. RESULTADO DEL CÁLCULO DE MÉTRICAS 2 26

FIGURA 5-6. LENGUAJE MYMTR EJEMPLO 5 27

FIGURA 5-7. RESULTADO DEL CÁLCULO DE MÉTRICAS 3 27

1 Introducción

1.1 Motivación

La calidad del software se ha convertido en esencial para la Ingeniería de Software, de manera que cada vez se obtienen más recursos para las tareas relacionadas con el aseguramiento de la calidad en los procesos de desarrollo de software. En particular, la evaluación temprana y continua de la calidad puede proporcionar indicadores cuantitativos para la calidad del modelo y ayudar a localizar problemas estructurales.

Sin embargo, la medición de ciertas propiedades del software es una tarea costosa, de tiempo y de recursos, ya que el soporte de herramientas para la medición de calidad automatizada sigue creciendo, aun se requiere de mucho trabajo manual.
En Desarrollo Dirigido por Modelos (MDD) [14] el modelo es el artefacto principal del proceso de desarrollo. La calidad de los modelos involucrados tiene una influencia significativa en la calidad del software final [1], ya que el software se genera automáticamente a partir de los modelos, y por tanto, se necesita asegurar que los modelos cumplen ciertos criterios de calidad. Dichos criterios de calidad pueden depender del dominio de aplicación y del lenguaje de modelado concretos.

Un medio utilizado con frecuencia para determinar la calidad del software es con el uso de métricas [13]. Aplicado a varios artefactos del software durante su ciclo de vida completo, pueden producir evaluaciones comparables de estos artefactos como base para evaluaciones posteriores de propiedades de calidad [2].

En este proyecto se construye un DSL (*Domain-Specific Language)* que permite definir métricas para lenguajes de modelado definidos mediante meta-modelado [15].

Este DSL proporciona primitivas de alto nivel que facilitan la definición de dichas métricas y a partir de la cual se generara una vista donde se podrá visualizar el resultado de calcular las métricas sobre un modelo en concreto, además de detectar que valores son límite para cada una de las métricas.

1.2 Objetivos

Este proyecto pretende proporcionar un lenguaje útil en la definición de métricas que nos permita asegurar la calidad de cualquier modelo que quiera ser analizado.
Para ello se proporcionan primitivas que facilitan su definición, como por ejemplo contar el número de objetos de un tipo dado.

Por otro lado, también se quiere poder visualizar los resultados de una manera clara, para ello se genera automáticamente de nuestra definición de métricas, un plugin que nos proporciona una vista de eclipse, en la que se podrán ver las diferentes métricas y los valores correspondientes al modelo que se esté analizando. Además se podrá localizar para los valores que exceden los umbrales definidos para cada métrica.

También se pretende poder exportar los datos en un formato de fichero *.csv,* para recoger y guardar los resultados, para no tener la necesidad de acceder a ellos reanalizando el modelo.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 2**: Conceptos previos y tecnologías a utilizar, necesarias para el desarrollo del proyecto
- **Capítulo 3**: Diseño del proyecto y del producto final generado.
- **Capítulo 4**: Proceso de desarrollo seguido en la implementación.
- **Capítulo 5**: Pruebas y resultados.
- **Capítulo 6**: Conclusiones y trabajo futuro.

2 Tecnologías a utilizar y conceptos previos

En este capítulo se van a dar a conocer las tecnologías necesarias para el desarrollo del proyecto así como los conceptos previos fundamentales para el correcto entendimiento de la funcionalidad y propósito de este trabajo.

Más concretamente, se van a explicar los siguientes apartados:

- **Desarrollo Dirigido por Modelos**: (MDD), prestando especial atención a los lenguajes de dominio específico (DSL) y la generación de código.

- **Eclipse Modeling Framework** (EMF): Ecore, Xtext y Xtend.

- **Desarrollo de plugins para Eclipse**

2.1 Desarrollo Dirigido por Modelos

El desarrollo dirigido por modelos (MDD) es una propuesta para la construcción de software en la que se les atribuye a los modelos el papel principal del proceso, frente a las propuestas más tradicionales basadas en lenguajes de programación, plataformas de objetos y componentes software [16].

Hay varias razones que han motivado la aparición de este nuevo paradigma. Tenemos en primer lugar la cada vez mayor complejidad de las aplicaciones de software, que han de satisfacer un mayor número de requisitos con mejores prestaciones y menos tiempos de desarrollo, por otra parte tenemos que las nuevas tecnologías evolucionan muy rápido lo que hace que las inversiones en las tecnologías concretas sean demasiado volátiles. Para solucionar estos problemas está comprobado que la mejor forma es elevando el nivel de abstracción de los modelos desde las primeras etapas del desarrollo [17].

Dentro de los objetivos principales del enfoque MDD se encuentran [4]:

- Mejorar la calidad del software.
- Mejorar la mantenibilidad del software.
- Mejorar los niveles de reusabilidad.
- Manejo de la complejidad – abstracción.
- Productividad.
- Interoperabilidad.

2.1.1 Modelo

Tal y como se recoge en [18], un modelo es una representación simplificada de una determinada realidad. Esa realidad representada en el modelo que puede ser cualquier tipo de sistema: no solo un sistema software, sino también un sistema humano, un sistema mecánico, o un sistema mixto con ambos elementos.

Un modelo es una abstracción, es decir, una simplificación utilizada para comprender mejor la realidad que representa [5]. Un modelo simplifica la realidad suprimiendo detalles irrelevantes y reteniendo los aspectos esenciales de modo que la esencia del sistema sea mejor conocida y podamos hacer frente a su complejidad.

Un modelo debe ser [6]:

- **Abstracto**, es decir, ser una versión reducida del sistema que representa.
- **Comprensible**, expresado de tal forma que se pueda entender fácilmente.
- **Preciso**, representa fielmente el sistema de modelado.
- **Predictivo**, se puede utilizar para obtener conclusiones correctas sobre el sistema.

2.1.2 Lenguajes de Modelado

Los lenguajes de modelado permiten definir los modelos, es decir, un lenguaje que permite representar la realidad de una manera simplificada y focalizada en los aspectos relevantes del problema a resolver.
El modelo representa una parte del sistema de estudio, y de modo acorde a las reglas del lenguaje de modelado. Esto conduce a que una definición lo más rigurosa posible del lenguaje es esencial para obtener un modelo lo más ajustado al sistema real analizado.

Una de las principales clasificaciones de los lenguajes de modelado son por su ámbito:

- **Lenguajes de Modelado de Propósito General**. *General Purpose Modeling Languages* (GPL), es un lenguaje de modelado diseñado para ser utilizado para crear modelos sobre una gran variedad de dominios de aplicación
- **Lenguajes de Modelado de Dominio Específico**. *Domain Specific Modeling Language* (DSL), cualquier lenguaje que esté especializado en modelar o resolver un conjunto específico de problemas.

La mayor parte de los lenguajes de programación no se pueden considerar DSL ya que están diseñados para resolver cualquier tipo de problema, en cambio el objetivo de DSL es resolver un conjunto específico de problemas.

El uso de lenguajes GPL ha tenido gran aceptación y éxito ya que poseen la ventaja de resolver cualquier tipo de problema, sin embargo no todos son igualmente sencillos de resolver.

2.1.3 Meta-modelo

Para poder realizar modelos resulta necesario un meta-modelo del lenguaje.

El meta-modelo es una descripción de todos los conceptos que pueden usarse en el mismo, y mediante el lenguaje de modelado definido tener como resultado los diferentes modelos posibles que se ajustan a nuestro dominio.

El meta-modelado consiste básicamente en dotar a los creadores de modelos de abstracciones apropiadas para su dominio particular, esto implica definir conceptos más específicos [19].

2.1.4 Generación de código

El último paso del desarrollo dirigido por modelos es conseguir que el modelo obtenido sea interpretado y ejecutado.

Para ello, una vez obtenido el modelo, se interpreta cada elemento y se genera el código correspondiente en lenguaje de alto nivel para poder ser ejecutado posteriormente.

