UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Un lenguaje de dominio específico para el análisis estático de programas Java

Sara Pérez Soler
Tutor: Juan de Lara Jaramillo

Enero 2017

Un lenguaje de dominio específico para el análisis estático de programas Java

AUTOR: Sara Pérez Soler
TUTOR: Juan de Lara Jaramillo

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Enero de 2017

Resumen

En la actualidad, el aumento del tamaño y la complejidad de los programas junto con la necesidad de seguridad y durabilidad han hecho crecer el uso de herramientas que automatizan el análisis de los programas. Concretamente, las herramientas de análisis estático, que permiten realizar un análisis sobre el código fuente sin necesidad de ejecutarlo.

Estas herramientas, normalmente, centran sus esfuerzos en encontrar errores de programación, dejando por explorar un uso más amplio del análisis estático como puede ser la consulta de características en el código fuente. Dichas características pueden ser usadas para la búsqueda de patrones de programación, hacer cumplir normas de estilo en el código o, simplemente, ser requerimientos que se exigen en un proyecto.

Para ello, se ha desarrollado JavaCheck, un lenguaje de dominio específico que permite generar reglas para definir propiedades esperables en código Java, como por ejemplo los modificadores de un atributo o el tipo de retorno de un método, que posteriormente serán comprobadas en proyectos. Se proporciona para ello un entorno de modelado de reglas integrado en Eclipse, por ser uno de los entornos de desarrollo más utilizado para Java, junto con una sintaxis concreta textual centrada en facilitar su creación. Para la comprobación de las reglas en proyectos Java, se ha creado un generador de código que utiliza una librería con funcionalidad para recorrer el árbol sintáctico de los programas. El desarrollo de JavaCheck se ha realizado con *Eclipse Modeling Framework*.

Para validar JavaCheck se han realizado tres tipos de pruebas en función de que aspecto del lenguaje que se quiere validar: pruebas de expresividad, pruebas de utilidad y pruebas de escalabilidad. Terminado este proyecto, el principal trabajo futuro que se puede desarrollar es la ampliación del lenguaje, debido al amplio abanico de características que puede tener un código Java.

Palabras clave

Java, análisis estático, modelado, lenguaje de dominio especifico, generación de código.

Abstract

Nowadays the increase in the program's size and complexity and the need for more security and durability have increased the use of tools to automate the analysis of programs, specifically static analysis tools, which perform analysis on the source code without having to run it.

These tools usually focus on finding programming errors, neglecting the definition and query of more general properties in the source code. These properties can be used to search programming patterns, enforce coding standards, or simply be requirements that are demanded in a project.

To do this, JavaCheck has been developed, a domain specific language that allows generating rules to define properties expected in Java code which will be verified in projects. For this purpose, a rule modeling environment integrated in Eclipse is provided, as it is one of the most used development environments for Java, along with a specific syntax focused on facilitating its creation. Finally, using a library and a code generator designed to analyze projects according to the rules. JavaCheck has been developed in the Eclipse Modeling Framework.

JavaCheck has been validated in three perspectives, depending on the aspect of the language to be validated: expressiveness, utility and scalability. Finished this project, the main future work that can be developed is the extension of the language, due to the large number of properties that can have Java code.

Keywords

Java, static analysis, modeling, Domain Specific Languages, code generation.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

1.1 Motivación

Uno de los problemas que surge actualmente en la ingeniería del software es la necesidad de programas de mayor tamaño, complejidad, seguridad y durabilidad. Debido al aumento de tamaño de los programas, surgen herramientas de análisis de código, para encontrar la mayor cantidad de errores de la forma más automatizada posible, ahorrando el máximo coste de tiempo y recursos.

Otra necesidad en el software actual es la limpieza de código, pero ¿qué es código limpio? Como dice Grady Booch, autor del libro *Object Oriented Analysis and Desing with Applications* y co-desarrollador del Lenguaje Unificado de Modelado (UML) [1]:

> *El código limpio es simple y directo. El código limpio se lee como un texto bien escrito. El código limpio no oculta la intención del diseñador, sino que muestra nítidas abstracciones y líneas de control.*

Y Dave Thomas fundador de *Object Technology International* (OTI), responsable del desarrollo inicial de *Eclipse Open Source IDE*[1]:

> *El código limpio se debe poder leer y mejorar por parte de un programador que no sea su autor original […].*

También Brian Kernighan, autor del primer libro de programación en C, dijo en su libro *The elements of programming style* [32]:

> *Todo el mundo sabe que depurar el código es el doble de duro que escribir un programa desde el primer momento. Así que, si ya eres todo lo inteligente que puedes cuando lo escribes, ¿cómo lo vas a depurar?*

La necesidad de código limpio surge de varios hechos que se cumplen en casi todo software:

- El 80% del coste de la vida de un software es su mantenimiento y, rara vez, se encarga de ello el mismo programador que lo creó.
- Si partimos de un código que no es limpio, el mantenimiento generará más problemas en el código hasta que llegue un momento que sea imposible de mantener.
- Además, cuando programamos dedicamos una alta proporción del tiempo a leer el código ya creado. Cuanto más claro sea, menos esfuerzo tendremos que dedicar en este punto.

Las convenciones de código o reglas de estilo son un elemento necesario para tener un código limpio y fácil de mantener. Pero, para que las reglas de estilo sean útiles, todos los programadores de un mismo proyecto deben seguir las mismas. Hay algunos estándares de estas convenciones como el de *Sun Code Conventions* [3], *The Elements of Java Style* [33]**¡Error! No se encuentra el origen de la referencia.** o *DougLea's Coding Standards* [4], no obstante, no todas las empresas siguen los mismos estándares, algunas, de hecho, siguen sus propia reglas. No se pretende abrir un debate sobre qué convención es mejor, porque la realidad es que el mero hecho de seguir unas reglas de estilo en la programación ya facilita la comprensión del código. Por ejemplo, el ver que

los mismos elementos siempre están escritos de la misma manera ya ayuda a identificarlos. Lo que sí se pretende resaltar es la importancia de seguir unas reglas de estilo y que todos los programadores de un proyecto sigan las mismas.

Por último, otro campo que motiva este Trabajo de Fin de Grado es la docencia. En la actualidad, en la carrera de ingeniería informática, la cantidad de prácticas que los profesores deben corregir es muy considerable. Automatizar todo lo posible ese trabajo haciendo una primera criba de normas que se deban esperar de los proyectos a corregir en relación al dominio ahorraría mucho tiempo en esta tarea.

1.2 Objetivos

El objetivo de este Trabajo de Fin de Grado es diseñar y desarrollar un lenguaje de dominio específico que sirva para definir reglas esperables en códigos Java y que se encargue del análisis estático, es decir, que no sea necesario ejecutar el código, de diferentes proyectos.

No se busca desarrollar un analizador que se encargue de buscar errores o determinar el rendimiento de tiempo, de recursos o seguridad de un programa. El fin es desarrollar un lenguaje que se encargue de buscar propiedades en el código. Como, por ejemplo, patrones de diseño, reglas de estilo o elementos dentro de un dominio.

Las aplicaciones que se le pueden dar a este lenguaje son, como se expresa en el apartado anterior, el control de calidad del código, la especificación de guías de estilo o la corrección semiautomatizada de programas. Para ello, es necesario definir una sintaxis específica y concisa con la que se puedan expresar estas propiedades. A su vez, debe ser fácilmente ampliable, debido a que estas propiedades crecen con la evolución del lenguaje de programación. También se debe proporcionar un mecanismo para la evaluación de dichas reglas sobre programas Java.

1.3 Organización de la memoria

En este apartado daremos una visión global sobre cómo está estructurado este documento.

- Capítulo 2: comenzaremos introduciendo una visión general sobre el estado actual de los analizadores estáticos.
- Capítulo 3: se explicarán las tecnologías usadas para desarrollar el lenguaje.
- Capítulo 4: se expondrá el diseñado el lenguaje.
- Capítulo 5: implementación.
- Capítulo 6: una vez que está implementado, pruebas de expresividad, utilidad y escalabilidad.
- Capítulo 7: conclusiones y trabajos futuros.

2 Estado del arte

2.1 Introducción

Debido al creciente tamaño de los proyectos, el análisis del código ha ido evolucionando y generando herramientas que ayudan a realizar esta tarea en diferentes etapas. Para esto, se pueden distinguir dos tipos de herramientas: las estáticas y las dinámicas. Las primeras permiten realizar un análisis sin ejecutar el código y se pueden utilizar desde las primeras fases de codificación. Las segundas, conocidas también como herramientas de *testing*, se utilizan en las últimas fases de codificación. El conjunto de herramientas trata de prevenir errores y de facilitar la mantenibilidad con vistas a generar un producto de mayor calidad.

2.2 Herramientas para el análisis estático

En este apartado se comentarán algunas de las características principales de algunos analizadores estáticos que funcionen sobre código Java, la mayoría de los cuales centran sus esfuerzos en la búsqueda de fallos de programación, código muerto o análisis del flujo de datos.

2.2.1 PMD

PMD [6] es un analizador estático de código libre que pertenece al conjunto de proyectos de *SourceForge.net*, que se encuentran bajo la licencia BSD (Berkeley Software Distribution) [9]. En su página encontramos una gran cantidad de documentación del software que trata desde cómo instalarlo a cómo ampliarlo. PMD escanea el código fuente de Java y otros lenguajes y busca problemas potenciales como:

- Posibles errores. Sentencias *try / catch /finally / switch* vacías
- Código muerto. Variables locales, parámetros y métodos privados no utilizados.
- Código no óptimo. Uso ineficiente de *StringBuffer*.
- Expresiones excesivamente complicadas. Sentencias *if* innecesarias, bucles *for* que podrían ser bucles *while*.
- Código duplicado. Copiar y pegar el código significa copiar y pegar los errores. También implica que, si se quiere cambiar algo del código, hay que hacerlo en varios sitios.

PMD tiene las reglas divididas en conjuntos de reglas (por ejemplo, conjunto de reglas para Android o conjunto de reglas para encontrar código muerto) y puedes seleccionar qué conjunto de reglas incluir para analizar el proyecto. Permite añadir nuevas reglas codificándolas en Java o con XPath, lenguaje que permite recorrer un árbol DOM, como el de un XML o un árbol de sintaxis abstracta (explicado en la sección 3.3), siendo este último el que utilizan para el análisis.

Es compatible con Java, JavaScript, PLSQL, Apache Velocity, XML, XSL. Adicionalmente se incluye CPD (Copy Paste Detector) que busca código duplicado en Java, C, C ++, C #, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Ruby, Scala, Objective C, Matlab, Python, Go, Swift.

2.2.2 FindBugs

FindBugs [9] es una herramienta de software libre, desarrollada inicialmente en la Universidad de Maryland, que pertenece al grupo de proyectos de *SourceForge.net*. Está bajo la licencia GPL (GNU Public License) [11]. También dispone de un manual muy completo que va desde la instalación, hasta cómo usarlo. En su página hay un listado de los errores que es capaz de encontrar en el código clasificados en las siguientes categorías:

- Sin corrección.
- Mala práctica.
- Rendimiento (*performance*).
- Internacionalización.
- Seguridad.
- Vulnerabilidad para código malicioso.
- Código dodgy. Código confuso, anómalo o escrito de forma que pueda llevar a errores.

FindBugs se centra sobre todo en buscar errores de codificación y no permite definir nuevas reglas. Opera en Java byteCode en lugar de en código fuente. Da soporte a lenguaje Java.

2.2.3 Simian

Simian [12] es un analizador estático de RedHill Consultoring [13]. Simian busca código duplicado en proyectos realizados con los lenguajes: Java, C#, C, C++, COBOL, Ruby, JSP, ASP, HTML, XML y Visual Basic. Los derechos de Simian son exclusivos de Simon Harris, no obstante tiene una licencia libre para uso no comercial, una licencia personal y otras licencias comerciales.

2.2.4 CheckStyle

CheckStyle [14] es un proyecto de *SourceForge.net,* con licencia GPL [11]. Es un analizador estático que centra sus esfuerzos en el análisis de reglas de estilo. En su página se encuentra una documentación muy completa sobre su instalación y uso. También trata sobre cómo ampliar sus funciones. Permite analizar las reglas de estilo según alguno de los estándares. La herramienta es altamente configurable, puede soportar casi cualquier estándar de codificación y permite seleccionar con qué convenciones se quiere analizar y con cuáles no. Las reglas se configuran en un fichero XML, también se pueden crear nuevas reglas codificándolas en Java usando un árbol de sintaxis abstracta (explicado en la sección 3.3).

Las reglas de validación y los parámetros que deben comprobarse se indican en un archivo de configuración XML y pueden clasificarse en:

- Comentarios Javadoc.
- Convenciones en nombres y cabecera de los archivos.
- Sentencias import.
- Tamaño del código.

- Espacios en blanco y bloques.
- Diseño de clases.
- Código duplicado.
- Chequeo de métricas.

2.2.5 SonarQube

SonarQube [15] (antes llamado solamente Sonar) es una plataforma de software libre (bajo la licencia GPL [11]). Usa diversas herramientas de análisis estático de código fuente como Checkstyle, PMD o FindBugs para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa. SonarQube analiza los siete ejes principales de calidad del código. Una vez analizados, nos muestra información detallada sobre la arquitectura y el diseño, comentarios de nuestro programa, código duplicado, reglas de programación acordes con el lenguaje que estemos utilizando, bugs potenciales y su posible solución, datos referentes a la complejidad del proyecto e, incluso, datos sobre pruebas unitarias (si tenemos alguna), como número de pruebas unitarias pasadas correctamente o porcentaje de cubrimiento de las mismas.

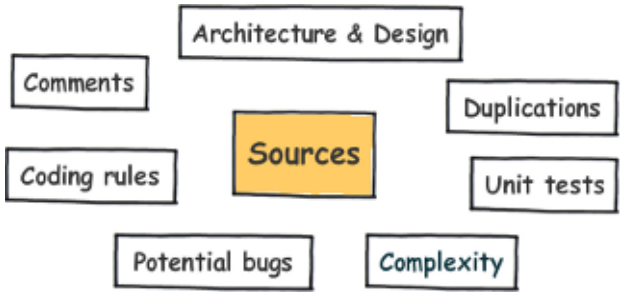

Figura 2.1. Siete ejes sobre los que analiza SonarQube[15]

Tiene soporte para más de 20 lenguajes de programación entre los que se encuentran Java, C#, C / C++, PL / SQL, Cobol, ABAP, Python, JavaScript…

2.3 Lenguajes de consulta.

Dentro de los analizadores estáticos también podemos encontrar algunos lenguajes de consulta. Para ello, primero es necesario procesar el código hasta obtener una representación relacional o bases de datos, para después poder buscar características del código mediante consultas sobre estas representaciones.

Los lenguajes de consulta sobre código de los que se habla a continuación se basan en Datalog [19], que es un lenguaje de consulta de bases de datos deductivas.

2.3.1 SemmleQL

SemmleQL[17] es un lenguaje de consulta orientada a objetos que se ejecuta sobre una representación relacional de programas Java, perteneciente a la empresa Semmle [16]. Es una

variante de Datalog con sintaxis modelada en Java y similar a la sintaxis de otros lenguajes de consulta como SQL. La recursividad y la sintaxis orientada a objetos le permiten hacer consultas muy sofisticadas entre las que destacan:

- Encontrar todas las instancias que vulneren la seguridad.
- Comprobar el uso correcto de una API.
- Buscar el uso de una biblioteca específica - dónde se utiliza y por quién.
- Realizar cualquier otra búsqueda o análisis que se pueda imaginar.

Este es un ejemplo de consulta en QL. Selecciona todas las clases que contienen más de 10 métodos públicos.

```
from Class c, int numofm
where numofm = count(Method m| m.getDeclaringType()=c
                    and m.hasModifier("public"))
      and numofm > 10
select c.getPackage(), c, numofm
```

Figura 2.2. Ejemplo de consulta en QL

La alta capacidad de sus consultas permite realizar análisis sobre reglas de estilo.