2.2 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) es un framework de Eclipse que nos proporciona las herramientas necesarias para construir aplicaciones basadas en MDD.

La versión de EMF que se ha utilizado para desarrollar el lenguaje es la **2.11.2**

Ahora vamos a explicar los elementos de EMF utilizados en este proyecto.

2.2.1 Ecore y Genmodel

El meta-modelo EMF consta de dos partes, el *ecore* y los archivos de descripción *genmodel*.

El archivo *ecore* contiene la información sobre las clases definidas. El archivo *genmodel* contiene información adicional para la generación del código, por ejemplo, la ruta y la información del archivo.

El archivo *ecore* permite definir los siguientes elementos.

- **EClass**: representa una clase, que puede contener atributos o referencias.
- **EAttribute**: representa un atributo que tiene un nombre y un tipo.
- **EReference**: representa el extremo de una asociación entre dos clases. Tiene propiedades para indicar si se trata de una contención, y una clase de referencia a la que apunta.
- **EDataType**: representa el tipo de un atributo, por ejemplo, int , float o java.lang.String

A partir del *ecore* se genera el *genmodel* y a partir de este se crean las clases, que van a ser el modelo de la aplicación, y el editor, donde vamos a definir nuestro lenguaje de modelado con el uso de Xtext.

Esta funcionalidad de EMF nos permite crear un meta-modelo para el proyecto, el cual será la base del lenguaje a definir.

Una de las ventajas que proporciona EMF es que ofrece una API que permite acceder reflexivamente a la definición de la clase y atributos de un objeto en tiempo de ejecución, sin necesidad de haber generado previamente las clases de implementación de ese objeto, es decir, sin utilizar las clases que se generan a partir de los *ecore* correspondiente [11].

2.2.2 Xtext

La versión elegida de Xtext para definir el lenguaje del proyecto a partir del *ecore* es la **2.8.4**.

Xtext [8] es un framework (o herramienta) para la creación de lenguajes, hecho en java y está conectado con otros proyectos para poder desarrollar modelos o meta-modelos (EMF).

Provee soporte para la creación de toda la infraestructura necesaria para desarrollar un lenguaje como:

- Compiladores
- Intérpretes
- Editores y herramientas de tipo IDE, a través de la integración con Eclipse.
- Se basa en desarrollo iterativo.

Permite refinar diferentes aspectos del lenguaje:

- Revisiones y validaciones.
- Plantillas de formato para el editor de texto
- Resaltado y coloreado de sintaxis
- Iconos e imágenes para cada entidad en el editor
- Asistente de contenido (autocompletado)

Xtext integra facilidades para la generación de código.
El objetivo del generador es, justamente a partir del input, generar un output. Por lo general, una cosa que se puede hacer es generar código fuente Java.

En el proyecto realizado, permite definir el lenguaje de una manera versátil y con funcionalidades útiles para obtener un lenguaje sencillo e intuitivo pero a la vez completo.

2.2.3 Xtend

La versión usada, al igual que en Xtext, es la **2.8.4**.

Xtend [9] **e**s un lenguaje de tipado estático para la plataforma Java.
Integra características para reducir el ruido al escribir, como evitar paréntesis, punto y coma y returns. También se puede hacer sobrecarga de operadores.

El objetivo de Xtend no es remplazar a Java, sino ser una alternativa en algunas situaciones en las que Java no es lo suficientemente útil.

Xtend compila a código java legible, es decir, en lugar de generar bytecode directamente, Xtend pasa nuestro código *.xtend* a otro fichero *.java* con la sintaxis habitual de java.

Esta característica es idónea para nuestro generador de código ya que nuestro código resultante es un plugin para eclipse desarrollado en java.
También usamos Xtend para definir las diferentes opciones de autocompletado y las restricciones que se explicaran más adelante.

2.3 Desarrollo de plugins para Eclipse

Este apartado es importante ya que tanto el proyecto en sí, como el proyecto que va a generar son ambos plugins para Eclipse.

El propio Eclipse nos permite desarrollar plugins. Cada plugin extiende de algún modo la funcionalidad de Eclipse.

Una de las características importantes de los plugin es su interoperabilidad, es decir que pueden usar servicios de otros plugins o por el contrario que sus funcionalidades sean extendidas por otros.

La versión de Eclipse que se ha utilizado para el desarrollo es **Mars.1 Release (4.5.1).**

2.3.4 Estructura del Plugin

Un plugin es una aplicación java que se empaqueta en un archive *.jar,* en el cual se encuentran los archivos necesarios para su ejecución e integración en eclipse:

- **Plugin.xml** (Plugin Manifest). Es el archivo donde se le da al plugin sus propiedades, tales como nombre, versión, autor, identificador. Este archivo se encarga de informar a Eclipse que la aplicación es un plugin y las clases que debe ejecutar para su correcto funcionamiento. Se pueden definir también puntos de extensión, es decir, de qué manera otros plugins podrán usar su funcionalidad para extenderla.
- **Código java.** Son los archivos java del proyecto, son los encargados de realizar la funcionalidad correspondiente del plugin.
- **Imágenes y otros recursos.** Cualquier recurso que complemente nuestro plugin, ya sean imágenes, archivos de configuración, etc.

3 Diseño

El siguiente capítulo está destinado a la especificación del diseño del proyecto.

Al proyecto que genera el lenguaje destinado a la definición y cálculo de métricas para modelos se le ha denominado MyMtr y a partir de ahora se recurrirá a él de esta manera.

No obstante, no solo se va a centrar en el diseño de MyMtr, sino también en el diseño elegido para el proyecto generado por el lenguaje, al cual se hará referencia como MetricAnalyzer.

3.1 Arquitectura MyMtr

La arquitectura del proyecto está basada en MDD y por tanto tiene las siguientes partes:

- **Sintaxis abstracta**, es el meta-modelo necesario para definir los futuros modelos
- **Sintaxis concreta**, es el lenguaje de modelado del proyecto.
- **Semántica**, es el generador de código que hace que se genere MetricAnalyzer de manera automática.

3.1.1 Meta-modelo

Con fin de definir la sintaxis abstracta del lenguaje se definen los siguientes elementos esenciales para el meta-modelo:

- **Metrics**, es el componente que será la raíz del modelo, ya que gestionará las diferentes métricas, umbrales o las maneras de mostrar cada tipo de objeto.
- **Metric**, este elemento es la base de la que extenderán las diferentes métricas para generar un valor, definiendo una operación que realizar sobre el modelo deseado.
- **Threshold,** se encargará de definir un umbral de valores para las diferentes métricas.
- **Showby,** su misión es proporcionar versatilidad a la hora de mostrar cada uno de los objetos sobre los que actúan las diferentes métricas, es decir, sobre un tipo de objetos se puede seleccionar por qué atributo se mostrarán.

El resto de elementos del meta-modelo extienden la funcionalidad de **Metric** y algunos necesitan de algún elemento extra como por ejemplo:

- **Node,** se utiliza para enlazar unos elementos del modelo con otros hasta llegar al requerido para obtener el valor. **Node** es usado por casi todas las métricas definidas en el proyecto y aporta una profundidad máxima en la inspección de los elementos.
- **Closure,** proporciona la posibilidad de navegar recursivamente a través de una misma referencia.
- **If,** necesario para poder condicionar la búsqueda de los diferentes objetos del modelo

Por ultimo también se definen algunos elementos para realizar operaciones entre las diferentes métricas, por ejemplo:

- **UnaryOperation**, se encarga de gestionar las operaciones unarias, es decir para una métrica con diferentes valores para cada objeto se agrupan esos valores acorde a la operación definida.
- **UnaryOp,** correspondiente enumerado para definir las diferentes operaciones disponibles a realizar.

3.1.2 Lenguaje

Para la definición de la sintaxis del lenguaje se ha utilizado la *Notación de Backus-Naur Extendido – Extended Backus-Naur Form (EBNF).*

En este apartado se van a ilustrar algunos ejemplos ya que la notación completa se incluye en el Anexo A.

En la figura 3-1 se puede ver el lenguaje para los elementos esenciales del meta-modelo.

Se puede ver como se indica el meta-modelo para el que se definirán las métricas, porque eso permite hacer revisión de tipos de las clases, atributos y referencias mencionados en la definición de las métricas (es decir, sólo pueden utilizarse tipos correctos en la definición de las métricas).

```
<Metrics>          ::= 'Metamodel' <STRING>
                       (<Showby>)*
                       <Metric>
                       (<Metric> | <Threshold>)*;

<Threshold>        ::= 'Threshold for ' (<Metric>|<ID>)
                       '('<FLOAT>','<FLOAT>')';

<Showby>           ::= 'SHOW(' <Class> '.' <Attribute> ')';

<Metric>           ::= <Count >
                     |  <Countbyfirst>
                     |  <Countbylast>
                     |  <ClosureCountbyfirst>
                     |  <IfCountbyfirst>
                     |  <Operation>;
```

Figura 3-1. Definición sintaxis del lenguaje 1

Para continuar se va a mostrar los elementos especiales **Node Closure** e **If**, de los cuales se ha hablado en el apartado anterior, y son usados por algunas métricas específicas para conseguir su objetivo.

En el caso de **Node** se puede ver que se puede definir un *type* o *supertype*, esto es, para un elemento del objeto de tipo agrupación, se definen los tipos que interesan para la métrica que se está definiendo.
En el caso de *type*, se escogen solo los objetos que sean del tipo elegido. En el caso de *supertype*, se escogerán todos los objetos cuyo supertipo sea ese tipo, es decir, cualquier clase que herede del tipo elegido se tendrá en cuenta de cara a calcular la métrica.

```
<Node>             ::= '.'<StructuralFeature>
                       ('(supertype->'<Class>')' | '(type->'<Class>')')?
                       ('['<Attribute>'='<STRING>']')?;

<Closure>          ::= 'CLOSURE('<Class> (<Node>)*')';

<If>               ::= 'IF(' <Class>(<Node>)*')';
```

Figura 3-2. Definición sintaxis del lenguaje 2

En la figura 3-3 veremos dos enumerados para tratar las diferentes operaciones definidas en MyMtr.

```
<UnaryOP>            ::= 'SUM'
                       | 'AVG'
                       | 'MAX'
                       | 'MIN'
                       | 'DIFF';

<BinaryOP>           ::= '+'
                       | '-'
                       | '/'
                       | '*';
```

Figura 3-3. Definición sintaxis del lenguaje 3

Por último, se van a mostrar algunas de las extensiones de **Metric** utilizadas para, contar el número de objetos (*Count*) y para contar el número de objetos pero agrupando los valores por el objeto raíz que contiene los objetos contados (*Countbyfirst*).

También se puede ver que se pueden definir las métricas como auxiliares, tan solo con poner *Auxiliary* antes de definirla. Esto permite crear métricas intermedias en el cálculo de otras (haciendo uso de las operaciones entre métricas) y que finalmente no se muestren en el plugin resultante porque no son útiles por sí mismas.

```
<Count>              ::= ('Auxiliary ')? 'Metric' <STRING> '='
                           'COUNT('<Class>
                           ('['<Attribute>'='<STRING>']')?
                           (<Node>)*')'
                          '{'<Description>'}';

<Countbyfirst>       ::= ('Auxiliary ')? 'Metric' <STRING> '='
                           'COUNTBYFIRST('<Class>
                           ('['<Attribute>'='<STRING>']')?
                           (<Node>)*')'
                          '{'<Description>'}';
```

Figura 3-4. Definición sintaxis del lenguaje 4

En la figura 3-4 se puede ver un pequeño ejemplo para mostrar la diferencia entre ambas definiciones.

En el ejemplo de la figura 3-5 se definen dos métricas para modelos del tipo de diagrama de clases.
En el primer caso se calculan los atributos totales de todas las clases, mientras que en el segundo se calculan los atributos por cada clase.

Metrics	Description	Value
ATT	Numero de atributos	8
◢ ATTBYCLASS	Numero de atributos por clase	
Tarjeta		1
TarjetaBancaria		2
TarjetaCredito		0
TarjetaDebito		1
Cliente		2
Movimiento		2

Figura 3-5. Resultado de aplicar Count y Countbyfirst

3.1.3 Generador de Código

Para generar el código se ha decidido dividirlo por ficheros, es decir que para cada fichero que este en el proyecto generado se tendrá su correspondiente función que lo genere.

Se tiene una función raíz que gestiona que ficheros deberán ser creados para cada situación y con qué datos variables como por ejemplo la ruta en la que deben ser creados, esto nos facilita el empaquetado de cada fichero y tener un producto final de calidad.

3.2 Arquitectura MetricAnalyzer

Este proyecto es el fruto de la generación automática de nuestro lenguaje creado en MyMtr e igual que este, también en su plugin que extenderá las funcionalidades de Eclipse.

Este trata de una vista de Eclipse en la que se verán las diferentes métricas definidas por MyMtr y se obtendrán los valores del fichero abierto en Eclipse. También dispondrá de la opción de exportar los datos a *.csv*.

Su diseño está basado en el patrón MVC (modelo, vista, controlador).
También se hace uso de patrones como *FactoryMethod* y *Singleton* para diseñar el código lo más eficientemente posible.

En las siguientes secciones se explicaran con más detalle cada uno de ellos.

3.2.1 Modelo

Se hace uso del patrón *Singleton* para diseñar una instancia de **Métricas**, y se encarga de gestionar cada una de las métricas del proyecto.

Cada métrica en particular tiene definido su nombre y su descripción y tiene también dos funciones para obtener su valor o sus diferentes opciones, en el caso de que las tenga, con su respectivo valor.

La manera en la que se obtienen los valores y las diferentes opciones es con dos factorías, una encargada para los valores de las métricas y otra para las opciones de estas.

Dentro del modelo también tenemos un módulo encargado de transformar los *path* o rutas de los modelos y meta-modelos, en recursos utilizables por la aplicación.

3.2.2 Vista

Para la vista se ha usado como modelo un árbol, de este modo se pueden mostrar las métricas con sus respectivas opciones, si las tuvieran, de una forma sencilla y con la posibilidad de desplegar o replegar los elementos.

Con esto se consigue una aplicación mucho más visual e intuitiva que permite centrarse en las métricas que interesen para cada caso en concreto.

3.2.3 Controlador

El controlador se encarga de enlazar la vista con el modelo.
Con las diferentes acciones del usuario que interactúa con la aplicación, el controlador hace uso del modelo y de la vista, y actualiza el contenido y/o la forma de mostrarlo al usuario según corresponda.

4 Desarrollo

Para el desarrollo del lenguaje MyMtr se ha utilizado EMF que facilita la implementación de cada uno de los elementos definidos en el capítulo anterior.

En el caso de MetricAnalyzer se va a usar Eclipse que va a facilitar el desarrollo de plugins y la instalación de estos.

Mas globalmente para el desarrollo del proyecto se utiliza un ciclo de vida de prototipado. De esta manera primero se tiene una aplicación básica funcional al principio y se irán añadiendo diferentes funcionalidades en continuas iteraciones.

En los siguientes apartados se explicaran las funcionalidades implementadas en el proyecto y también el proceso de desarrollo de cada uno de los subproyectos que lo forman como ya se ha hecho con el capítulo anterior.

4.1 Funcionalidades

Como se ha mencionado el ciclo de vida elegido es el de prototipado y por ello se van a comentar las diferentes funcionalidades desarrolladas en el proyecto.