2.4 Conclusiones

Tras la inmersión en las diversas herramientas de análisis estático de código que actualmente se encuentran disponibles, encontramos que, en su mayoría, centran sus esfuerzos en búsqueda de errores de codificación o reglas de estilo según estándares. Algunas como CheckStyle o PMD usan un árbol de sintaxis abstracta (explicado en la sección 3.3) para este fin, y aunque en varias de ellas se pueden definir nuevas reglas, para ello es necesario casi siempre añadir nueva codificación.

Tanto para la implementación de nuevas reglas, como para búsqueda de reglas más allá del análisis en busca de errores, son más útiles los lenguajes de consulta, que te permiten, usando ciertas características del código, expresar tus propias reglas. Sin embargo sus sintaxis suelen ser muy amplias, lo que genera una mayor complejidad a la hora de utilizarlas que un DSL. También tienen el inconveniente de que, antes de poder usarlos, es necesario transformar los códigos fuente en una representación relacional y cada vez que se realice un cambio en el código es necesario volver a realizar la transformación, aumentando la carga de trabajo en comparación con el árbol de sintaxis abstracta.

3 Conceptos previos

En este capítulo se estudiarán las distintas tecnologías, conceptos y herramientas que han jugado un papel importante en el desarrollo del lenguaje y que serán mencionadas frecuentemente a lo largo del documento.

En concreto, se profundizará en los siguientes puntos:

- **Desarrollo Dirigido por Modelos**: (MDD, de sus siglas en inglés), con especial énfasis en los Lenguajes de Dominio Específico (DSLs, de sus siglas en inglés) y la generación de código.
- **Eclipse Modeling Framework** (EMF): Ecore, Xtext y Xtend.
- **Abstract Syntax Tree** (AST): en especial el AST proporcionado por la API de Eclipse *Java Development Tools* (JDT).

3.1 Desarrollo dirigido por modelos

El paradigma de desarrollo que se ha seguido se corresponde con el denominado Desarrollo Dirigido por Modelos – *Model Driven Development* (MDD), que utiliza los modelos como principal elemento en el desarrollo.

Igual que la aparición de los lenguajes de programación supuso un mayor nivel de abstracción, puesto que permiten expresarse con instrucciones de más alto nivel que equivalen a varias instrucciones máquina, el desarrollo dirigido por modelos tiene como objetivo elevar el nivel de abstracción con respecto a los lenguajes de programación, lo que permite automatizar más el desarrollo. A partir de modelos de alto nivel, que pueden ser expresados con una sintaxis gráfica o textual, se podría genera código para la aplicación final.

A continuación, veremos los aspectos de MDD más relevantes en esta discusión.

La fuente utilizada para esta sección es el libro *Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas* de J. G Molina y otros. Ed Ra-Ma. (2013) [23]

3.1.1 Modelo

Los ingenieros siempre han usado modelos antes de construir sus obras y artefactos para: especificar y validar el sistema (detectar errores y omisiones del sistema, prototipado) y guiar en la implementación del mismo.

Un modelo es una representación simplificada de la realidad que contiene la información necesaria para llevar a cabo una tarea. Por lo tanto, un modelo tiene que ser:

- Abstracto: simplifica la realidad suprimiendo los detalles irrelevantes y centrándose en los aspectos esenciales, de modo que la esencia del sistema sea mejor conocida.
- Comprensible: expresa de una forma que se pueda entender fácilmente por los usuarios.
- Preciso: representa fielmente el sistema modelado.
- Predictivo: se puede utilizar para sacar conclusiones correctas sobre el sistema.

Una de las ventajas del MDD en ingeniería informática es que el modelado es la implementación, cerrando así la brecha que existe entre la fase de requisitos, análisis y de implementación.

3.1.2 Lenguaje de Modelado

Un lenguaje de modelado es una herramienta para describir la realidad de forma explícita con cierto nivel de abstracción. Los lenguajes de modelados pueden ser gráficos, que representan la realidad con símbolos gráficos en forma de diagrama (diagramas UML) o con estructura de árbol; o textuales. Los lenguajes de modelado se definen mediante la semántica (significado de cada símbolo), la sintaxis abstracta (estructura lógica del modelo) y la sintaxis concreta (conjunto de símbolos gráficos utilizados).

Una de las principales clasificaciones de los lenguajes de modelado se da en función de su dominio:

- Lenguajes de Modelado de Dominio Especifico – *Domain Specific Modeling Language* (DSL) es un lenguaje centrado en el dominio o contexto de un problema, con el objetivo de facilitar la tarea de las personas que necesitan describir modelos en ese dominio. Un ejemplo de DSL podría ser *Verilog* [34] un lenguaje de descripción hardware.

- Lenguajes de Modelado de Propósito General – *General Purpose Modeling Languages* (GPL) comprende nociones de modelado que pueden ser aplicadas a cualquier sector o dominio. Un ejemplo de GPL es Lenguaje Unificado de Modelado (UML) [35]

3.1.3 Meta-modelo

Dado que MDD intenta elevar el nivel de abstracción en el desarrollo con el uso de modelos, un paso natural es representar los modelos como "instancias" de otros modelos más abstractos. Así pues, surge el meta-modelo, un modelo que describe un conjunto de modelos. Es decir, igual que un modelo es una abstracción de la realidad, podemos definir el meta-modelo como una abstracción superior que destaca las propiedades del modelo en sí.

De este modo, el diseño de un meta-modelo se puede abordar del mismo modo que el diseño de un modelo de cualquier dominio, solo que ahora se modela un lenguaje y se trata de representar los conceptos y las relaciones que hay entre ellos.

Así pues, un meta-modelo representa la sintaxis abstracta o estructura lógica de un lenguaje de modelado.

Si utilizamos los conceptos orientados a Objetos tenemos:

- Un concepto del lenguaje se modela como una **clase**.

- Propiedades del concepto como **atributos** de la clase.

- Relaciones entre conceptos como referencias (**asociaciones**) o **composiciones**.

- Especificaciones de un concepto como generalizaciones (**herencia**) entre clases.

- Definición del lenguaje encapsulada en un **paquete**.

A su vez, dado que un meta-modelo es también un modelo, se debe utilizar un lenguaje de meta-modelado necesitando una sintaxis concreta y abstracta para definirlo, es decir un meta-metamodelo. Para explicarlo usamos la **arquitectura de los cuatro niveles** [21][22][23].

- **M0. Nivel de datos del usuario o del mundo real:** datos del mundo real que son manipulados por el software.
- **M1. Nivel de modelado:** los modelos que representan los datos del nivel M0.
- **M2. Nivel de meta-modelado:** meta-modelos que representan los modelos del nivel M1.
- **M3. Nivel de meta-metamodelado:** meta-metamodelos que representan los meta-modelos del nivel M2. El meta-metamodelo se define con los mismos conceptos con los que son definidos (definición circular).

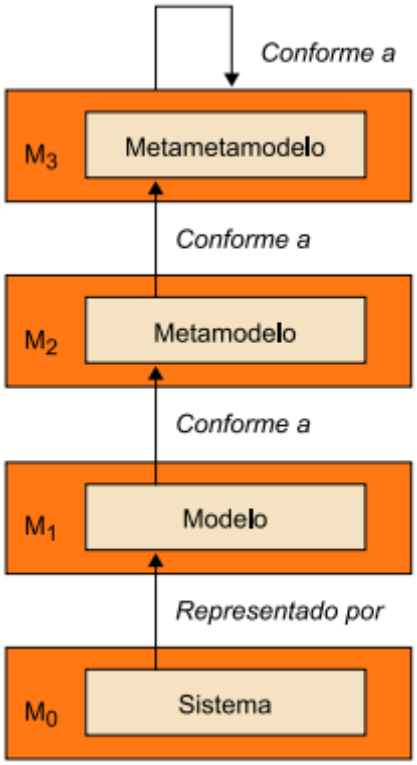

Figura 3.1. Arquitectura de los cuatro niveles [24]

3.1.4 Generación de Código e Interpretación de Modelos

Para entender un modelo es necesario determinar su interpretación, es decir, la relación entre cada elemento del modelo y la realidad que representa. La traducción del modelo a otro lenguaje (lenguaje natural o de programación) permite definir, sin ambigüedad, la semántica del modelo. Además, si queremos que el modelo sea ejecutable es necesario tener un motor ejecución. Hay dos maneras alternativas para establecer la semántica del modelo y que, a su vez, sean ejecutables:

- Generación de código. Un generador de código se puede pensar como un "compilador de modelos", que genera código ejecutable desde un modelo de alto nivel para crear

una aplicación funcional. Esta generación se suele realizar con plantillas con huecos que se rellenan una vez estén instanciados los elementos del modelo.

- Interpretación de modelos. Se basa en la implementación de un motor genérico que traduce y ejecuta el modelo sobre la marcha. Es el mismo enfoque que para los lenguajes de programación interpretados.

3.2 Eclipse Modeling Framework

Eclipse es un entono de desarrollo de código abierto, extensible. Consiste en un núcleo y un entorno de desarrollo en Java al que se le puede añadir diferentes componentes llamados Plugins.

Eclipse Modeling Framework (EMF) es un framework de Eclipse que ofrece soporte a la construcción de las herramientas y soluciones MDD. A partir de una especificación del modelo, genera una implementación en Java del mismo y unos editores básicos para poder instanciarlo.

Aunque dentro del EMF se pueden encontrar muchos plugin para el modelado, a continuación se van a explicar los utilizados para el desarrollo de este proyecto.

Para este apartado se han utilizado principalmente como fuentes los libros:

- *Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas*, J. G Molina y otros. Ed Ra-Ma. (2013) [23]
- *Eclipse Modeling Framework*, Frank Budinsky y otros, Addison Wesley, (2003) [25]

3.2.1 Ecore

El modelo usado para representar los meta-modelos en EMF se llama Ecore. Es en un subconjunto de diagramas UML (Unified Modeling Language) simplificados, utilizados para describir el meta-modelo.

A continuación, mostraremos un modelo simplificado de cómo se relacionan los componentes.

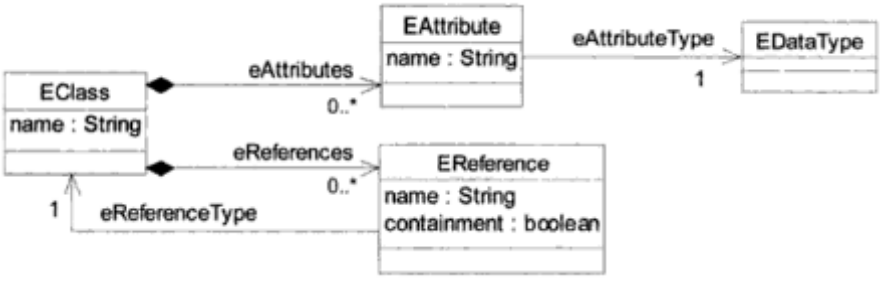

Figura 3.2. Modelo simplificado de la relación de los componentes de Ecore [25]

- EClass se usa para representar las clases de nuestro meta-modelo. Tiene nombre, cero o más atributos y cero o más referencias.
- EAttribute se usa para representar los atributos de las clases de nuestro modelo. Contiene un nombre, un tipo y una cardinalidad.

- EReference se usa para representar una asociación entre clases. Tiene un nombre, un tipo que referencia a otra clase y un booleano para indicar si es una relación de composición o no.
- EDataType se usa para representar los tipos de los atributos. Pueden ser datos primitivos de Java, como int o double u objetos como java.lang.String o java.util.Date.

3.2.2 Xtext

Xtext es un framework de Eclipse dirigido para la creación de sintaxis textual de modelos. Asocia una representación textual (sintaxis concreta) a los elementos del meta-modelo (sintaxis abstracta). Xtext, además del analizador sintáctico, proporciona resaltado de las palabras clave, completado de código, análisis estático y validación de modelos y generación de código a partir de los modelos.

El primer paso con Xtext es definir la gramática del lenguaje. Para ello podemos partir desde un meta-modelo ya creado o empezar a definir la gramática desde cero, en cuyo caso al final se generará un meta-modelo a partir de la gramática.

Una vez definida la gramática, Xtext generará una serie de *plugins* de Eclipse en los que se implementará toda la funcionalidad antes mencionada.

3.2.3 Xtend

Xtend es un dialecto de Java más flexible y expresivo. Se centra en proporcionar una sintaxis más concisa y una funcionalidad adicional a la que ofrece Java, como inferencia de tipos (se asigna automáticamente un tipo de datos sin necesidad de escribirlo) o algunas características de la programación funcional. Puede utilizar cualquier biblioteca existente en Java. Además, permite la programación de plantillas, lo que lo hace muy adecuado para la generación de código de un DSL.

3.3 Abstract Syntax Tree

El Árbol de Sintaxis Abstracta – *Abstract Syntax Tree* (AST) es una representación en forma de árbol de un código fuente escrito en un lenguaje de programación. Cada nodo del árbol aporta información de lo que ocurre en el código. La sintaxis es abstracta porque no representa cada detalle del código (por ejemplo, la sintaxis concreta de cada lenguaje), solo la información relevante.

En esta sección se explicará el AST que proporciona Eclipse para código Java. Como fuente en esta sección se han utilizado publicaciones del sitio web de Eclipse [29] [30].

Eclipse internamente genera, de los ficheros Java, un AST que es lo que le permite, por ejemplo, saltar a la declaración de un método o un atributo al pulsar F3, reemplazar el nombre de una variable en todos los sitios del método que es usada al mismo tiempo y gran parte de la funcionalidad de Eclipse que modifica el código fuente. Este AST es comparable a un árbol DOM de un XML. Al igual que el DOM, te permite modificar el árbol y reflejarlo en el código fuente.

Un flujo de trabajo típico de una aplicación que usara AST sería:

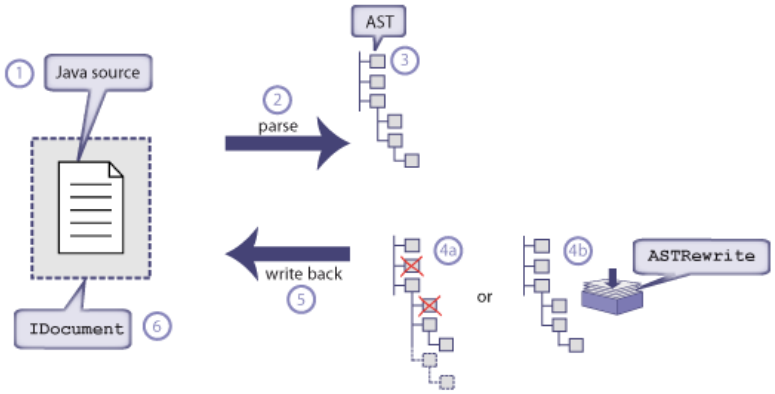

Figura 3.3. Flujo de Trabajo típico con AST [29]

1. **Java Source**: se parte de un fichero Java o de código Java almacenado en un array de caracteres.
2. **Parser**: se analiza el código fuente.
3. **AST**: es el resultado del apartado 2. La información del código fuente es representada en forma de árbol para que sea más sencillo manipularla.
4. **Manipular el AST**: leer o modificar lo que se necesite en el árbol.
5. **Escribir los cambios**: si se hubiesen realizado cambios, estos tienen que estar otra vez en el fichero.
6. **Documento**: volvemos a tener el fichero Java, esta vez con los cambios realizados.

Gracias a la facilidad de acceso que proporciona al código de los ficheros Java, el AST de Eclipse es la herramienta que se usa para la parte del análisis estático en este proyecto.

4 Diseño

Al lenguaje creado en este proyecto se le ha llamado JavaCheck. Como se ha visto anteriormente (capítulo 2) JavaCheck es un DSL para definir reglas esperables en códigos Java que, posteriormente, serán comprobadas en los proyectos que se indiquen.