El prototipo inicial consta de un lenguaje básico que permite contar el número de objetos de un modelo que instancie el meta-modelo definido en nuestro lenguaje. Además de contar, el plugin generado facilitará las cosas para poder visualizar las métricas con respecto al modelo analizado.

```
Metamodel "/prueba/ClassDiagram.ecore"

Metric DSC = COUNT(Class){"Numero de clases"}
```

Figura 4-1. Lenguaje MyMtr ejemplo 1

En las siguientes iteraciones del ciclo se implementan las diferentes versiones de la métrica inicial, las cuales permiten contar por el primer elemento de la ruta a contar (Ej. contar el número de atributos que tiene cada clase), contar agrupados por el último elemento de la ruta (Ej. contar el número de clases hija), la posibilidad de contar a través de ciclos (Ej. contar el número de ancestros de una clase), contador condicionado (Ej. contar el número de métodos públicos que tiene cada una de las clases heredadas).

Más adelante se ha implementado la posibilidad de proporcionar un umbral de validez, de tal manera que definiéndolo, en nuestro resultado final se mostrarán en rojo las métricas que no cumplan la condición.

```
Metamodel "/prueba/ClassDiagram.ecore"

Metric DSC = COUNT(Class){"Numero de clases"}

Threshold for DSC(1.0,3.5)
```

Figura 4-2. Lenguaje MyMtr ejemplo 2

Otra de las características insertadas es la capacidad de mostrar los objetos por el atributo deseado, aunque se puede no definir y por defecto se mostraran por el atributo "name".

Más adelante se implementan las operaciones entre métricas y con ello llega también la posibilidad de hacer que algunas de las métricas definidas no se muestren en el resultado final, así pues se define un boolean Auxiliar que nos permite elegir si una métrica se mostrará o no en el producto final.

Para terminar se añade la funcionalidad de exportar a un fichero *.csv* para tener los datos guardados y accesibles.

En la figura 4-3 se pueden ver algunos ejemplos para métricas sobre diagramas de clases y de las funcionalidades mencionadas.

```
Metamodel "/prueba/ClassDiagram.ecore"

SHOW(Class.name)

Metric DSC = COUNT(Class){"Numero de clases"}

Threshold for DSC(1.0,3.5)

Metric NAS = COUNTBYFIRST(Class.features(type->Association))
                              {"Numero de asociaciones de una clase"}

Auxiliary Metric AUXAPPM = COUNTBYFIRST(Operation.params){""}
Metric APPM = AVG(AUXAPPM){"Numero medio de parametros por operacion"}

Metric DIT = COUNTBYFIRST(CLOSURE(Class.levels(type->Generalization).super))
                    {"Longitud desde la clase hasta raiz de arbol de herencia"}

Metric NOC = COUNTBYLAST(Generalization.super){"Numero de hijos de una clase"}
```

Figura 4-3. Lenguaje MyMtr ejemplo 3

4.2 Desarrollo MyMtr

Como se ha dicho anteriormente, para el desarrollo de los diferentes elementos necesarios se ha utilizado:

- Para definir el meta-modelo se usa el editor *Sample Ecore Model Editor* y obtenemos el *ecore* final.
- En el caso del lenguaje, tenemos Xtext para definirlo.
- Tanto la generación de código, como las restricciones, autocompletado y proveedor de contenido se desarrollan con el lenguaje Xtend.

4.2.1 Ecore

Para ilustrar el meta-modelo definido con el editor proporcionado por Eclipse se va a usar la figura 4-4, en la que se pueden ver el diagrama de los diferentes elementos que compondrán el *ecore.*

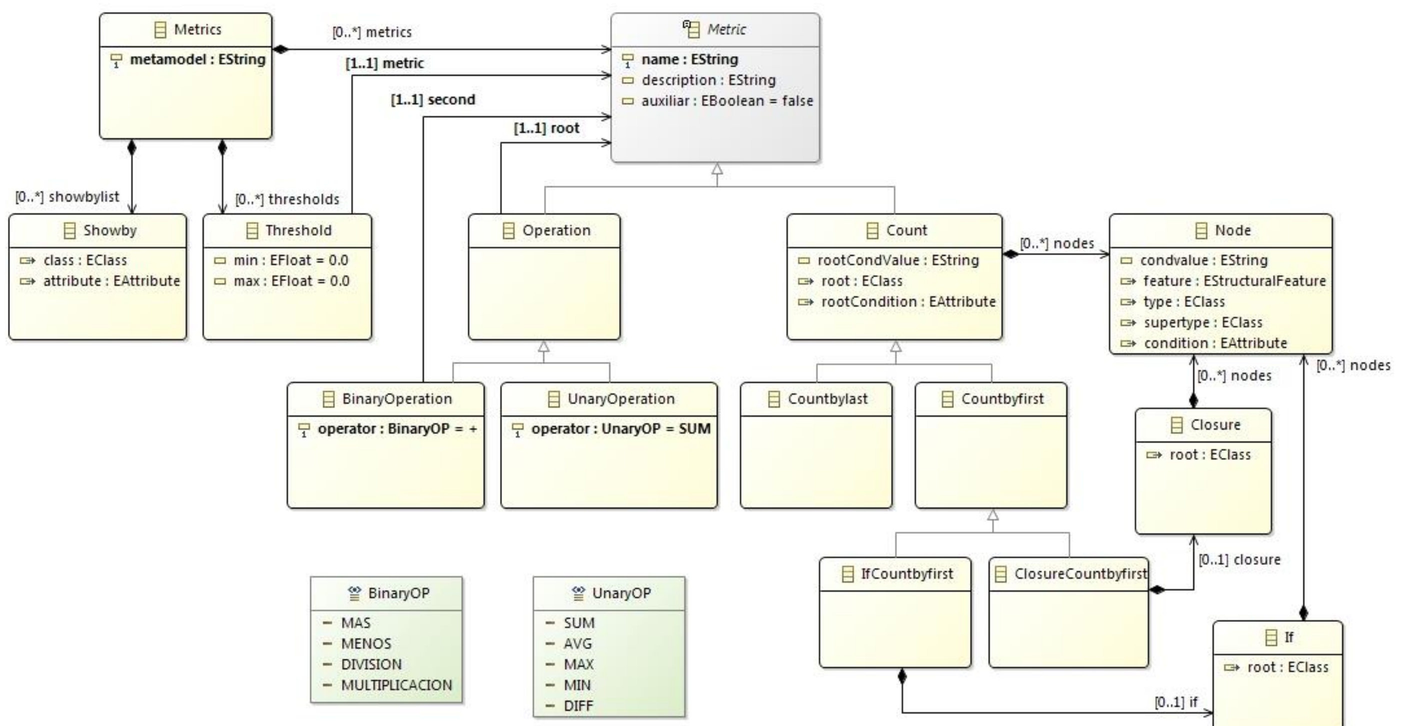

Figura 4-4. Meta-modelo del proyecto MyMtr

4.2.2 Xtext

El lenguaje definido con Xtext sigue fielmente el definido en nuestro Anexo A y por ello solo se van a mostrar algunos de los casos con la finalidad de ver la sintaxis que usa Xtext para definir cada uno de los elementos.

```
Metrics returns Metrics:
    'Metamodel' metamodel=EString
    (showbylist+=Showby)*
    metrics+=Metric
    (metrics+=Metric | thresholds+=Threshold)*
    ;

Showby returns Showby:
    'SHOW(' class=[ecore::EClass|ID] '.' attribute=[ecore::EAttribute|ID]   ')'
;

Threshold:
    'Threshold for ' metric=[Metric|ID]
    '('min=EFloat','max=EFloat ')'
;

Metric returns Metric:
    Count | Countbyfirst |Countbylast | ClosureCountbyfirst | IfCountbyfirst | Operation
    ;
```

Figura 4-5. Definición de la sintaxis en Xtext 1

En la figura 4-5 se puede ver algunos de los cambios con respecto a la manera de definir los atributos de cada objeto.
Para la asignación se usa un “=” y en los casos de tener más de un elemento del mismo tipo se usa “+=”, en los casos de que sea un tipo boolean el operador a usar es “=?”.

En la figura 4-6 se puede ver la forma en la que se declara un enumerado.

```
enum UnaryOP returns UnaryOP:
    SUM='SUM' | AVG='AVG' | MAX='MAX'| MIN='MIN'| DIFF = 'DIFF'
;

enum BinaryOP returns BinaryOP:
    MAS='+' | MENOS='-' | DIVISION='/'| MULTIPLICACION='*'
;
```

Figura 4-6. Definición de la sintaxis en Xtext 1

Por último se puede ver la definición para Count, antes comentada, en la figura 4-7.

```
Count returns Count:
    (auxiliar?= 'Auxiliary ')?
    'Metric'
    name=EString
    '='
    'COUNT'
    '('root=[ecore::EClass|ID]
    ('['rootCondition=[ecore::EAttribute|ID]'='rootCondValue=EString']')?
        (nodes+=Node)*')'
    '{'description=EString'}'
    ;
```

Figura 4-7. Definición de la sintaxis en Xtext 3

4.2.3 Xtend

Esta sección está dedicada tanto al generador de código, como a la implementación del autocompletado (muy útil en el lenguaje desarrollado) y las restricciones del lenguaje.

En el caso del generador se va a visualizar la función raíz, la cual será la encargada de llamar a generar cada uno de los ficheros según corresponda y también de hacer la copia del *ecore* de los modelos que serán analizados.
La finalidad de hacer la copia del *ecore* es dejar el proyecto generado listo para su empaquetación e instalación.

```
override void doGenerate(Resource resource, IFileSystemAccess fsa) {

    for (metrics : resource.allContents.toIterable.filter(Metrics)) {
        myplugin = metrics.getMetamodelName() + "Analyzer";
    }

    fsa.generateFile(myplugin +"/plugin.xml", compilePlugin());
    fsa.generateFile(myplugin +"/build.properties", compileBuild());
    fsa.generateFile(myplugin +"/.project", compileProject());
    fsa.generateFile(myplugin +"/.classpath", compileClasspath());
    fsa.generateFile(myplugin +"/contexts.xml", compileContexts());
    fsa.generateFile(myplugin +"/.settings/org.eclipse.jdt.core.prefs", compileSettings());
    fsa.generateFile(myplugin +"/META-INF/MANIFEST.MF", compileManifest());
    fsa.generateFile(myplugin +"/src/handler/Activator.java", compileActivator());
    fsa.generateFile(myplugin +"/src/view/MetricAnalyzer.java", compileView());

    // Llamamos al generador del main
    for (metrics : resource.allContents.toIterable.filter(Metrics)) {

        var root = ResourcesPlugin.getWorkspace().getRoot()
        var f = new File(root.rawLocationURI.path, metrics.metamodel)

        var entireEcoreText = new Scanner(f).useDelimiter("\\A").next();

        fsa.generateFile(myplugin +"/"+ metrics.getMetamodelFile, entireEcoreText);

        fsa.generateFile(myplugin +"/src/model/ModelManager.java", metrics.compileModelManager());
        fsa.generateFile(myplugin +"/src/model/Metric.java", compileMetric());
        fsa.generateFile(myplugin +"/src/model/Metrics.java", metrics.compileMetrics());
        fsa.generateFile(myplugin +"/src/model/MetricsCalculator.java", metrics.compileMetricsCalculator());
        fsa.generateFile(myplugin +"/src/model/ChildrenCalculator.java", metrics.compileChildrenCalculator());
    }
```

Figura 4-8. Generador de código

En la figura 4-9 se quiere mostrar un ejemplo sencillo de cómo se personalizan cada uno de los ficheros que se van a generar, incluido el nombre del plugin, que ira relacionado con el nombre del meta-modelo que se quiere analizar.

```
def compilePlugin(){
    '''
    <?xml version="1.0" encoding="UTF-8"?>
    <?eclipse version="3.4"?>
    <plugin>
       <extension
             point="org.eclipse.ui.views">
          <category
                id="«myplugin»"
                name="Metric Analyzer Category">
          </category>
          <view
                category="«myplugin»"
                class="view.MetricAnalyzer"
                id="view.MetricAnalyzer"
                name="«myplugin»">
          </view>
       </extension>
       <extension
             point="org.eclipse.ui.perspectiveExtensions">
          <perspectiveExtension
                targetID="org.eclipse.jdt.ui.JavaPerspective">
             <view
                   id="view.MetricAnalyzer"
                   ratio="0.5"
                   relationship="right"
                   relative="org.eclipse.ui.views.ProblemView">
             </view>
          </perspectiveExtension>
       </extension>
       <extension
             point="org.eclipse.help.contexts">
          <contexts
                file="contexts.xml">
          </contexts>
       </extension>

    </plugin>
    '''
}
```

Figura 4-9. Generador del código del plugin

A la hora de corregir o hacer válido el lenguaje desarrollado se ha hecho a través sobre todo del autocompletado o proveedor de contenido, con el cual se puede asegurar que el campo seleccionado es válido para dicha situación en el lenguaje, ya que se inspecciona el meta-modelo indicado en la primera instrucción del lenguaje y se mostrarán las opciones disponibles en cada caso.

Por ejemplo, en la figura 4-10 se puede ver como con Xtend se define un proveedor de contenido para que se escojan los diferentes tipos del meta-modelo.

```
def IScope scope_Count_root(Count c, EReference ref) {

    if(c == null) return null
    if(c.eContainer == null) return null

    var metrics = c.eContainer as Metrics
    var metamodel = ModelManager.loadMetaModel(metrics.getMetamodel)
    var classes = new ArrayList<EClass>()

    for (EPackage pck : metamodel) {
        for (EClassifier cl : pck.EClassifiers) {
            if (cl instanceof EClass)
                classes.add(cl as EClass)
        }

    }

    Scopes.scopeFor(classes)
}
```

Figura 4-10. Proveedor de contenido para la métrica Count

Esta funcionalidad se implementa en prácticamente todo el lenguaje, para obtener los tipos de objeto, para obtener el siguiente objeto en la cadena, para conseguir los atributos de los objetos, también incluso para seleccionar las métricas por su tipo, con objetivo de que las operaciones entre ellas sean correctas.

Sin embargo, la funcionalidad de validar, ha quedado más relegada a ciertos puntos más concretos, como por ejemplo que no se repitan los nombres de las métricas, o que el intervalo de los umbrales definidos sean correctos, también que los ciclos estén bien definidos (primer y último elemento se correspondan con el mismo tipo de objeto).

En la figura 4-11 se muestra la implementación del validador que procura que las métricas no repitan nombres.

```
@Check
def metricNames(Metric m) {
    var contador = 0
    var metrics = m.eContainer as Metrics
    for (metric : metrics.metrics) {
        if (m.name == metric.name) {
            contador++;
        }
    }

    if (contador > 1 || contador == 0) {
        error('Los nombres de las metricas no pueden repetirse', MetricsPackage.Literals.METRIC__NAME,
            'nombre de metrica repetido')
    }

}
```

Figura 4-11. Validador de los nombres de las métricas

5 Pruebas y resultados

Las pruebas se han realizado durante el ciclo de desarrollo, de tal manera que con cada prototipo de este, se evalúa la nueva funcionalidad y se vuelve a probar lo anterior, es decir, para cada prototipo se prueba su funcionalidad completa, de esta manera se evita que una nueva funcionalidad pueda corromper otra ya implementada y probada.

La filosofía de pruebas utilizada ha sido la de caja negra. Se basa en evaluar la salida con respecto a la entrada y decidir si el comportamiento del proyecto es el adecuado.

En este capítulo se van a mostrar algunas de las entradas y resultados del proyecto. Se va a realizar la definición de métricas sobre diagramas de clases para ilustrarlo.