En este apartado se estudiará la arquitectura, así como las decisiones de diseño tomadas a la hora de crear JavaCheck.

4.1 Arquitectura

Lo que se pretende desarrollar es un *plugin* de Eclipse. Es decir, es una ampliación del entorno de desarrollo y se ejecutará siempre en una instancia de eclipse. Por lo que trabajaremos con el *workspace* y otros elementos propios de eclipse.

Como todo proyecto desarrollado con el paradigma MDD, JavaCheck tiene tres componentes principales (explicados en el capítulo 3):

- Sintaxis abstracta, especificada en un meta-modelo en el que se definen los componentes que debe tener el lenguaje.
- Sintaxis concreta, en este caso textual, definida con Xtext.
- Semántica, un generador de código que, acompañado de una librería programada para este propósito, se encarga de generar el AST de los ficheros Java y de recorrerlos verificando que se cumplen estas reglas.

4.2 Sintaxis de JavaCheck

Se va a comenzar exponiendo unos ejemplos de reglas que después usaremos para ilustrar mejor el diseño:

```
10
11  AttConstantes: Attribute satisfy is modified with [static and final];
12  AttNoConstantes: Attribute satisfy is not modified with [static and final];
13
14  all Method which is not constructor satisfy name type= lower camel case;
15  all Attribute in AttNoConstantes satisfy is not modified with [public] and name type= lower camel case;
16  all Attribute in AttConstantes satisfy name type=upper case;
17
```

Figura 4.1. Ejemplo de JavaCheck 1.

AttrConstantes son todos Atributos constantes, es decir, con los modificadores *static* y *final*. *AttrNoConstantes* son todos los atributos que no pertenecen al primer grupo. La primera regla especifica que todos los métodos que no son constructores deben cumplir que su nombre está representado en *camel case* minúscula. La siguiente indica que todos los atributos no constantes no deben ser públicos y estar escritos en *camel case* minúscula. La tercera regla indica que los atributos constantes deben estar escritos en mayúsculas.

```
61  Equals:  Method satisfy name ="equals" and return type=Primitive.boolean and parameters size=1 types=["Object"];
62  HashCode: Method satisfy name="hashCode" and return type=Primitive.int and parameters size=0;
63
64  exists Class satisfy implements types{"Comparable"};
65
66  all Class which implements types{"Comparable"} satisfy have{
67    one Method in Equals
68  } and have{
69    one Method in HashCode
70  };
71
```

Figura 4.2. Ejemplo de JavaCheck 2.

Equals y *HashCode* representan todos los métodos que sobrescriben a *equals* y *hashCode*, respectivamente, de la clase *Object*. La primera regla indica que tiene que existir, al menos, una clase que implemente la interfaz *Comparable*. Y la siguiente regla especifica que todas las clases que implemente *Comparable* deben tener sobrescrito el método *equals* y el método *hashCode*.

```
74  AttrPrimitive: Attribute satisfy type=isPrimitive();
75
76  all Class using AttrPrimitive which have{
77   exists Attribute in AttrPrimitive
78  } satisfy have{
79   one Method satisfy return type=AttrPrimitive.type and name start "get" and name end AttrPrimitive.name
80  };
81
```

Figura 4.3. Ejemplo de JavaCheck 3.

La última regla indica que todas las clases que tienen atributos de tipo primitivo deben tener un método que devuelva el tipo de uno de los atributos y que empiece por *get* y termine por el nombre de ese atributo, es decir, que si tiene atributos primitivos debe haber un *getter* para uno de ellos.

4.2.1 Sintaxis Abstracta

En este apartado se va a exponer el meta-modelo y a explicar cada uno de los componentes del DSL.

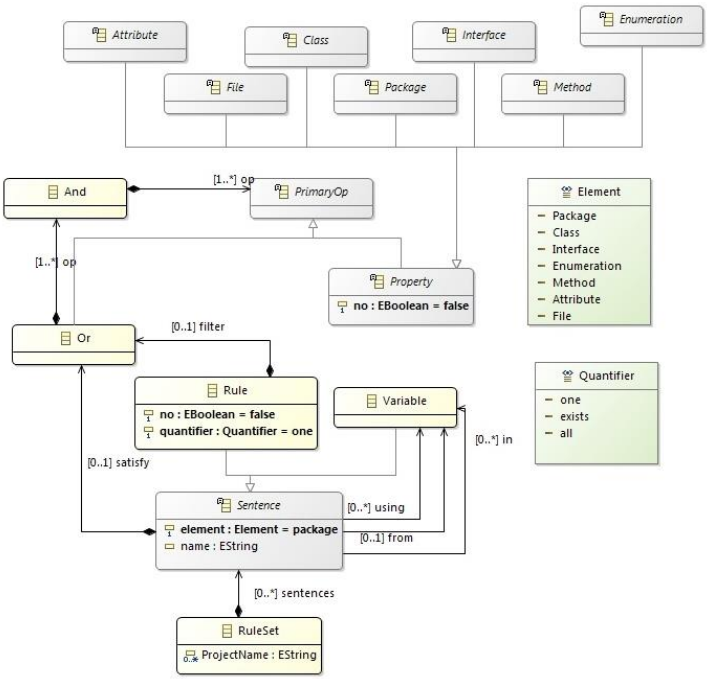

Figura 4.4. Meta-modelo de JavaCheck reducido.

- **RuleSet**: es la clase principal, la que contiene toda la información del modelo. En ella hay un atributo que se llama *ProjectName*. Es un conjunto de *EString* para definir los nombres de los proyectos a lo que se le quieren validar las Reglas. También tiene un conjunto de *Sentences* que son las que van a definir las reglas.

- **Sentence**: hay dos tipos de *Sentence:*

    - **Variable:** son sentencias que no se evalúan como reglas, sino que almacenan una lista de elementos que cumplan las propiedades definidas, para después utilizar el nombre o el tipo de los elementos o de los subelementos en otras propiedades. Todas las propiedades que tienen un atributo para especificar nombres de elementos (clases, tipos, interfaces, enumerados) pueden tomar como valor un String o una referencia a una variable. Son necesarias para buscar, por ejemplo, algunos patrones. De los ejemplos del comienzo del capítulo las variables serían:

```
26 AttConstantes: Attribute satisfy is modified with [static and final];
27 AttNoConstantes: Attribute satisfy is not modified with [static and final];
28
29 Equals:  Method satisfy name ="equals" and return type=Primitive.boolean and parameters size=1 types=["Object"];
30 HashCode: Method satisfy name="hashCode" and return type=Primitive.int and parameters size=0;
31
32
33 AttrPrimitive: Attribute satisfy type=isPrimitive();
```

Figura 4.5. Ejemplo Variables.

Y cada vez que son mencionadas (figuras 4.1, 4.2 y 4.3) podemos ver unos ejemplos de uso.

- **Rule:** son las reglas que se evalúan sobre el código.

Tanto las reglas como las variables se ejecutan sobre unos tipos de elementos que pueden ser *File*, *Package*, *Interface*, *Class*, *Enum*, *Method* o *Attribute.* Estos tipos están definidos en un enumerado llamado **Element** y es uno de los atributos de *Sentence* (es decir, común para *Rule* y *Variable*). El otro atributo es un EString, para almacenar el nombre de las variables y después referirnos a ellas. *Sentence* también tiene dos relaciones con Variable:

- **In:** esta relación de la sentencia reduce el número de elementos sobre los que mirar. Es decir, si queremos mirar la propiedad en las clases y ponemos en la referencia *in* una variable, en lugar de validar la propiedad en todas las clases, las valida sólo en el conjunto de clases que están en la variable (las que cumplen las propiedades de la variable).

- **From:** esta relación es bastante parecida a la anterior, salvo que en lugar de mirar en el conjunto de una variable, mira en los subelementos de cada uno de los objetos de la variable. Es decir, si estamos creando una *Sentence* que se evalúa sobre atributos, deberíamos poner en *from* una referencia a una variable que se evalúe, por ejemplo, sobre clases. Así pues, la *Sentence* en lugar de evaluarse sobre todos los atributos, lo hará únicamente sobre los atributos de todas las clases de la variable de la referencia *from.*

- **Using:** esta relación es para declarar las variables que después se usarán en las propiedades

El último elemento que tiene una sentencia es una composición llamada **satisfy** que apunta hacia la clase **Or.** En definitiva, en este componente se define una expresión lógica con **ands**, **ors** y **paréntesis** para poner tantas propiedades como queramos evaluar en cada *Sentence*,

Los elementos que además tiene *Rule* son:

- **No**: un atributo booleano, para expresar la negación de toda la regla.
- **Quantifier:** un cuantificador para regla, para expresar sobre qué cantidad de elementos se debe cumplir la regla para que sea correcta: **all**, **one**, **exist.** Este cuantificador se define con un enumerado. Junto con la negación podemos también indicar ninguno (*no exis*t)
- **Filter:** una composición de la clase *Or*, que como se ha explicado antes se usa para definir las propiedades. Sirve para filtrar los elementos sobre los que comprobar la regla. A continuación se pondrá un ejemplo de regla y sus diferentes partes**:**

```
84
85  rootPackage: Package satisfy name = "root";
86  ClassAbstract: Class satisfy is modified with [abstract];
87  methAbstract: Method satisfy is modified with [abstract];
88
89  no exists Class from rootPackage in ClassAbstract using methAbstract
90   which have {
91   one Method satisfy name=methAbstract.name
92  } satisfy is not superclass;
93
```

Figura 4.6. Ejemplo de JavaCheck 4.

No: **no**
Quantifier: `exists`
Element: `Class`
Fom: `from rootPackage`
In: `in ClassAbstract`
Using `using methAbstract`
Filter:
```
which have {
one Method satisfy is modified with [abstract]
}
```
Satisfy: `satisfy is not superclass;`

- **Property:** son las propiedades que podemos comprobar en las reglas. Tiene un booleano **no** para negar el resultado. Como se ve en el diagrama, hay una clase para cada uno de los elementos que hereda de las propiedades y, a su vez, cada una de las propiedades heredarán de estas clases dependiendo de si se pueden aplicar a ese elemento.

Para completar el meta-modelo, a continuación se muestran las clases que heredan de *Property*, explicando un poco cada una de ellas. El resto del meta-modelo se completa en el Anexo A.

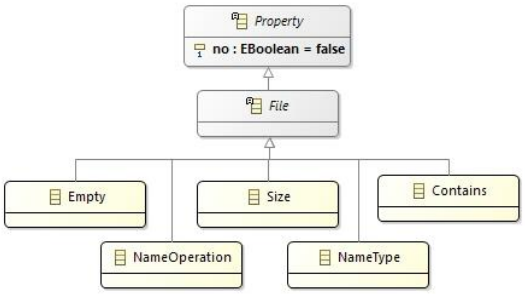

Figura 4.7. Parte del meta-modelo simplificado: propiedades que heredan de File

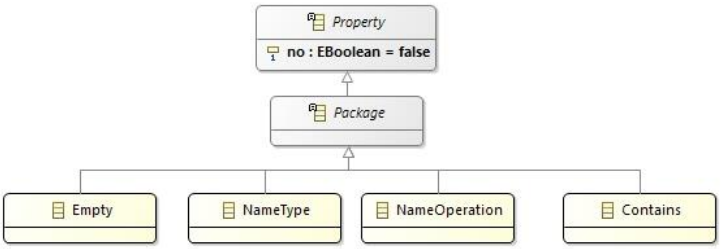

Figura 4.8. Parte del meta-modelo simplificado: propiedades que heredan de Package

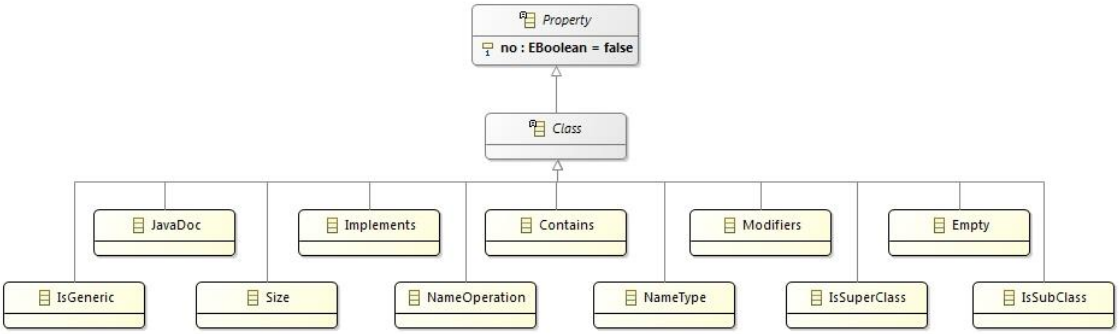

Figura 4.9. Parte del meta-modelo simplificado: propiedades que heredan de Class

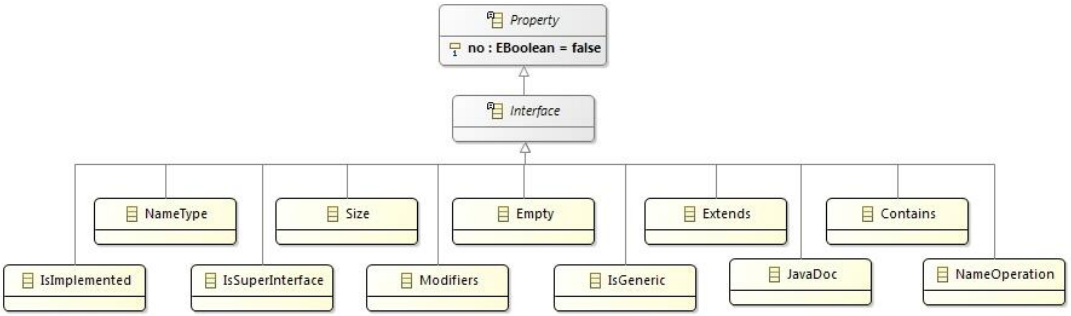

Figura 4.10. Parte del meta-modelo simplificado: propiedades que heredan de Interface

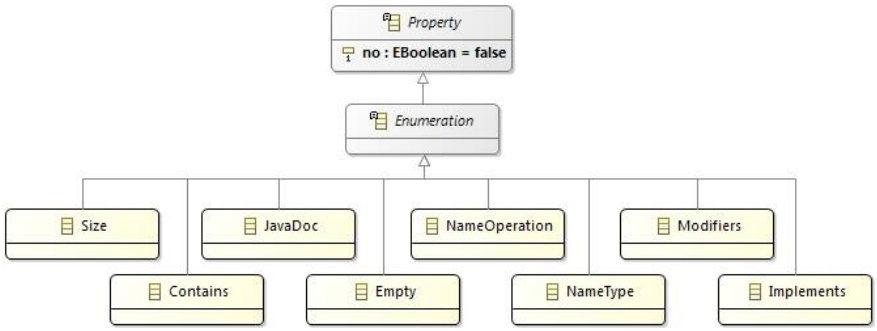

Figura 4.11. Parte del meta-modelo simplificado: propiedades que heredan de Enumeration

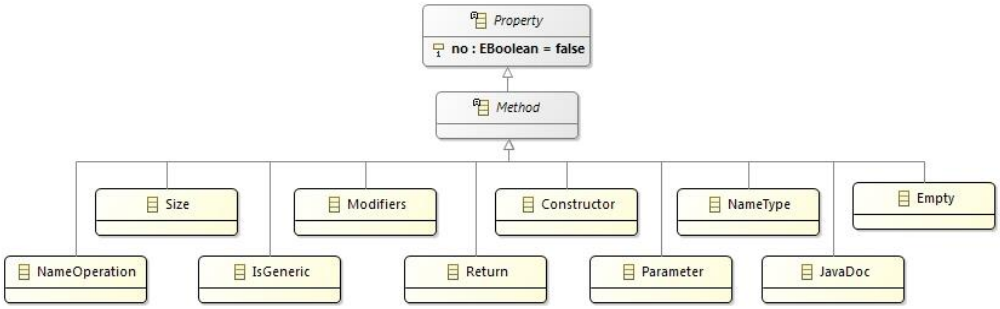

Figura 4.12. Parte del meta-modelo simplificado: propiedades que heredan de Method

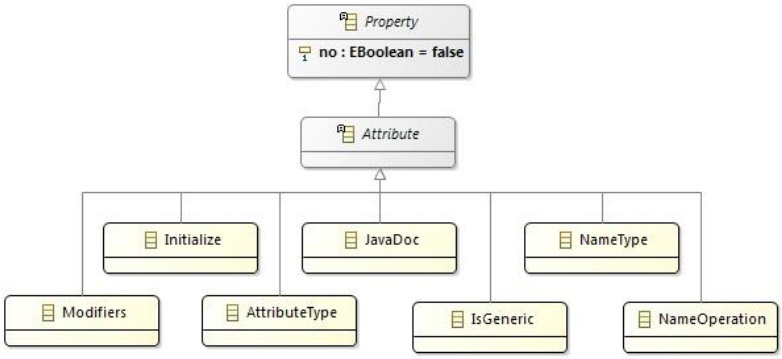

Figura 4.13. Parte del meta-modelo simplificado: propiedades que heredan de Attribute

- Size: propiedad de *File, Class, Interface, Enum* y *Method*, que comprueba que el elemento tenga un cierto tamaño o el tamaño dentro de un rango.
- Empty: propiedad de *File*, *Package, Class, Interface, Enum* y *Method*. Comprueba si un elemento está vacío
- NameType: propiedad común a todos elementos que comprueba si el nombre está escrito siguiendo un estilo como, por ejemplo, *camel case* en minúsculas, todo en mayúsculas o que la primera letra empiece en mayúsculas.
- NameOperation: propiedad común a todos elementos que comprueba el nombre de los elementos: que contenga cierta cadena de caracteres, el prefijo, el sufijo, que sea exactamente igual a otra cadena de caracteres o que sea sinónimo de alguna palabra.

- Contains: propiedad de *File*, *Package*, *Class*, *Interface* y *Enum*, que comprueba si ese elemento contiene otros elementos que sigan otra regla.
- IsGeneric: propiedad de *Class*, *Interfaces*, *Method* y *Attribute*. Comprueba si ese elemento es genérico. Los métodos genéricos son aquellos que tienen algún parámetro o el retorno de tipo genérico. Los atributos genéricos son los que pertenecen a una clase genérica y el tipo es uno de los parámetros genéricos.
- JavaDoc: propiedad de *Class*, *Interface*, *Enum*, *Method* y *Attributes*, que comprueba si el elemento tiene el comentario *JavaDoc* y qué etiquetas del comentario están puestas.
- Modifiers: propiedad de *Class*, *Interface*, *Enum*, *Method* y *Attributes*, que comprueba los modificadores que tiene el elemento.
- IsSuperClass: comprueba si una clase es una superclase (es decir, hay otras clases que heredan de ella), cuántas clases heredan de ella y qué clases heredan de ella.
- IsSubClass: comprueba si una clase hereda de otra y de qué clase hereda.
- Implements: comprueba si una clase implementa interfaces, cuántas interfaces implementa y qué interfaces.
- IsSuperInterface: comprueba si una interfaz es superinterfaz (es decir, que otras interfaces heredan de ella), cuántas interfaces heredan de ella y cuáles.
- Extends: comprueba si una interfaz hereda de otra, de cuántas interfaces hereda y de cuáles.
- IsImplemented: comprueba si una interfaz es implementada por una clase o por un enumerado, cuantas veces es implementada y por qué clases o enumerados.
- Constructor: propiedad para comprobar si los métodos son constructores.
- Parameter: propiedad que comprueba el número de parámetros de un método, si hay exactamente un número o está dentro de un rango y de qué tipo es alguno o todos.
- Initialize: comprueba si un atributo está inicializado en su declaración.
- AttributeType: comprueba que el tipo de un atributo sea de cierto tipo.

Restricciones

Para el correcto funcionamiento del lenguaje es necesario crear algunas restricciones.

- Los proyectos que se quieran analizar deben estar en el *workspace*.
- El tipo de todas las propiedades de una sentencia deben ir acorde con el elemento, es decir, no se puede poner una propiedad que hereda de *Method* con un elemento que sea *Package*.
- Cuando se vaya a usar una variable en una propiedad es necesario que la variable esté puesta en el campo *using*.
- No se puede poner en la cláusula *from* una variable cuyo elemento no contenga el elemento de la sentencia. Es decir, no se puede poner una variable que sea de *Attributes* en la cláusula *from* de una sentencia que tiene como elemento *Package* porque los atributos no contienen paquetes.
- La variable de la cláusula *in* de una sentencia debe ser del mismo tipo de elemento que la sentencia.
- Todas las variables que se usen en una sentencia deben haberse declarado antes de la sentencia.
- No puede haber dos o más variables con el mismo nombre.

Aunque hay más restricciones, son específicas para cada propiedad y se explican en el Anexo C.

4.2.2 Sintaxis Concreta

Como ya se ha comentado en el apartado de Arquitectura, la sintaxis concreta de JavaCheck es una sintaxis textual. En este apartado se va a mostrar parte de la sintaxis concreta usando para ello el estándar *Extended Backus-Naur Form* (EBNF)[31].

```
<RuleSet>     ::= 'Projects Name :'(( <STRING> (',' <STRING>)*) | '*') <Sentence> (<Sentence>)? ;
<Sentence>    ::= (<Rule> | <Variable>) ';' ;
<Variable>    ::= <VariableName>':'<Element> ('from' <VariableName>)?
                  ('in' <VariableName> (',' <VariableName>)*)?
                   ('using' <VariableName> (',' <VariableName>)*)? ('satisfy' <Or>);
<Rule>        ::= ('no')? <Quantifiers> <Element> ('from' <VariableName>)?
                  ('in' <VariableName> (',' <VariableName>)*)?
                  ('using' <VariableName> (',' <VariableName>)*)? ('which' <Or>)('satisfy' <Or>);
<Quantifiers> ::= 'all'
                | 'one'
                | 'exist';
<Element>     ::= 'File'
                | 'Package'
                | 'Class'
                | 'Interface'
                | 'Enumeration'
                | 'Method'
                | 'Attribute';
<Or>          ::= <And> ('or' <And>)*;
<And>         ::= <PrimaryOp> ('and' <PrimaryOp>)*
<PrimaryOp>   ::= '('<Or>')'
                | <Property>;
<Property>    ::= <File>
                | <Package>
                | <Class>
                | <Interface>
                | <Enumeration>
                | <Method>
                | <Attribute>;
```

La sintaxis concreta de cada una de las propiedades se especifica en el Anexo B.

4.3 Semántica de JavaCheck

Para la semántica del proyecto se ha decidido crear una librería en la que se implementa la creación y el manejo del AST, usado el paquete de eclipse org.eclipse.jdt.core.dom, donde se encuentran todas las clases necesarias para ello.

También se genera código en el cual se crearán las reglas específicas que se han definido y con el que se ejecutará el analizador.

5 Implementación

En este capítulo se expondrá la implementación del diseño propuesto en el capítulo 4. Para la implementación se ha utilizado diferentes componentes de EMF y el AST de Eclipse explicados en la sección 3.1:

- Para implementar el meta-modelo se ha utilizado el editor gráfico en forma de árbol de Ecore.
- La sintaxis concreta se ha definido con Xtext.
- Para la semántica se ha utilizado Xtend para generar código, llamando a métodos de una librería propia que se encarga de controlar el AST de eclipse.

5.1 Meta-Modelo

Para el meta-modelo de JavaCheck se ha utilizado el editor gráfico en forma de árbol de Eclipse para Ecores. La implementación del meta-modelo corresponde con la descripción del diseño, que se completa en el Anexo A.

Una vez se dispone de un meta-modelo Ecore, se pueden generar automáticamente los elementos en código Java. Dicha generación consta una interfaz para cada elemento y una implementación de cada interfaz.

5.2 Sintaxis Concreta

Para definir sintaxis concreta, como ya se ha indicado, se ha usado Xtext. La sintaxis sigue fielmente el diseño del Anexo B, por lo que solo se mostrarán algunos ejemplos de la implementación con Xtext.

```
7  RuleSet returns RuleSet:
8      'Projects' 'Name:' ((ProjectName+=EString ( "," ProjectName+=EString)* )|'*')
9       sentences+=Sentence (sentences+=Sentence)* ;
10
11 Sentence returns Sentence:
12   (Variable | Rule)';';
```

Figura 5.1. Sintaxis concreta Xtext: *RuleSet* y *Sencence*.

```
14 enum Element returns Element:
15         Package = 'Package' | Class = 'Class' | Interface = 'Interface'
16         | Enumeration = 'Enumeration' | Method = 'Method' | Attribute = 'Attribute'
17         | File = 'File';
18
```

Figura 5.2. Sintaxis concreta Xtext: *Enum Element*.

```
194 enum Quantifier returns Quantifier:
195         one = 'one' | exists = 'exists' | all = 'all';
196
```

Figura 5.3. Sintaxis concreta Xtext: *Enum Quantifier*.

En la figura 5.1 están las reglas para *RuleSet* y *Sentence*. A cada atributo o referencia se le asigna un valor con el signo. Para ello se usa el signo =, o += para añadir elementos a una lista. Si el atributo es booleano se usará el signo ?=.

Como se muestra en la figura 5.2 cuando la función devuelve un enumerado del meta-modelo primero se pone la palabra clave *enum.*

```
Variable returns Variable:
  name=EString':'
  element=Element ('from' from=[Variable|EString])? ('in' in+=[Variable|EString](',' in+=[Variable|EString])*)?
  ('using' using+=VariableSubtype(',' using+=VariableSubtype)*)?('satisfy' satisfy=Or)?;


Rule returns Rule:
  (no?='no')? quantifier=Quantifier element=Element ('from' from=[Variable|EString])?
  ('in' in+=[Variable|EString](',' in+=[Variable|EString])*)?('using' using+=VariableSubtype(',' using+=VariableSubtype)*)?
  ('which' filter=Or)? ('satisfy' satisfy=Or)?;

Or returns Or:
   op+=And ( 'or' op+=And)*;

And returns And:
  op+=PrimaryOp ( 'and' op+=PrimaryOp)*;

PrimaryOp returns PrimaryOp:
  '('Or')'|Property;

```

Figura 5.4. Sintaxis concreta Xtext: *Variable, Rule, Or, And y PrimaryOp*.

Como se ve la implementación es sencilla de entender y sigue el diseño mostrado en el Anexo B.

Restricciones

Tras definir la sintaxis concreta en un fichero con extensión xtex, el entorno te proporciona una serie de ficheros para implementar parte de la funcionalidad mencionada en la sección 3.2.2 (completado de código, análisis estático y validación de modelos y generación de código a partir de los modelos).

Entre estos ficheros se encuentra el *XXXValidator.xtend*, donde XXX se sustituye por el nombre del lenguaje, en el que se pueden definir restricciones para el lenguaje con Xtend (sección 3.2.3).

A continuación, pondremos algunos ejemplos de la implementación de estas restricciones.

```
@Check
def checkProject(RuleSet rs) {
  if (!rs.projectName.isEmpty) {
    var workspace = ResourcesPlugin.getWorkspace().getRoot();
    for (name : rs.projectName) {
      if (name == "") {
        error("The project " + name + " is not into worksapce",
          JavaRulePackage.Literals.RULE_SET__PROJECT_NAME, "invalidProject")
      } else {
        var project = workspace.getProject(name)
        if (!project.exists) {
          error("The project " + name + " is not into worksapce",
            JavaRulePackage.Literals.RULE_SET__PROJECT_NAME, "invalidProject")
        }
      }
    }
  }
}
```

Figura 5.5. Restricción: todos los proyectos deben estar en el *workspace*.

Esta primera restricción especifica que todos los proyectos que se declaren para analizar deben estar en el *workspace* de eclipse.

Como se ve la sintaxis es muy similar a la de Java. Lo primero que encontramos es la anotación *@Check* necesaria para que Xtext sepa que es una restricción del lenguaje y la valide.

Para definir un método usamos la palabra clave *def*. *N*o es necesario poner un valor de retorno puesto que Xtend es un lenguaje que usa inferencia de tipos (se asigna automáticamente un tipo de datos sin necesidad de escribirlo). Y, dentro de los paréntesis los parámetros, en este caso debe ser un único parámetro que se corresponda a un elemento del meta-modelo sobre el cual se quiera validar la restricción.

Para leer los campos de un objeto se usa punto seguido del nombre del campo (rs.projectName), esto es equivalente a los métodos getters de Java. Y para escribir en los campos de un objeto, hay que poner tras el punto el nombre del campo seguido de igual (rs.projectName=), es equivalente a los métodos setters de Java.

Así pues, en la primera línea comprobamos que la lista de nombre de la clase *RuleSet* no esté vacía. En caso de que esté vacía significa que no se ha escrito ningún nombre, por lo que se deben validar sobre todos los proyectos del *workspace*. Recorremos toda la lista de nombres y si tenemos nombre vacío o no el proyecto no está en el *workspace* lanzamos un error.

Para lanzar un error se usa el método error. Recibe un *String* que es el mensaje que se mostrará y un enumerado para indica sobre que elemento del meta-modelo está el error (parte que marcará en rojo).

El error descrito anteriormente quedaría así:

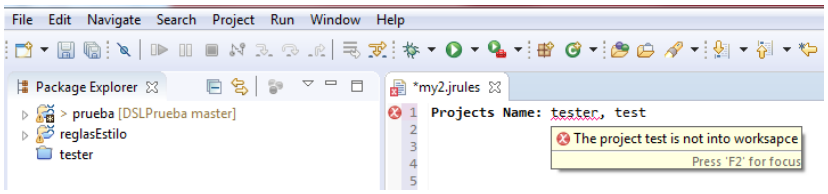

Figura 5.6. Ejemplo de *error*.

En lugar de errores también se pueden lanzar advertencias cambiando el método *error* por el método *warning.* A continuación, se muestra un ejemplo de cómo codificar una advertencia y como quedaría:

```
244  @Check
245  def checkJavaDoc(JavaDoc jd) {
246
247    var s = getSentece(jd);
248    if (s.element != Element.METHOD && jd.parameter) {
249      warning("The tag @parameter is used for methods", JavaRulePackage.Literals.JAVA_DOC__PARAMETER,
250        'inadvisableJavaDoc')
251    }
252    if (s.element != Element.METHOD && jd.^return) {
253      warning("The tag @return is used for methods", JavaRulePackage.Literals.JAVA_DOC__RETURN,
254        'inadvisableJavaDoc')
255    }
256    if (s.element != Element.METHOD && jd.throws) {
257      warning("The tag @throws is used for methods", JavaRulePackage.Literals.JAVA_DOC__THROWS,
258        'inadvisableJavaDoc')
259    }
260  }
261
```

Figura 5.7. Restricción: uso de las etiquetas @return @parameter y @throws, del comentario de documentación, reservadas para métodos.

La restricción que se implementa en la figura 5.7 sirve para advertir que las etiquetas *@parameter @return* y *@throws* del *JavaDoc* se usan específicamente para métodos y no tiene sentido ponérselas a otros elementos Java.

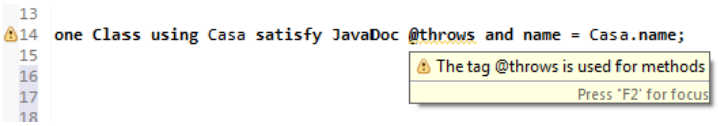

Figura 5.8. Ejemplo de *warning*

Se muestra un ejemplo de cada restricción en el Anexo C

5.3 Semántica

La semántica de JavaCheck, como ya se ha mencionado antes, se define con una librería y el generador de código.

En la librería están definidas todas las funcionalidades, mientras que el generador de código declara las reglas específicas que se han codificado llamando a la librería.