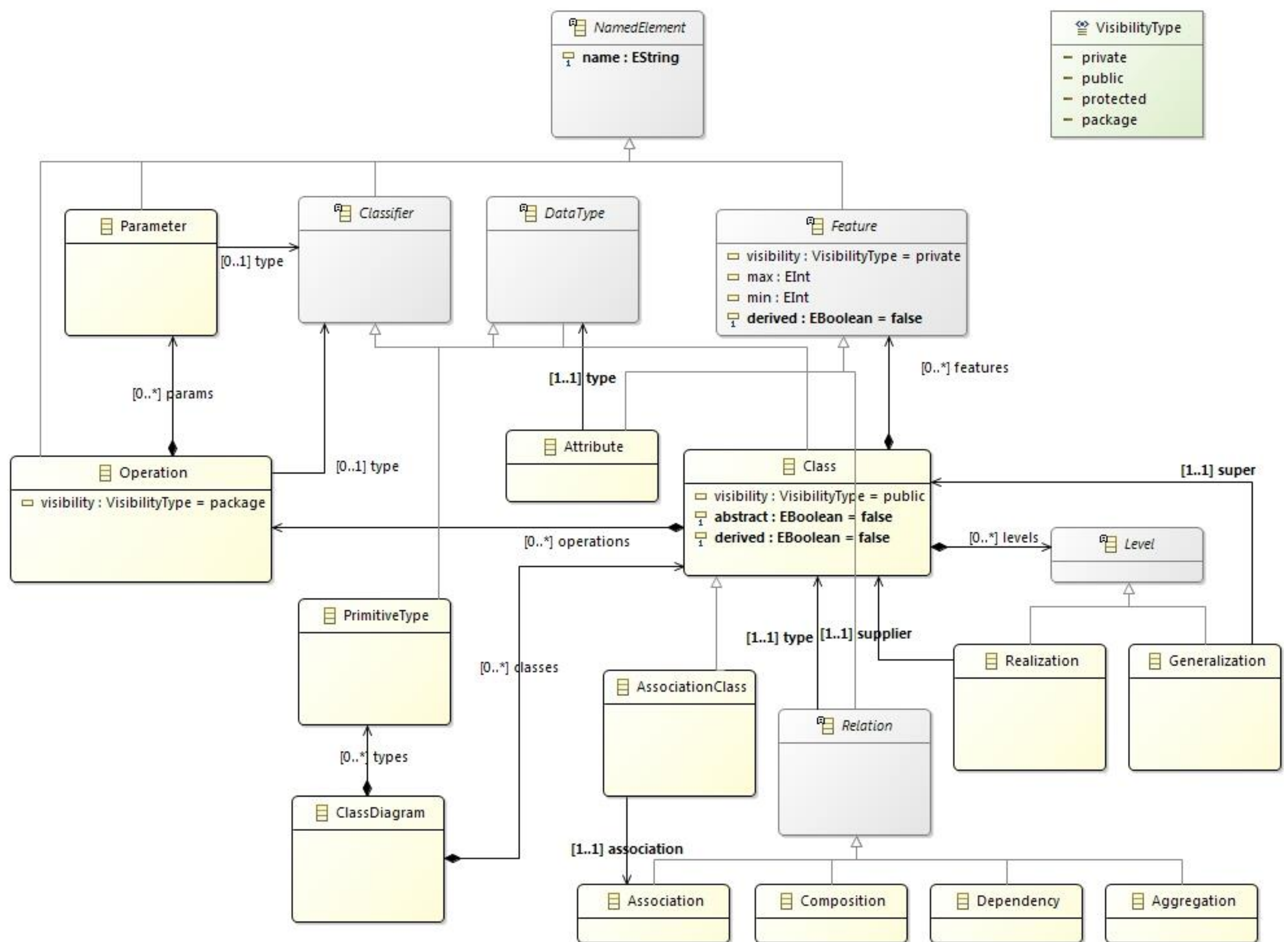

Figura 5-1. Meta-modelo para la definición de Diagramas de Clases

Un ejemplo de diagrama de clases conforme al meta-modelo anterior, que usaremos para ilustrar la herramienta desarrollada.

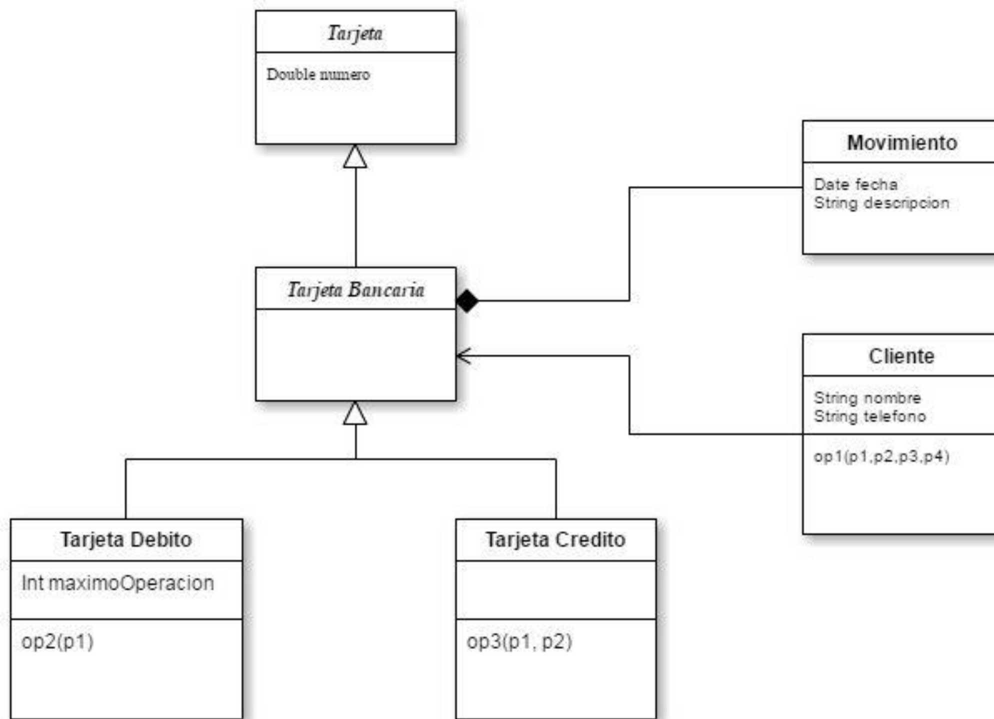

Figura 5-2. Diagrama de clases de ejemplo

Usando el lenguaje descrito en la Figura 4-3 del capítulo 4, el resultado sería el siguiente. Donde podemos ver que el valor para DSC aparece resaltado en rojo, esto se debe a que no cumple con el umbral definido para dicha métrica.

Reload Export

Metrics	Description	Value
DSC	Numero de clases	6
NAS	Numero de asociaciones de una clase	
Tarjeta		0
TarjetaBancaria		1
TarjetaCredito		0
TarjetaDebito		0
Cliente		0
Movimiento		0
APPM	Numero medio de parametros por operacion	2.3333333
DIT	Longitud desde la clase hasta raiz de arbol de herencia	
Tarjeta		0
TarjetaBancaria		1
TarjetaCredito		2
TarjetaDebito		2
Cliente		0
Movimiento		0
NOC	Numero de hijos de una clase	
Tarjeta		1
TarjetaBancaria		2
TarjetaCredito		0
TarjetaDebito		0
Cliente		0
Movimiento		0

Figura 5-3. Resultado del cálculo de métricas 1

A continuación se van a mostrar dos casos que son más complejos de especificar con el lenguaje, para ver la expresividad de este y las posibilidades que nos ofrece.
Se trata de la métrica AIF, que es el porcentaje de atributos heredados respecto a los atributos totales del modelo. En esta métrica se puede ver el uso de las métricas auxiliares y de las operaciones, tanto binarias como unarias, entre métricas.

```
Metamodel "/prueba/ClassDiagram.ecore"

Auxiliary Metric AUX1 = COUNTBYFIRST(
                    CLOSURE(Class.levels(type->Generalization).super)
                    .features(type->Attribute)
                ){""}
Auxiliary Metric sumaux1 = SUM(AUX1){"Atributos heredados"}

Auxiliary Metric AUX2 = COUNTBYFIRST(Class.features(type->Attribute)){""}
Auxiliary Metric sumaux2 = SUM(AUX2){"Atributos totales"}


Metric AIF = OP->/( sumaux1, sumaux2)
            {"Porcentaje de atributos heredados por atributos totales"}
```

Figura 5-4. Lenguaje MyMtr ejemplo 4

Y como resultado se obtiene:

Reload Export

Metrics	Description	Value
AIF	Porcentaje de atributos heredados por atributos totales	0.5

Figura 5-5. Resultado del cálculo de métricas 2

Esto se explica de manera que el atributo número de Tarjeta es heredado tres veces, por Tarjeta Bancaria, Tarjeta de Crédito y Tarjeta de Débito, es decir un total de 3 atributos heredados. Y el total de atributos de modelo es 6.