5.3.1 Librería AST

Encargada de generar y trabajar sobre el AST. Vamos a explicar un poco la estructura de la librería y algo de código de sus funcionalidades principales. Lo primero que vamos a mencionar son las clases necesarias para crear el AST y manejarlo a más bajo nivel, es decir, accediendo a los nodos.

```java
public class ParserAst {
    /**
     * Funcion para crear el AST de un código JAva
     *
     * @param str String con el código Java
     * @param visitor visitor del AST, recorre todo el árbol
     * @return CompilationUnit
     */
    public static CompilationUnit parse(String str, ASTVisitor visitor) {

        ASTParser parser = ASTParser.newParser(AST.JLS8); //creamos objeto ASTParser asignando una versión de Java, en este caso Java 8
        parser.setResolveBindings(true); //Configuracion de algunas opciones.
        parser.setBindingsRecovery(true);
        parser.setSource(str.toCharArray());//Se pasa la cadena de caracteres con el codigo para parsear.
        parser.setKind(ASTParser.K_COMPILATION_UNIT);
        Map<String, String> options = JavaCore.getOptions(); //Configurancion de opciones para poder acceder a los tipos Enumerados.
        options.put(JavaCore.COMPILER_SOURCE, JavaCore.VERSION_1_5);
        parser.setCompilerOptions(options);

        //Se crea el AST
        final CompilationUnit cu = (CompilationUnit) parser.createAST(null);
        //Se le añade el visitor.
        cu.accept(visitor);
        if (visitor instanceof UnitVisitor){
            ((UnitVisitor)visitor).setComp(cu);
        }
        return cu;
    }
```

Figura 5.9. Método *parse* de la librería.

```java

public class UnitVisitor extends ASTVisitor{

    private String nameFile;
    private String path;
    private CompilationUnit comp;
    private PackageDeclaration packageDeclaration;
    private List<MInterface> interfaces;
    private List<MClass> classes;
    private List<MEnumeration> enumerations;
    private List<MMethod> methods;
    private List<MAttribute> attributes;

    public UnitVisitor(String nameFile, String path) {

    @Override
    public boolean visit(PackageDeclaration node) {
        packageDeclaration = node;
        return super.visit(node);
    }
    @Override
    public boolean visit(TypeDeclaration node) {
        if (node.isInterface()) {
            interfaces.add(new MInterface(node, this));
        } else {
            classes.add(new MClass(node, this));
        }
        return super.visit(node);
    }
    @Override
    public boolean visit(EnumDeclaration node) {
        enumerations.add(new MEnumeration(node, this));
        return super.visit(node);
    }

    @Override
    public boolean visit(MethodDeclaration node) {
        methods.add(new MMethod(node, this) );
        return super.visit(node);
    }

    @Override
    public boolean visit(FieldDeclaration node) {
        attributes.add(new MAttribute(node, this));
        return super.visit(node);
    }
```

Figura 5.10. Clase *UnitVisitor*.

Al crear el AST (figura 5.9) en el método *parse*, hay que pasarle una clase *ASTVisitor* (con el método *accept(ASTVisitor)*) que se encarga de recorrer todo el árbol con los métodos *visitor*. La clase *UnitVisitor*, que hereda de *ASTVisitor*, guardado los nodos en listas, a las cuales se pueden acceder más tarde. Nótese que no guarda directamente el nodo sino que lo encapsula dentro de otro objeto, esto se hace para facilitar el acceso a las propiedades de cada nodo. Así pues tenemos las clases *MAtributo, MMethod MInterface, MClass, MEnumeration, MPackage* y *MFile* para encapsular. La figura 5.11 muestra el diagrama que siguen esas clases.

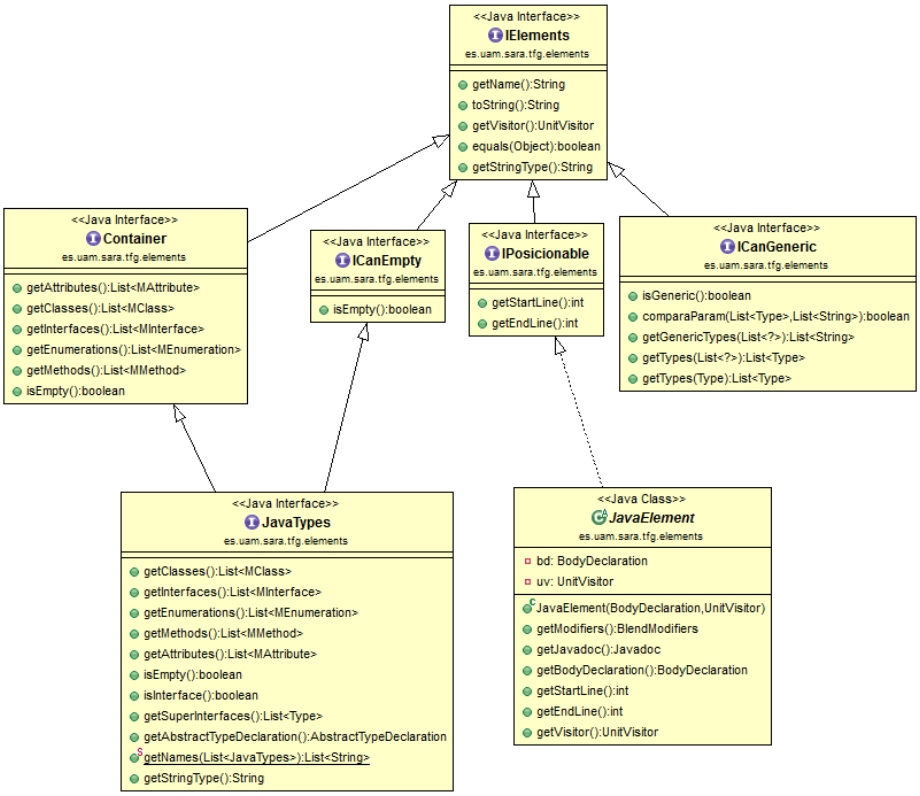

Figura 5.11. Diagrama de Clases de los elementos de la librería.

- *MAtributo* hereda de *JavaElement* e implementa *ICanGeneric.*
- *MMethod* hereda de *JavaElement* e implementa *ICanGeneric* e *ICanEmpty*.
- *MInterface* hereda *JavaElement* e implementa *JavaTypes* e *ICanGeneric.*
- *MClass* hereda *JavaElement* e implementa *JavaTypes* e *ICanGeneric*
- *MEnumeration* hereda *JavaElement* e implementa *JavaTypes*.
- *MPackage* implementa *Container* e *ICanEmpty*
- *MFile* implementa *Container*, *IPosicionable* e *ICanEmpty*

A continuación vamos a comentar las clases que se encargan de ejecutar las reglas. Las sentencias se organizan con una estructura de clases muy parecida al meta-modelo. De esta manera, como se muestra en la figura 5.12, *sentence* es una clase abstracta que tiene como atributos: los elementos sobre los que se va a evaluar, *from* (contiene directamente los subelementos de un elemento superior, por lo que funciona como un *in*), *in*, *using*, *filter* (que por una cuestión de codificación está en *sentence* en lugar de en *rule*, pero *variable* siempre se crea con un *filter* nulo) y *satisfy*. Además tiene un método *analize* que se encarga de separar los elementos que cumple la regla de los que no.

La clase *variable* es una extensión de la clase *Sentence* sin añadir funcionalidad. Pero la clase Rule, que también hereda de *Sentence*, añade dos atributos que son un booleano para el valor de *no* y *quantifier*, y el método *check* que devuelve si la regla se cumple en el proyecto o no.

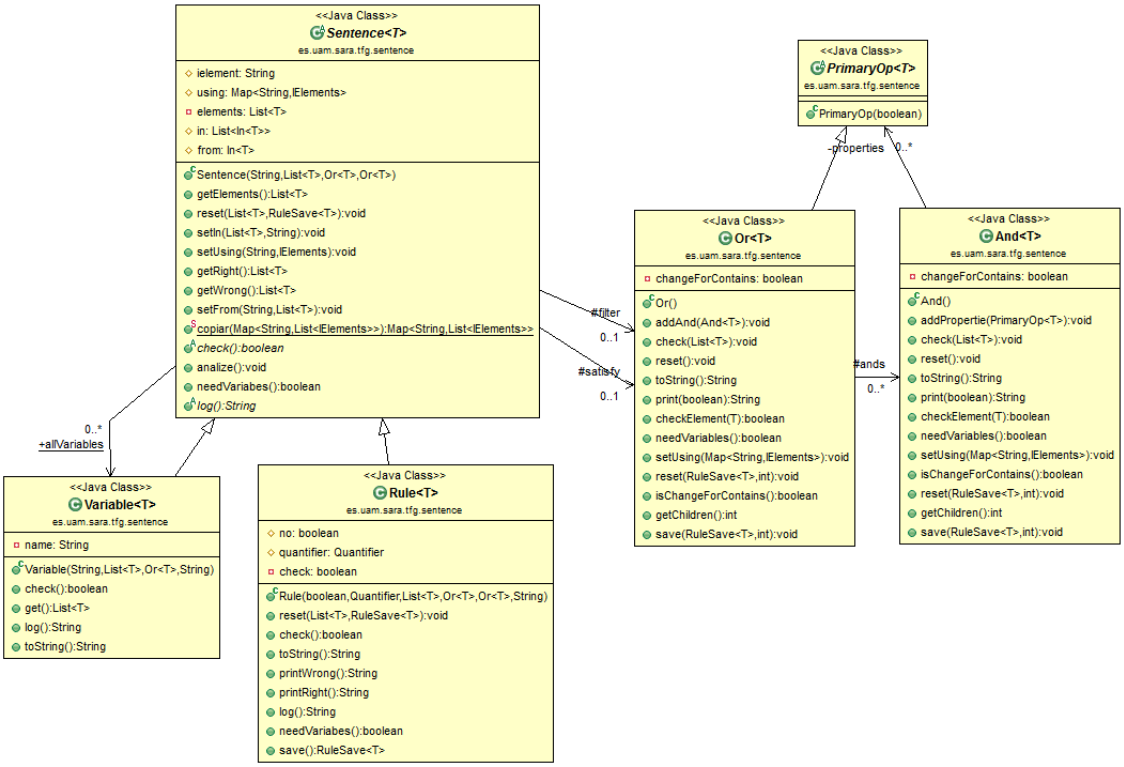

Figura 5.12. Diagrama de Clases *sentence*.

Como muestra la figura 5.12, todas las clases que ejecutan las *sentence* son genéricas, para que valgan para cualquier elemento (*MAtributo, MMethod MInterface, MClass, MEnumeration, MPackage* o *MFile*)

Por último, vamos a explicar la estructura de las propiedades (figura 5.13). Todas las propiedades heredan de la clase abstracta *Checkeable*. Dicha clase tiene un atributo booleano para la negación de la propiedad y dos listas, una para guardar los elementos que cumplen la propiedad y otro para los elementos que no la cumplen. Además, tiene un método llamado *check(List<T> analize)* (figura 5.14), que comprueba si sobre la lista se cumple la propiedad concreta y los guarda en las listas correspondientes y un método que debe implementar cada propiedad *checkElements(T analize)*, que comprueba si cada propiedad se cumple sobre un elemento en particular.

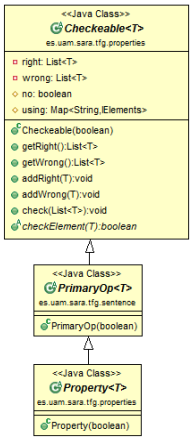

Figura 5.13. Diagrama de Clases de las propiedades.

```
94      public void check(List<T> analyze) {
95          for (T t : analyze) {
96              if (no) {
97                  if (!checkElement(t)) {
98                      addRight(t);
99                  } else {
100                     addWrong(t);
101                 }
102             } else {
103                 if (checkElement(t)) {
104                     addRight(t);
105                 } else {
106                     addWrong(t);
107                 }
108             }
109         }
110     }
111
112     public abstract boolean checkElement(T analyze);
113
```

Figura 5.14. Métodos *check* y *checkElements* de la clase *Checkeable*.

Las propiedades pueden ser genéricas, como *NameOperation* que se ejecuta sobre cualquier elemento (figura 5.15), restringirse a un elemento, como *Initialize* que es específica para *MAttribute* únicamente (figura 5.16), o una cosa intermedia usando la jerarquía de clases de la figura 5.11, como *IsGeneric* que es una propiedad de clases, atributos, métodos e interfaces (figura 5.17).

```
15 public class NameOperation<T extends IElements> extends StringProperty<T> {
16
```

Figura 5.15. Declaración de la clase *NameOperation*.

```
15 public class Initialize extends Property<MAttribute>{
16
```

Figura 5.16. Declaración de la clase *Initializa*.

```
6 public class IsGeneric<T extends ICanGeneric> extends Property<T> {
7
```

Figura 5.17. Declaración de la clase *IsGeneric*.

5.3.2 Generador de código.

El generador de código genera dos ficheros, el primero es el fichero *Main.java*, que es la clase ejecutable del analizador, el otro es el fichero *RuleFactory.java* que crea y ejecuta todas las reglas.

La clase *main* crea los ASTs, llamando al método *parser*, de todos los proyectos que se van a analizar. Y después pasa esos ASTs a la clase *RuleFactory* para que pueda ejecutar las reglas.

RuleFactory se encarga de crear las reglas y después las ejecuta: Para crear los ficheros se hace uso de plantillas programadas en Xtend.

```
143         //Generar sentencias
144         «FOR Sentence s : sentences»
145           «IF s instanceof Variable»
146             «var v= s as Variable»
147             «genetateVariable(v)»
148             sentence.add(«s.name»);
149             Sentence.allVariables.put("«v.name»", «v.name»);
150           «ELSE»
151             «var r= s as Rule»
152             «IF r.eContainer instanceof RuleSet»
153               «r.name="rule"+i»
154               «genetateRule(r, ""+(i++))»
155               sentence.add(«r.name»);
156             «ENDIF»
157           «ENDIF»
158         «ENDFOR»
159
160         //Ejecutar las sentencias
161         «generateDependences(getPrimarySencence(sentences))»
162         return sentence;
163         }
164       }
```

Figura 5.18. Plantilla de *RuleFactory*.

En la figura 5.18 se usa una plantilla en Xtend para generar el código de las reglas, de manera que recorre toda una lista de sentencias y dependiendo de si son reglas o variables genera el código de una manera o de otra, y después llama a los métodos para ejecutar las sentencias. Y en la figura 5.19 se muestra un ejemplo de cómo sería un código generado por esa plantilla.

```
// all Attribute in AttConstantes satisfy name type=upper case;
Or<MAttribute> orFilter3=   null;
Or<MAttribute> or3= new Or<MAttribute>();
And<MAttribute> and31 = new And<MAttribute>();
Property<MAttribute> p311= new NameType<MAttribute>(NameType.Type.UPPER_CASE);
and31.addPropertie(p311);

or3.addAnd(and31);

Rule<MAttribute> rule3=new Rule<MAttribute> (false, Rule.Quantifier.ALL,visitors.getAttributes(),orFilter3, or3, "attribute");
sentence.add(rule3);
```

Figura 5.19. Regla generada a partir de una plantilla.

6 Pruebas y resultados

Después de toda la implementación de JavaCheck, las pruebas que se han realizado se han centrado en determinar tres aspectos fundamentales:

- Expresividad: intentar determinar cómo de expresivo es JavaCheck y qué limitaciones tiene.
- Utilidad: probarlo sobre proyectos para intentar detectar errores reales.
- Escalabilidad: comprobar si es un analizador útil para ser usado a gran escala, en proyectos grandes.

6.1 Expresividad de JavaCheck

Para comprobar la expresividad lo primero en lo que nos fijamos es en las reglas de estilo, viendo hasta qué punto podemos expresarlas. Para ello, se va a codificar un poco del estándar de estilo de Oracle *Sun Code Conventions* [3] utilizando JavaCheck.