Con la otra métrica, que se trata de obtener el número de clases abstractas totales, se pretende enseñar cómo es posible condicionar cada elemento por sus atributos, ya se trate de una lista de objetos o de un objeto en particular.

La especificación del lenguaje para definir esta métrica es la siguiente.

```
Metamodel "/prueba/ClassDiagram.ecore"

Metric PRU = COUNT(Class[abstract= "true"])
           {"Numero de clases abstractas"}
```

Figura 5-6. Lenguaje MyMtr ejemplo 5

Y el resultado es:

Figura 5-7. Resultado del cálculo de métricas 3

Donde se puede ver que el cálculo resultante es dos, que se corresponde a las clases Tarjeta y Tarjeta Bancaria.

En el Anexo B, se muestran algunos ejemplos para la definición de métricas sobre diagramas de clases.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

El objetivo de este proyecto era el de proporcionar un lenguaje para la definición de métricas sobre modelos y a su vez generar una herramienta capaz de calcular dichas métricas y hacerlas visibles de una manera clara y concisa.

Se tiene como resultado MyMtr, un lenguaje de dominio especifico que permite definir innumerables métricas y para cualquier meta-modelo deseado, con las ventajas de que el proveedor de contenido hace de guía para que la definición de las métricas sea correcta, haciendo el uso del lenguaje intuitivo además de expresivo, y de la generación automática del plugin analizador de modelos, que estará preparado para analizar los modelos requeridos inmediatamente después de ser generado.

6.2 Trabajo futuro

Una posible línea a seguir en el proyecto es la de proporcionar más opciones para la definición de métricas.
Como ejemplo se pueden ampliar las operaciones unarias disponibles, se pueden añadir la funcionalidad de calcular la moda o la mediana.

Otra forma de continuar con el trabajo realizado podría ser tomando el camino de mejorar la visualización de los resultados del proyecto generado por el lenguaje, ya sea tratando de modificar la manera en la que se muestran los resultados o bien ampliando las posibilidades de exportación a diferentes formatos.

Referencias

[1] Christian Hein, Marcus Engelhardt, Tom Ritter, Michael Wagner: “Generation of Formal Model Metrics for MOF based Domain Specific Languages”. ECEASST 24 (2009)

[2] Nebras Nassar, Thorsten Arendt, Gabriele Taentzer: “Deducing Model Metrics from Meta Models”. Modellierung 2016: 29-44

[3] Jordi Cabot, Robert Clarisó, Daniel Riera: “Verification of UML/OCL Class Diagrams using Constraint Programming”. ICST Workshops 2008: 73-80

[4] Raúl Orué Rotela: “MDA Model Driven Architecture Arquitectura Dirigida por Modelos MDD Model Driven Development Desarrollo Dirigido por Modelos”, España, Universidad Católica Nuestra Señora de la Asunción. 2007.

[5] Grady Booch, James Rumbaugh, Ivar Jacobson: “The Unified Modeling Language User Guide”, Addison-Wesley, 1999.

[6] Bran Selic: “The Pragmatics of Model-Driven Development”, IEEE Software 20(5), 2003.

[7] Eclipse Modeling FrameWork (EMF) – Tutorial [Último Acceso 30/05/2017] URL: http://www.vogella.com/tutorials/EclipseEMF/article.html.

[8] Xtext [Último Acceso: 30/05/2017] URL: https://sites.google.com/site/programacionhm/conceptos/dsls/domainspecificlanguage/dsl---xtext

[9] Xtend [Último Acceso: 30/05/2017] URL: https://www.eclipse.org/xtend/index.html

[10] Jordi Cabot, Martin Gogolla: “Object Constraint Language (OCL): A Definitive Guide”. SFM 2012: 58-90.

[11] Build metamodels with dynamic EMF [Ultimo Acceso: 30/05/2017] URL: https://www.ibm.com/developerworks/library/os-eclipse-dynamicemf.

[12] Beatriz Marín, Nelly Condori-Fernández, Oscar Pastor: “Calidad en modelos conceptuales: un análisis multidimensional de modelos cuantitativos basados en la ISO 9126”, RPM-AEMES, VOL. 4, Nº Especial, Octubre 2007.

[13] N.E. Fenton, S.L. Pfleeger: “Software Metrics: A Rigorous and Practical Approach", second ed., PWS, 1998.

[14] Marco Brambilla, Jordi Cabot, and Manuel Wimmer: “Model-Driven Software Engineering in Practice”, Morgan & Claypool Publishers, 2012

[15] Steven Kelly, Juha-Pekka Tolvanen: “Domain-Specific Modeling - Enabling Full Code Generation”, Wiley. 2008.

[16] Juan Manuel Cueva Lovelle, B. Cristina Pelayo García-Bustelo: “MDE: Ingeniería dirigida por modelos. Otra forma de construir software”, 2008.

[17] MDD [Ultimo Acceso: 30/05/2017] URL: http://tecsoftwa.blogspot.com.es/2012/09/que-es-mdd.html

[18] Gonzalo Génova: “Conceptos básicos de modelado”, Universidad Carlos III de Madrid.

[19] Rafael Martín Jiménez: “Metodología de generación automática de aplicaciones colaborativas”, España, Universidad Autónoma de Madrid: 142, 2011.

Glosario

API	Application Programming Interface
DSL	Domain Specific Language
GPL	General Purpose Modeling Languages
EMF	Eclipse Modeling Language
MDD	Model Driven Development
MDE	Model Driven Engineering
EBNF	Extended Backus-Naur Form
OCL	Object Constraint Language

Anexos

A Sintaxis Concreta MyMtr

En este anexo se muestra la sintaxis completa del lenguaje MyMtr:

```
<Metrics>               ::= 'Metamodel' <STRING>
                            (<Showby>)*
                            <Metric>
                            (<Metric> | <Threshold>)*;

<Threshold>             ::= 'Threshold for ' (<Metric>|<ID>)
                            '('<FLOAT>','<FLOAT>')';

<Showby>                ::= 'SHOW(' <Class> '.' <Attribute> ')';


<Metric>                ::= <Count >
                         |  <Countbyfirst>
                         |  <Countbylast>
                         |  <ClosureCountbyfirst>
                         |  <IfCountbyfirst>
                         |  <Operation>;

<Operation>             ::= <UnaryOperation>
                         |  <BinaryOperation>;

<BinaryOperation>       ::= ('Auxiliary ')? 'Metric' <STRING> '='
                            'OP->'<BinaryOP>
                            '('(<Metric>|<ID>)','(<Metric>|<ID>])')'
                            '{'<Description>'}';

<UnaryOperation>        ::= ('Auxiliary ')? 'Metric' <STRING> '='
                            <UnaryOP> '('(<Metric>|<ID>)')'
                            '{'<Description>'}';

<Count>                 ::= ('Auxiliary ')? 'Metric' <STRING> '='
                            'COUNT('<Class>
                            ('['<Attribute>'='<STRING>']')?
                            (<Node>)*')'
                            '{'<Description>'}';

<Countbyfirst>          ::= ('Auxiliary ')? 'Metric' <STRING> '='
                            'COUNTBYFIRST('<Class>
                            ('['<Attribute>'='<STRING>']')?
                            (<Node>)*')'
                            '{'<Description>'}';
```

```
<Countbylast>              ::= ('Auxiliary ')? 'Metric' <STRING> '='
                                'COUNTBYLAST('<Class>
                                ('['<Attribute>'='<STRING>']')?
                                (<Node>)*')'
                               '{'<Description>'}';

<ClosureCountbyfirst>      ::= ('Auxiliary ')? 'Metric' <STRING> '='
                                'COUNTBYFIRST('<Closure>
                                ('['<Attribute>'='<STRING>']')?
                                (<Node>)*')'
                                '{'<Description>'}';


<IfCountbyfirst>           ::= ('Auxiliary ')? 'Metric' <STRING> '='
                                'COUNTBYFIRST('<If>
                                 ('['<Attribute>'='<STRING>']')?
                                 (<Node>)*')'
                                '{'<Description>'}';

<Node>                     ::= '.'<StructuralFeature>
                                 ('(supertype->'<Class>')' |  '(type->'<Class>')')?
                                 ('['<Attribute>'='<STRING>']')?;

<Closure>                  ::= 'CLOSURE('<Class> (<Node>)*')';

<If>                       ::= 'IF(' <Class>(<Node>)*')';


<UnaryOP>                  ::= 'SUM'
                                |  'AVG'
                                |  'MAX'
                                |  'MIN'
                                |  'DIFF';

<BinaryOP>                 ::= '+'
                                |  '-'
                                |  '/'
                                |  '*';

<Description>              ::=  <STRING>;
<Class>                    ::=  <STRING>;
<Attribute>                ::= <STRING>;
<StructuralFeature>        ::= <STRING>;
<ID>                       ::= <STRING>;
```

B Métricas definidas con MyMtr para diagramas de clases

A continuación se van a mostrar algunos ejemplos de las métricas que se podrían definir para modelos de diagramas de clases.

La fuente utilizada para realizar este anexo es *Calidad en modelos conceptuales: un análisis multidimensional de modelos cuantitativos basados en la ISO 9126* de Beatriz Marín, Nelly Condori-Fernández y Oscar Pastor (2007) [12].

Se presupone que la línea necesaria para cargar el meta-modelo va antes de la definición de las métricas, *Metamodel "Classdiagram.ecore":*

.

- DSC, Número total de clases:
  Metric DSC = COUNT(Class){" Número total de clases"}

- NAS, Número de asociaciones de una clase:
  Metric NAS = COUNTBYFIRST(Class.features(type->Association)){" Número de asociaciones de una clase"}

- APPM, Número medio de parámetros por operación:
  Auxiliary Metric AUXAPPM = COUNTBYFIRST(Operation.params){""}
  Metric APPM = AVG(AUXAPPM){"Número de parámetros por operación"}

- DIT, Longitud desde la clase hasta la raíz de árbol de herencia:
  Metric DIT = COUNTBYFIRST(CLOSURE(Class.levels (type->Generalization).super)){"Longitud desde la clase hasta la raíz del árbol de herencia"}

- NOC, Número de hijos de una clase:
  Metric NOC = COUNTBYLAST(Generalization.super){"Número de hijos de una clase"}

- MIF, Métodos heredados por métodos totales:
  Auxiliary Metric AUX1 = COUNTBYFIRST(CLOSURE(Class.levels (type->Generalization).super).operations){""}
  Auxiliary Metric sumaux1 = SUM(AUX1){""}

  Auxiliary Metric AUX2 = COUNT(Operation){""}
  Auxilary Metric AUX3 = OP->+(sumaux1, AUX2){""}

  Metric MIF =OP->/(sumaux1, AUX3){"Métodos heredados por métodos totales"}

- AIF, Atributos heredados por atributos totales:
  Auxiliary Metric AUX1 = COUNTBYFIRST(CLOSURE(Class.levels (type->Generalization).super).features(type->Attribute)){""}
  Auxiliary Metric sumaux1 = SUM(AUX1){""}

Auxiliary Metric AUX2 = COUNT(Attribute){""}
Auxilary Metric AUX3 = OP->+(sumaux1, AUX2){""}

Metric MIF = OP->/(sumaux1, AUX3){"Atributos heredados por atributos totales"}

- PIM, Número total de métodos públicos de una clase:
  Auxiliary Metric AUX1 = COUNTBYFIRST(Class.operations [visibility = "public"]){""}
  Auxiliary Metric AUX2 = COUNTBYFIRST(CLOSURE(Class.levels (type->Generalization).super).operations[visibility = "public"]){""}

  Metric PIM = OP->+(AUX1, AUX2){"Número total de métodos públicos de una clase"}

- NHO, Número de métodos heredados de una clase:
  Metric NHO = COUNTBYFIRST(CLOSURE(Class.levels (type->Generalization).super).operations){"Número de métodos heredados de una clase"}

- NOM, Número de métodos definidos en una clase:
  Metric NOM = COUNTBYFIRST(Class.operations){"Número de métodos definidos en una clase"}

- NASSOC, Número total de asociaciones:
  Metric NASSOC = COUNT(Association){"Número total de asociaciones"}

- NAGG, Número total de relaciones de agregación:
  Metric NAGG = COUNT(Aggregation){"Número total de relaciones de agregacion"}

- NGEN, Número total de relaciones de generalizacion
  Metric NGEN = COUNT(Generalization){"Número de relaciones de generalizacion"}

- MAXDIT, Máxima longitud del árbol de herencia:
  Metric DIT = COUNTBYFIRST(CLOSURE(Class.levels (type->Generalization).super)){"Longitud desde la clase hasta la raíz del árbol de herencia"}

  Metric MAXDIT = MAX(DIT){"Máxima longitud desde una clase a la raíz del árbol de herencia"}

- ANA, Número medio de ancestros:
  Auxiliary Metric AUX = COUNTBYFIRST(CLOSURE(Class.levels(type->Generalization).super)){""}
  Metric ANA = AVG(AUX){"Número medio de ancestros"}

- DAM, Atributos privados por atributos totales de una clase:
  Auxiliary Metric AUX = COUNTBYFIRST(Class.features (type->Attribute)[visibility = "private"]){“”}
  Auxiliary Metric AUX1 = COUNTBYFIRST((CLOSURE(Class.levels (type->Generalization).super).features(type->Attribute) [visibility = "private"]){“”}

  Auxliary Metric SUMAUX = OP->+(AUX, AUX1){“”}
  Auxiliary Metric AUX2 = COUNTBYFIRST(Class.features(type->Attribute)){“”}

  Metric DAM = OP->/ (SUMAUX, AUX1){“Atributos privados por atributos totales de una clase”}

- NMA, Número total de métodos que se definen en una subclase:
  Metric NMA = COUNTBYFIRST((IF(Class.levels (type->Generalization).super).operations){“Número total de métodos que se definen en una subclase”}

- DCC, Número de clases diferentes con las que está relacionada una clase:
  Auxiliary Metric AUX = COUNTBYFIRST(Class.features (type->Relation).type){“”}
  Auxiliary Metric AUX1 = COUNTBYFIRST(Class.operations.params.type (type->Class)){“”}

  Auxiliary Metric SUMAUX = OP->+(AUX, AUX1){“”}

  Metric DCC = DIFF(SUMAUX){“Número de clases diferentes con las que se relaciona una clase”}

- CIS, Número de métodos públicos de una clase:
  Metric CIS = COUNTBYFIRST(Class.operations[visibility = "public"]){“Número de métodos públicos de una clase”}

- NODP, Número de partes directas que contiene una clase que pertenece a una jerarquía de agregación:
  Metric NODP = COUNTBYFIRST(Class.features(type->Aggregation)){“Número de partes directas que contiene una clase que pertenece a una jerarquía de agregación”}

- NP, Número de partes de una clase todo:
  Metric NP = COUNTBYFIRST(CLOSURE(Class.features (type->Composition).type)){“Número de partes de una clase todo”}

- NW, Número de clases todo de una clase parte:
  Metric NW = COUNTBYLAST(Class.features (type->Composition).type){“Número de clases todo de una clase parte”}