JavaCheck no puede expresar las reglas de estilo que se centran en la posición de los elementos como por ejemplo:

- Los atributos de clase deben ordenarse según su visibilidad: primero los públicos, después los protegidos y por último los privados
- Los atributos de instancia van tras los atributos de clase y deben ordenarse según su visibilidad: primero los públicos, después los protegidos y por último los privados
- Los métodos constructores van tras la declaración de atributos y después el resto de métodos.

Sin embargo, aunque no el orden, sí se puede expresar el contenido de ficheros, paquetes, clases, interfaces y enumerados y comprobar la visibilidad de estos. En la figuras 6.1, 6.2 y 6.3 se muestra como comprobar la visibilidad de un elemento y en las figuras 6.1 y 6.2 se muestra un ejemplo de cómo se referencia al contenido de los elementos.

```
105 all Class satisfy have{exists Attribute satisfy is modified with [public and static]};
```

Figura 6.1. Regla: todas las clases tienen al menos un atributo de clase (estático) público.

```
13  AttNoConstantes: Attribute satisfy is not modified with [static and final];
14
15  all Attribute in AttNoConstantes satisfy is not modified with [public];
```

Figura 6.2. Regla: no hay atributos públicos a menos que sean constantes (*static* y *final*).

```
108 PublicClass: Class satisfy is modified with [public];
109 PublicInter: Interface satisfy is modified with [public];
110
111 all File satisfy
112 ( have{one Class in PublicClass} and don't have {exists Interface in PublicInter})or
113 (don't have{exists  Class in PublicClass} and  have {one Interface in PublicInter});
114
```

Figura 6.3. Regla: todos los ficheros tienen una única Clase o Interfaz pública.

El tamaño máximo de una línea tampoco puede determinarlo. No obstante, sí puede el tamaño máximo de un fichero, de una clase, de una interfaz, de un enumerado o de un método:

```
116 all File satisfy size[0..2000];
```

Figura 6.4. Regla: todos los ficheros tienen un tamaño máximo de 2000 líneas.

Los únicos comentarios que se pueden comprobar con JavaCheck son los comentarios de documentación. Se puede comprobar que una clase, interfaz, métodos o atributo los tenga y qué etiquetas son las que tiene:

```
163 all Method satisfy JavaDoc @parameter @return @version @throws;
```

Figura 6.5. Regla: todos los métodos tienen el comentario JavaDoc con etiquetas @parameter, @return, @version y @throws.

También se pueden comprobar que los atributos están inicializados en su declaración:

```
23 all Attribute satisfy is initialize;
```

Figura 6.6. Regla: todos los atributos están inicializados en su declaración.

Por último, dentro de reglas de estilo tendríamos las convenciones de nombres, de las cuales se podría mirar en que formato están escritas, pero no si son sustantivos o verbos. No obstante, lo que sí se ha añadido es que el nombre empiece, termine, contenga, sea exactamente igual o sinónimo de otra palaba. En las figuras 6.7, 6.8 y 6.9 están algunas de las distintas maneras que hay en JavaCheck de formato de nombres. Y en la figura 6.10 se comprueba que el nombre de usa clase sea sinónimo de la palabra *User* y que tenga un atributo con nombre *name.*

```
19 all Method which is not constructor satisfy name type= lower camel case;
```

Figura 6.7. Regla: todos los métodos que no son constructores empiezan por minúscula en *camel case.*

```
21 all Attribute in AttConstantes satisfy name type=upper case;
```

Figura 6.8. Regla: todos los atributos constantes están escritos en mayúscula.

```
28 all Class satisfy name type=start upper case;
```

Figura 6.9. Regla: Todas las clases empiezan por mayúscula en *camel case*

```
33 one Class satisfy name like "User", English and have {
34   one Attribute satisfy name ="name"
35 };
```

Figura 6.10. Regla: existe una clase que se llama *user* o sinónimo de *user* en inglés y que tiene un atributo que se llama *name*.

JavaCheck no analiza el código que hay dentro de los métodos, se centra más en las declaraciones de los diferentes elementos, por lo que no mira reglas de estilo de los distintos tipos de sentencias (sentencias *for*, *if*, *while*, asignaciones, operaciones o llamadas a métodos).

Además de reglas de estilo hay una serie de reglas de programación que normalmente se deben cumplir en los proyectos, como, por ejemplo, que todas las clases abstractas tienen alguna clase hija (figura 6.11) o que una interfaz es implementada o tiene alguna interfaz hija.

```
37 all Class which is modified with [abstract] satisfy is superclass;
```

Figura 6.11. Regla: todas las clases abstractas tiene alguna clase hija.

Por último, se pueden querer definir reglas específicas para cada programa, como pueden ser comprobar el nombre de los elementos, buscar la existencia de patrones de programación [36] (singleton o clases de datos), existencia de clases genéricas o comprobar los parámetros de los métodos. De estas últimas algunas limitaciones que se pueden resaltar son, por ejemplo: que no se puede mirar si un método sobrescribe a otro, no se puede comprobar la declaración de anotaciones y el uso de las mismas y no se puede mirar dentro de las sentencias de un método para asegurarse que no hay métodos con más de tres bucles anidados o no hay métodos recursivos.

6.2 Utilidad de JavaCheck

En este apartado se estudia cómo de útil es JavaCheck. ¿Realmente detecta problemas en el código? Para ello se van a ejecutar unas series de pruebas sobre un proyecto final de una asignatura de programación en Java de la carrera. Se van a comprobar tres tipos de reglas: reglas de estilo, reglas de programación y reglas específicas del proyecto.

```
4  PublicClass: Class satisfy is modified with [public];
5  PublicInter: Interface satisfy is modified with [public];
6
7  AttConstantes: Attribute satisfy is modified with [static and final];
8  AttNoConstantes: Attribute satisfy is not modified with [static and final];
9
10 all File satisfy
11 ( have{one Class in PublicClass} and don't have {exists Interface in PublicInter})or
12 (don't have{exists  Class in PublicClass} and  have {one Interface in PublicInter});
13
14 all File satisfy size [0..2000];
15
16
17 all Attribute in AttNoConstantes satisfy is not modified with [public] and name type= lower camel case;
18 all Attribute in AttConstantes satisfy name type=upper case;
19 all Attribute satisfy is initialize;
20
21 all Method which is not constructor satisfy name type= lower camel case;
22 all Method satisfy size [0..30];
23 all Method satisfy JavaDoc @parameter @return @throws;
24
25 all Enumeration satisfy name type=start upper case;
26 all Enumeration satisfy JavaDoc @author;
27
28 all Class satisfy name type=start upper case;
29 all Class satisfy JavaDoc @author;
30
31 all Interface satisfy name type=start upper case and name start 'I';
32 all Interface satisfy JavaDoc @author;
33
34 all Package satisfy name <> "(default)";
35
```

Figura 6.12. Reglas de estilo para las pruebas de utilidad.

Las reglas de estilo que se han decidido comprobar son:

- Los ficheros tienen una única clase o interfaz pública.
- Los ficheros tienen un tamaño máximo de 2000 líneas.

- Los atributos que no sean constantes (estáticos y finales) deben empezar por minúscula en *camel case*.
- Los atributos que son constantes (estáticos y finales) deben estar escritos en mayúsculas.
- Todos los atributos deben estar inicializados en su declaración.
- El nombre de todos los métodos, que no sean constructores, debe empezar por minúscula en *camel case*.
- El tamaño máximo de un método son 30 líneas.
- Todos los métodos tienen que tener un comentario de documentación con las etiquetas @parameter, @return, @throws.
- Todos los enumerados deben empezar por mayúscula en *camel case*.
- Todos los enumerados tienen que tener un comentario de documentación con la etiqueta @author.
- Todas las clases deben empezar por mayúscula en *camel case*.
- Todas las clases tienen que tener un comentario de documentación con la etiqueta @author.
- Todas las interfaces deben empezar por mayúscula en *camel case* y su nombre empezar por "I".
- Todas las interfaces tienen que tener un comentario de documentación con la etiqueta @author.
- No debe existir el paquete por defecto, es decir, todos los elementos deben estar en un paquete.

Dentro de las reglas de programación se ha decidido comprobar:

```
39 all Class which is modified with [abstract] satisfy is superclass;
40 all Interface satisfy is superinterface or is implemented;
41
42 Equals:  Method satisfy name ="equals" and return type=Primitive.boolean and parameters size=1 types=["Object"];
43 HashCode: Method satisfy name="hashCode" and return type=Primitive.int and parameters size=0;
44
45 all Class which implements types{"Comparable"} satisfy have{
46     one Method in Equals
47   } and have{
48     one Method in HashCode
49   };
50
51 all Method satisfy return type<>"Object";
```

Figura 6.13. Reglas de programación para las pruebas de utilidad.

- Todas las clases abstractas son heredadas.
- Todas las interfaces son heredadas por otra interfaz o implementadas.
- Todas las clases que implementen la interfaz *Comparable* debe sobrescribir los métodos *equals* y *hashCode*
- Ningún método puede tener como tipo de retorno la clase *Object*.

El proyecto consiste en programar una aplicación para la gestión de ventas y subastas de un anticuario. Tras leer las características, se ha decidido que para hacer un análisis de muestra se van a pasar las reglas:

```
Item: Class satisfy name like item, English and is modified with [public and abstract];


one Class in Item satisfy is superclass [3..4] and have{
   one Attribute satisfy name="id" and (type=Primitive.int or type=Primitive.long)
}and have{
   one Attribute satisfy (name like"description", English or name like "name", English) and type=Primitive.String
}and have{
   one Attribute satisfy (name contain "year" or name contain date) and type=Primitive.String
}and have{
   exists Attribute satisfy type=Primitive.double
}and have{
   exists Attribute satisfy type="Date"
};

one Class using Item satisfy is subclass of Item.name and name ="Small" and have{
   one Attribute satisfy type=Primitive.double
};

one Class using Item satisfy is subclass of Item.name and name="Bulky" and  have{
   one Attribute satisfy type=Primitive.double
};

one Class using Item satisfy is subclass of Item.name and name= "WorkOfArt" and  have{
   one Attribute satisfy type=Primitive.String and name contain "author"
}and have{
   one Attribute satisfy type=Primitive.boolean and name contain "certificate"
};


one Class using Item satisfy have{
   one Attribute satisfy type= isCollection(Item.name)
}and have{
   one Method satisfy is constructor and parameters size=4 types=[Primitive.double, Primitive.String, Primitive.String, "Date"]
};
```

Figura 6.14. Reglas de específicas para las pruebas de utilidad.

- Hay una clase con nombre parecido a *Item* que es abstracta y publica que es heredada entre 3 y 4 veces y tiene:
    - Un atributo que se llama *id* cuyo tipo es *int* o *long.*
    - Un atributo que se llama un sinónimo de *description* o de *name* cuyo tipo es *String*.
    - Un atributo que contiene la palabra *year* o *date* de tipo *String.*
    - Atributos de tipo *double.*
    - Atributos de tipo *Date*.
- Hay una clase que hereda de *Item* que se llama *Small* y que tiene un atributo de tipo *double.*
- Hay una clase que hereda de *Item* que se llama *Bulky* y que tiene un atributo de tipo *double.*
- Hay una clase que hereda de *Item* que se llama *WorkOfArts* y que tiene un atributo de tipo *String* que contiene la palabra *author* en el nombre y un atributo de tipo *boolean* que contiene la palabra *certificate.*
- Hay una clase que contiene una colección de *Items* y que tiene un método constructor con 4 parámetros cuyos tipos son: *double*, *String*, *String*, *Date*.

Resultados:

A continuación, vamos a poner unas tablas con los resultados de las pruebas para este proyecto, con un comentario, en caso de que no pase la regla, sobre por qué no la pasa. Toda la información está sacada del informe de JavaCheck, sin haber revisado el código.

Reglas de estilo		
Regla	**Se Cumple**	**Comentario**
Los ficheros tienen una única clase o interfaz pública	Ok	Pasada.
Los ficheros tienen un tamaño máximo de 2000 líneas.	Ok	Pasada.
Los atributos que no sean constantes (estáticos y finales) deben empezar por minúscula en *camel case*.	Error	Escrito en mayúscula todos los atributos estáticos, aunque no fueran finales. Y por el nombre se considera que, tal vez, alguno sí que debería haber sido final.
Los atributos que son constantes (estáticos y finales) deben estar escritos en mayúsculas.	Error	Solo los atributos que hay se llaman serialVersionUID son constantes, y son generados por la interfaz Serializable automáticamente.
Todos los atributos deben estar inicializados en su declaración.	Error	Algunos lo están, otros no.
El nombre de todos los métodos, que no sean constructores, debe empezar por minúscula en *camel case*.	Error	Algunos lo están, otros no.
El tamaño máximo de un método son 30 líneas.	Error	Algunos métodos tienen un tamaño de hasta 122 líneas.
Todos los métodos tienen que tener un comentario de documentación con las etiquetas @parameter, @return, @throws.	Error	Ningún método lo pasa, no tienen comentarios de documentación o no tienen todas las etiquetas.
Todos los enumerados deben empezar por mayúscula en *camel case*.	Ok	Pasada.
Todos los enumerados tienen que tener un comentario de documentación con la etiqueta @author.	Error	Solo hay un enumerado y o no tiene comentarios de documentación o no tiene la etiqueta.
Todas las clases deben empezar por mayúscula en *camel case*.	Ok	Pasada.
Todas las clases tienen que tener un comentario de documentación con la etiqueta @author.	Error	Solo hay una clase que lo pase, el resto o no tienen comentarios de documentación o no tiene la etiqueta.
Todas las interfaces deben empezar por mayúscula en *camel case* y su nombre empezar por "I".	Error	Solo hay una interfaz, y si está en mayúscula en *camel case*, pero no empieza por "I".
Todas las interfaces tienen que tener un comentario de documentación con la etiqueta @author.	Error	Solo hay una interfaz y o no tiene comentarios de documentación o no tiene la etiqueta.
No debe existir el paquete por defecto de eclipse, es decir todos los elementos deben estar en un paquete.	OK	Pasada.

Tabla 6.1: Resultados de las reglas de estilo en el primer proyecto.

Reglas de Programación		
Regla	**Se Cumple**	**Comentario**
Todas las clases abstractas son heredadas.	Error	Algunas clases abstractas, por ejemplo *Load* o *Notifications*, no son heredadas.
Todas las interfaces son heredadas por otra interfaz o implementadas.	OK	Pasado.
Todas las clases que implementen la interfaz *Comparable* debe sobrescribir los métodos *equals* y *hashCode.*	OK	Ninguna clase implementa comparable.
Ningún método puede tener como tipo de retorno la clase *Object.*	Error	Hay 7 métodos que devuelve *Object.*

Tabla 6.2: Resultados de las reglas de programación en el primer proyecto.

Reglas de Programación		
Regla	**Se Cumple**	**Comentario**
Hay una clase con nombre parecido a *Item* que es abstracta y publica que es heredada entre 3 y 4 veces y tiene:	Error	Existe la clase que es heredad 2 o 4 veces, pero no tiene atributos de tipo Date.
Un atributo que se llama id cuyo tipo es *int* o *long*.	Ok	Pasado.
Un atributo que es un sinónimo de *description* o de *name* y cuyo tipo es *String*.	Ok	Pasado.
Un atributo que contiene la palabra *year* o date de tipo *String*.	Ok	Pasado.
Atributos de tipo *double*.	Ok	Pasado.
Atributos de tipo Date.	Error	No hay atributos de tipo date.
Hay una clase que hereda de *Item* que se llama *Small* y que tiene un atributo de tipo *double.*	Ok	Pasado.
Hay una clase que hereda de *Item* que se llama *Bulky* y que tiene un atributo de tipo *double.*	Ok	Pasado.
Hay una clase que hereda de *Item* que se llama *WorkOfArts* y que tiene un atributo de tipo *String* que contiene la palabra *author* en el nombre y un atributo de tipo *boolean* que contiene la palabra *certificate.*	Ok	Pasado.
Hay una clase que contiene una colección de *Items* y que tiene un método constructor con 4 parámetros cuyos tipos son: *double*, *String*, *String*, *Date*.	Ok	Pasado.

Tabla 6.3: Resultados de las reglas de especificas al proyecto.

De este experimento podemos concluir que JavaCheck es capaz de encontrar problemas en proyectos reales.

6.3 Escalabilidad de JavaCheck

Para comprobar la escalabilidad de JavaCheck se van a ejecutar las reglas de estilo y de programación del apartado anterior sobre el proyecto *org.eclipse.jdt.core* de la librería *org.eclipse.jdt.core* de Eclipse. Este proyecto tiene 1.442 clases, 238 interfaces, 17 enumerados, 24.290 métodos, 12.709 atributos, 1.443 ficheros y 62 paquetes.

En comprobar las reglas mencionadas anteriormente sobre el proyecto, JavaCheck ha tardado 362.305 milisegundos que son, aproximadamente, 6 minutos.

Aunque el proyecto es grande no es un tiempo excesivamente elevado de análisis, lo que hace que JavaCheck sea completamente apto para ejecutar en grandes proyectos. También es importante destacar que, cuantas más variables se pongan en la cláusula *using*, más se lenta será la ejecución de la regla. En cambio, las variables de la cláusula *in* apenas influyen en el tiempo de ejecución de una única regla y pueden resultar útiles para proporcionar claridad. Para demostrarlo, vamos a realizar una comparativa de tres reglas que tienen los mismos resultados pero están expresados de diferentes maneras. Una poniendo variables en la cláusula *in*, otra poniendo las variables en la cláusula *using* y por último, sin uso de variables. Las reglas son:

Con una variable en la cláusula *in*:

```
3  ClasesA: Class satisfy name start 'a';
4
5  all Class in ClasesA satisfy name end "n";
```

Figura 6.15. Regla con una variable en la cláusula *in*.

Con una variable en la cláusula *using:*

```
3  ClasesA: Class satisfy name start 'a';
4
5  all Class using ClasesA which name= ClasesA.name satisfy name end "n";
```

Figura 6.16. Regla con una variable en la cláusula *using*.

Con un filtro, pero sin variables:

```
28 all Class which name start 'a' satisfy name end "n";
```

Figura 6.17. Regla con un filtro en lugar de variables.

Con dos variables en la cláusula *in*:

```
3  ClasesA: Class satisfy name start 'a';
4  ClasesB: Class satisfy name contain 'b';
5
6  all Class in ClasesA, ClasesB satisfy name end "n";
```

Figura 6.18. Regla con dos variables en la cláusula *in*.

Con dos variables en la cláusula *using:*

```
ClasesA: Class satisfy name start 'a';
ClasesB: Class satisfy name contain 'b';

all Class using ClasesA, ClasesB which name= ClasesA.name and name=ClasesB.name satisfy name end "n";
```

Figura 6.19. Regla con dos variables en la cláusula *using*.

Con dos filtros, pero sin variables:

```
28 all Class which name start 'a' and name contain 'b' satisfy name end "n";
```

Figura 6.20. Regla con dos filtros en lugar de variables.

Con tres variables en la cláusula *in*:

```
3 ClasesA: Class satisfy name start 'a';
4 ClasesB: Class satisfy name contain 'b';
5 ClasesC: Class satisfy name contain 'c';
6
7 all Class in ClasesA, ClasesB, ClasesC satisfy name end "n";
```

Figura 6.21. Regla con tres variables en la cláusula *in*.

Con tres variables en la cláusula *using:*

```
ClasesA: Class satisfy name start 'a';
ClasesB: Class satisfy name contain 'b';
ClasesC: Class satisfy name contain 'c';

all Class using ClasesA, ClasesB, ClasesC which name= ClasesA.name and name=ClasesB.name and name=ClasesC.name satisfy name end "n";
```

Figura 6.22. Regla con tres variables en la cláusula *using*.

Con tres filtros, pero sin variables:

```
29 all Class which name start 'a' and name contain 'b' and name contain 'c' satisfy name end "n";
```

Figura 6.23. Regla con tres filtros en lugar de variables.

El tiempo que tarda JavaCheck en generar el AST es 10.947 milisegundos para el proyecto anterior.

	Tiempo por regla, medido en milisegundos.		
Nº Variables	In	Using	Sin variables
1	46	2.617	39
2	75	168.563	47
3	150	829.685	54

Tabla 6.4: Medida del tiempo de ejecución en función del uso de las variables.

Como se puede ver, el tiempo de ejecución de las sentencias que usan las variables en la cláusula *in* o las que no usan variables apenas es significativo en comparación con el tiempo que se tarda en construir el AST. Mientras que el tiempo que tarda en ejecutar las reglas que usan variables en la cláusula *using* es considerable y crece de forma exponencial a medida que se añaden variables. No obstante, usando una sola variable es un tiempo que podemos considerar aceptable, en caso de necesidad, en comparación con el tiempo que tarda en generar el AST. También cabe destacar que, el tiempo que tarda en ejecutarse la regla con tres variables en la cláusula *in* es tres veces mayor que

su equivalente sin variables. En cambio, si vamos a usar varias veces las variables sale más rentable en cuestión de tiempo el declarar las variables y luego usarlas en las clausulas *in* que tener que filtrar todas las veces el valor. En la tabla 6.2 se muestra esto ejecutando 4, 8 y 12 veces las mismas reglas, en un lado usando tres variables en la cláusula *in* y en el otro su equivalente sin usar variables, es decir, filtrando los valores.

Veces que se ejecutan las sentencias	Tiempo medido en milisegundos	
	Sin variables	In
4	156	151
8	275	152
12	411	155

Tabla 6.5: Medida del tiempo de ejecución en función del número de sentencias que se ejecutan.

7 Conclusiones y trabajo futuro

7.1 Conclusiones

El objetivo de este proyecto era el desarrollar un lenguaje de dominio específico para el análisis estático de código Java. Para ello ha sido elaborado JavaCheck.

Con JavaCheck se puede expresar una amplia gama de reglas. No obstante, se puede ampliar con un gran número de propiedades del código Java. Aun así, ya se ha probado que es útil para encontrar problemas en proyectos reales.

También se ha visto que en proyectos muy grandes el tiempo de ejecución no es excesivo, con la excepción del uso de variables en la cláusula using.

En el capítulo 2 se ven herramientas para el análisis estático. JavaCheck añade, dentro de ese campo, que no es un analizador que se centra en buscar errores, sino que busca propiedades o patrones que un proyecto deba seguir y que además, no sigue estándares, sino que las reglas las puedes programar de manera sencilla, con las necesidades que se precisen para cada proyecto.

7.2 Trabajo futuro

Una evidente línea de trabajo para este proyecto es la ampliación de las propiedades. Algunas de las que se podrían añadir son:

- Comprobar la posición e inicialización de las variables dentro de un método.
- Comprobar si un método está sobrescribiendo a otro.
- El número de bucles anidados dentro de un método.
- Comprobar si un método es recursivo.
- Comprobar la declaración de anotaciones
- Comprobar el uso de anotaciones.

También se ve la necesidad de mejorar el rendimiento del uso de variables en la cláusula using.

Por último, como *plugin* de Eclipse hay algunos elementos que estaría bien añadir. Como por ejemplo:

- Un asistente para crear proyectos específicos de JavaCheck.
- Un mecanismo para añadir, de forma sencilla y lo más abstracta posible, nuevas propiedades de los elementos.
- Un mecanismo para reportar errores, que sea sencillo de leer y que contenga toda la información necesaria.

Referencias

[1] Robert C. Martin “Clean Code: A Handbook of Agile Software Craftsmanship”, Julio 2008.

[2] Coding Standard List [Último acceso: 23/01/2017] URL: http://c2.com/cgi/wiki?CodingStandardList

[3] Sun Code Conventions [Último acceso: 23/01/2017] URL: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

[4] DougLea's Coding Standards [Último acceso: 23/01/2017] URL: http://gee.cs.oswego.edu/dl/html/javaCodingStd.html

[5] Isabel Lamas Codesido, “Comparación de analizadores estáticos para código java. Proyecto de investigación básica o aplicada.” Enero 2011, 19 páginas.

[6] PMD [Último acceso: 23/01/2017] URL: https://pmd.github.io/.

[7] “PMD Applied”, November 2005

[8] Ian F. Darwin “Checking Java Programs”, March 2007

[9] FindBugs [Último acceso: 23/01/2017] URL: http://findbugs.sourceforge.net/

[10] FreeBSD License [Último acceso: 23/01/2017] URL: https://www.freebsd.org/copyright/freebsd-license.html

[11] GNU Public License [Último acceso: 23/01/2017] URL: https://www.gnu.org/licenses/gpl-3.0.html

[12] Simian [Último acceso: 23/01/2017] URL: http://www.harukizaemon.com/simian/

[13] RedHill Consultoring [Último acceso: 23/01/2017] URL: http://www.redhill-consulting.com/

[14] CheckStyle [Último acceso: 23/01/2017] URL: http://checkstyle.sourceforge.net/

[15] SonaQube [Último acceso: 23/01/2017] URL: http://www.sonarqube.org/

[16] Semmle [Último acceso : 23/01/2017] URL: https://semmle.com/

[17] SemmleQL [Último acceso: 23/01/2017] URL: https://semmle.com/products/semmle-ql/

[18] Datalog Wikipedia [Último acceso: 23/01/2017] URL: https://en.wikipedia.org/wiki/Datalog

[19] Datalog [Último acceso: 23/01/2017] URL: http://www.learndatalogtoday.org/

[20] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor “CodeQuest: Scalable Source Code Queries with Datalog” Programming Tools Group, Oxford University Computing Laboratory, 2006

[21] Atkinson, C; Kühne, T “Model-Driven Development: A Metamodeling Foundation”, IEEE Software, pp. 36-41, 2003.

[22] González-Pérez, C; Henderson-Seller, B “Metamodeling for Software Engineering” John Wiley & Sons, 2008.

[23] Jesús García Molina, Felix O. Gracía Rubio, Vicente Pelechano, Antonio Vallecillos, Juan Manuel Vara, Cristina Vicente-Chicote “Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas” RA-MA Editorial, 2013.

[24] Francisco Durán Muñoz, Javier Troya Castilla, Antonio Vallecillo Moreno "Desarrollo de Software Dirigido por Modelos" FUOC. Fundación para la Universitat Oberta de Catalunya,

[25] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose "Eclipse Modeling Framework", Addison Wesley, 2003.

[26] Eclipse Modeling Framework [Último Acceso: 23/01/2017] URL: https://www.eclipse.org/modeling/emf/

[27] Eclipse [Último Acceso: 23/01/2017] URL: http://download.eclipse.org/modeling/emf/emf/javadoc/xsd/2.9.0/

[28] Xtend [Último Acceso: 23/01/2017] URL: https://www.eclipse.org/xtend/index.html

[29] Articulo AST Eclipse [Último Acceso: 23/01/2017] URL: http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html

[30] Help Eclipse AST [Último Acceso: 23/01/2017] URL: http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2Fpackage-summary.html

[31] ISO/IEC 14977:1996 - Information technology - Syntactic metalanguage - Extended BNF [Último Acceso: 23/01/2017] URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:14977:ed-1:v1:en

[32] Brian W. Kernighan, P. J. Plauger. "The elements of Programming style" McGraw-Hill, 2ª Edición, 1978

[33] Allan Vermeulen,Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur. "The Elements of Java(TM) Style". Cambridge University Press; Reprint edition (January 28, 2000)

[34] Verilog [Último Acceso: 23/01/2017] URL: http://www.verilog.com/

[35] UML [Último Acceso: 23/01/2017] URL: http://www.uml.org/

[36] Laurent Debrauwer, "Patrones de diseño en Java. Los 23 modelos de diseño: descripción y soluciones ilustradas en UML 2 y Java" Editorial ENI, 2013.

Glosario

API	Application Programming Interface
AST	Abstract Syntax Tree
DOM	Document Object Model
DSL	Domain Specific Modeling Language
EMF	Eclipse Modeling Framework
GPL	General Purpose Modeling Languages
IDE	Integrated Development Environment
JDT	Java Development Tools
MDD	Model Driven Development
UML	Unified Modeling Language
XML	Extensible Markup Language

Anexos

A. Meta-Modelo Completo

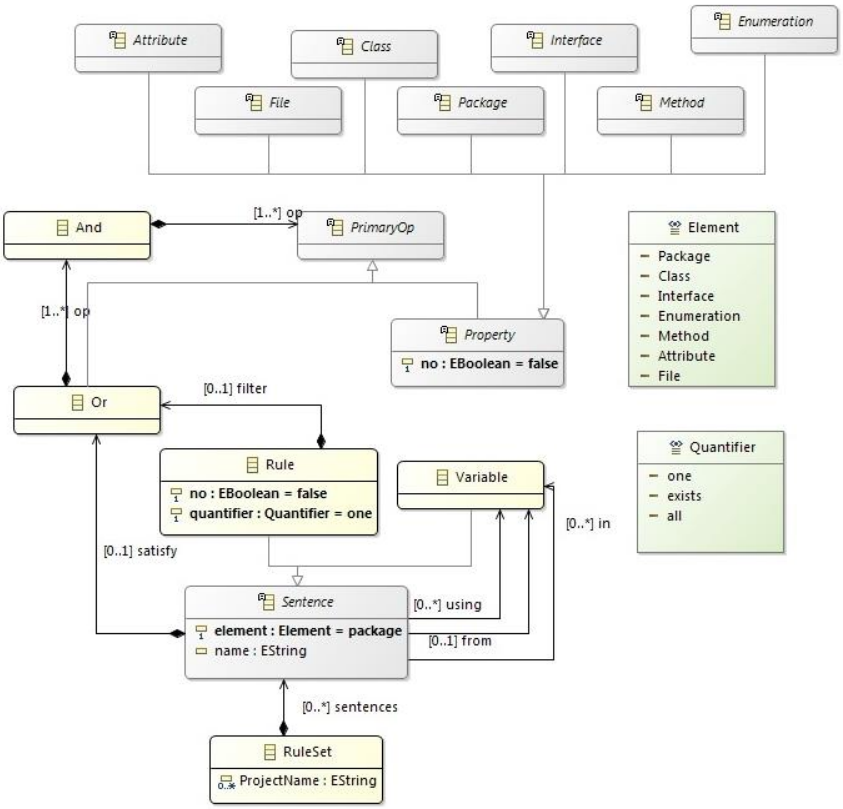

Figura A.1. Meta-modelo de JavaCheck.

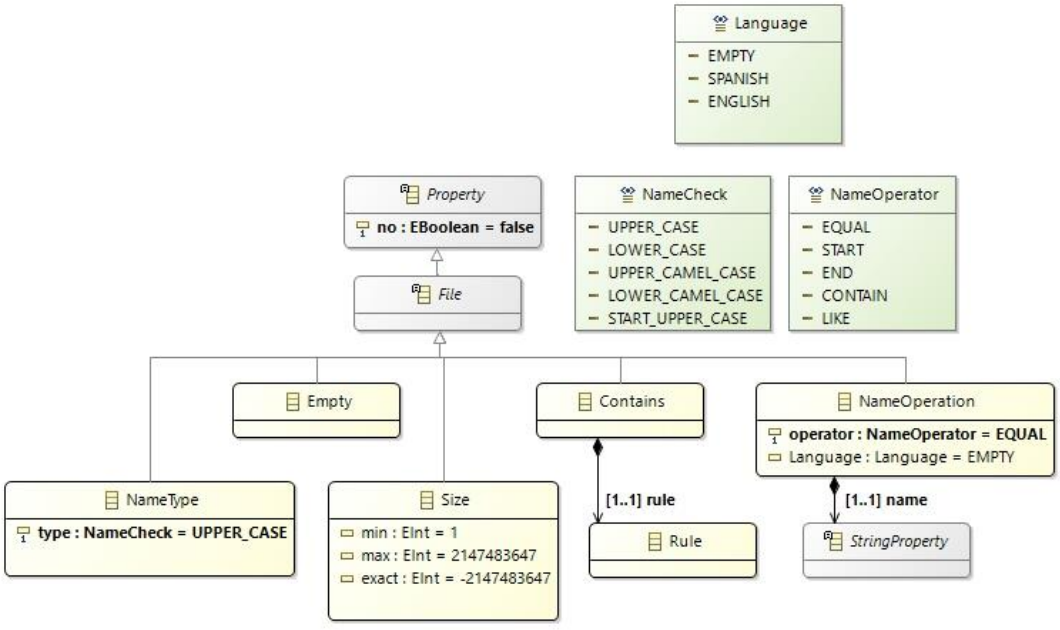

Figura A.2. Parte del meta-modelo: propiedades que heredan de *File*

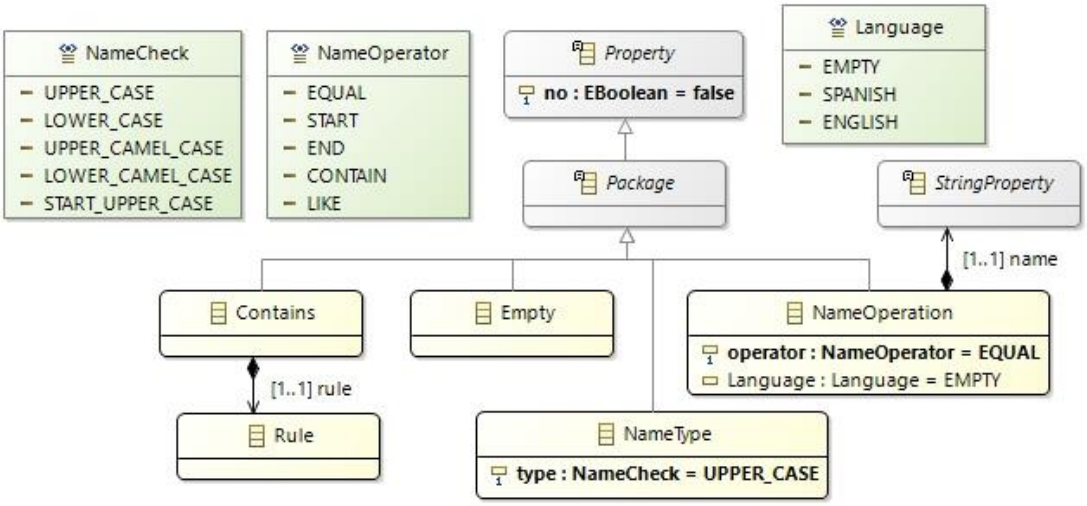

Figura A.3. Parte del meta-modelo: propiedades que heredan de *Package*

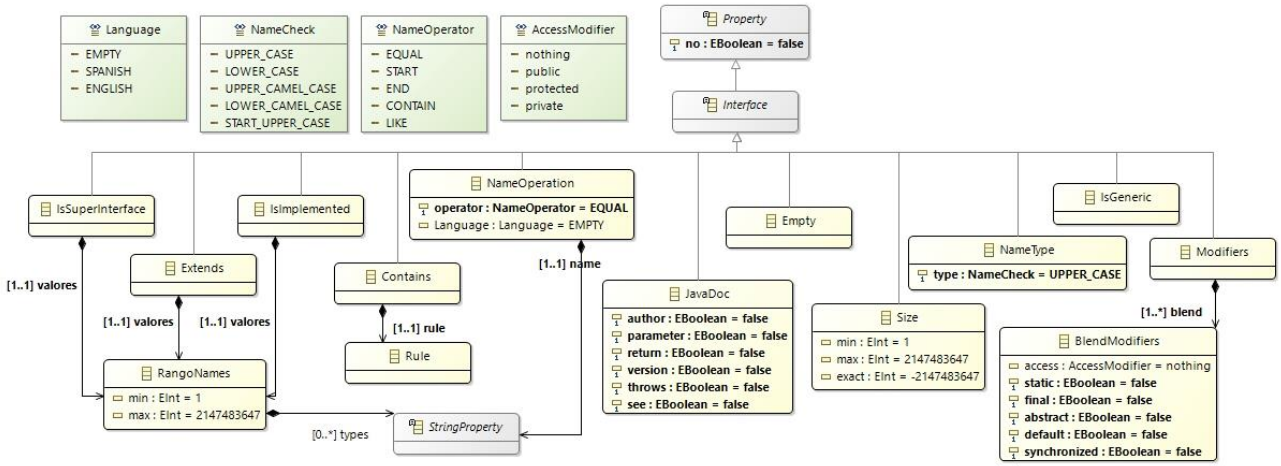

Figura A.4. Parte del meta-modelo: propiedades que heredan de *Interface*

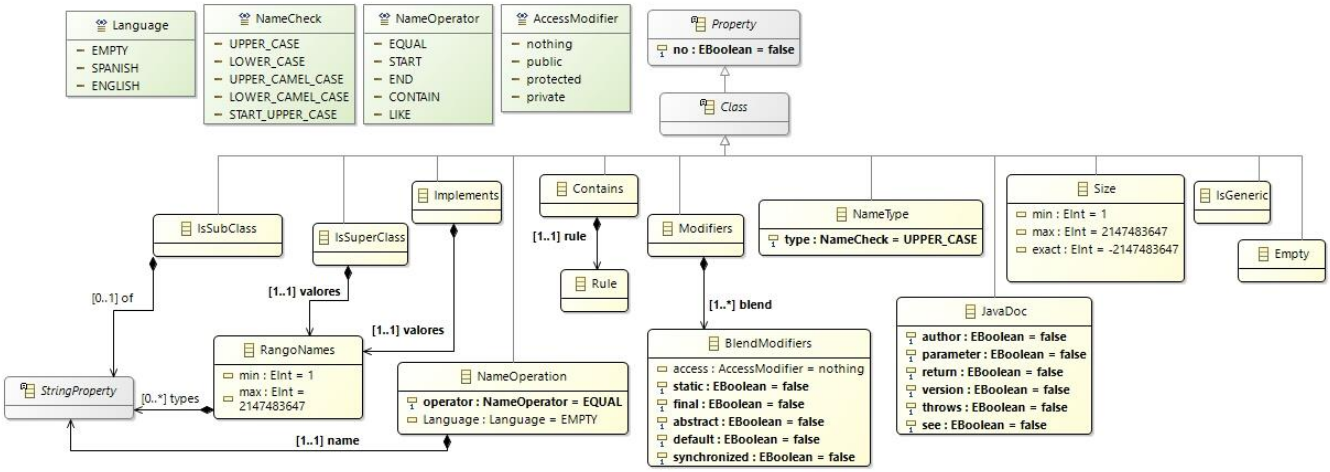

Figura A.5. Parte del meta-modelo: propiedades que heredan de *Class*

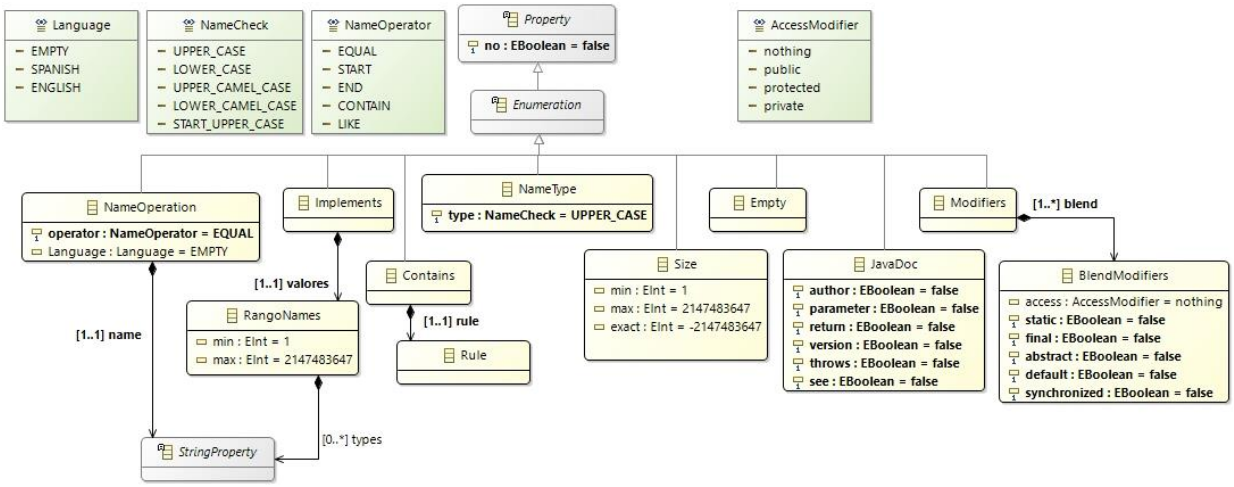

Figura A.6. Parte del meta-modelo: propiedades que heredan de *Enumeration*

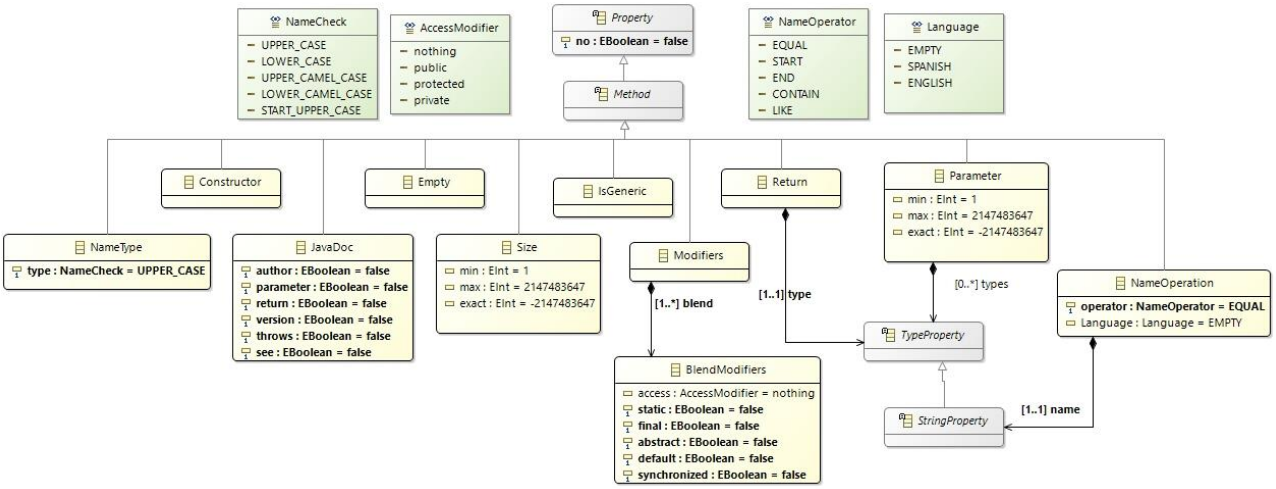

Figura A.7. Parte del meta-modelo: propiedades que heredan de *Method*

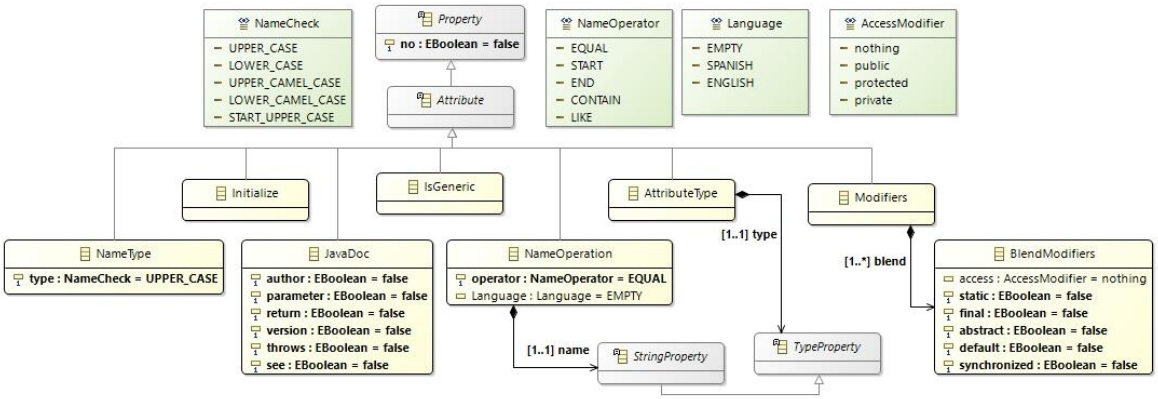

Figura A.8. Parte del meta-modelo: propiedades que heredan de *Attributes*

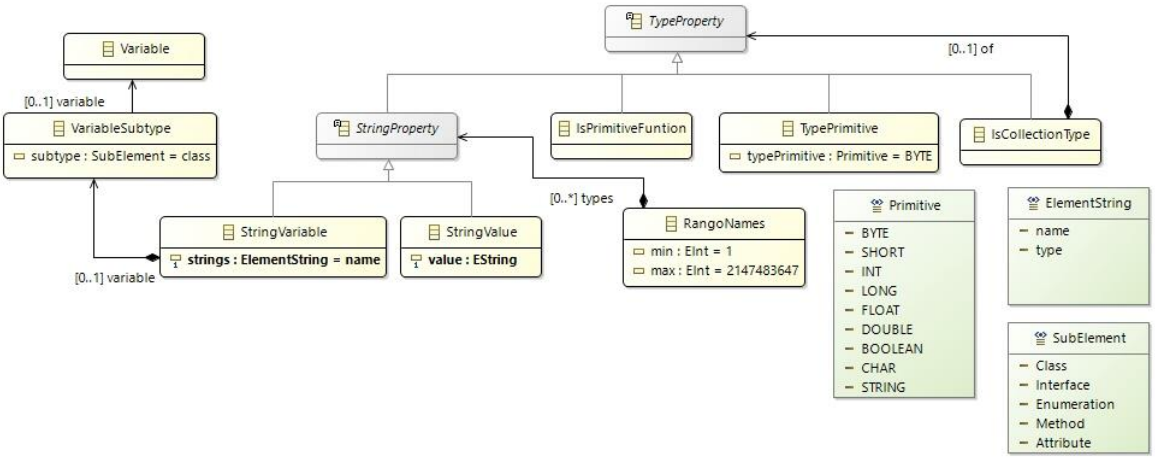

Figura A.9. Parte del meta-modelo: elementos usados por las propiedades

B. Sintaxis concreta completa

<RuleSet> **::=** '**Projects Name :**'((<STRING> ('**,**' <STRING>)*) | '*') <Sentence> (<Sentence>)* ;
<Sentence> **::=** (<Rule> | <Variable>) ';' ;
<Variable> **::=** <VariableName>':'<Element> ('**from**' <VariableName>)?
('**in**' <VariableName> ('**,**' <VariableName>)*)? ('satisfy' <Or>);
<Rule> **::=** ('**no'**)? <Quantifiers> <Element> ('**from**' <VariableName>)?
('**in**' <VariableName> ('**,**' <VariableName>)*)? ('**which**' <Or>)('**satisfy**' <Or>);
<Quantifiers> **::=** '**all**'
| '**one**'
| '**exist**';
<Element> **::=** '**File**'
| '**Package**'
| '**Class**'
| '**Interface**'
| '**Enumeration**'
| '**Method**'
| '**Attribute**';
<Or> **::=** <And> ('**or**' <And>)*;
<And> **::=** <PrimaryOp> ('**and**' <PrimaryOp>)*;
<PrimaryOp> **::=** '**(**' <Or> '**)**'
| <Property>;
<Property> **::=** <File>
| <Package>
| <Class>
| <Interface>
| <Enumeration>
| <Method>
| <Attribute>;
<File> **::=** <Size>
| <Empty>
| <NameType>
| <NameOperation>
| <Contains>;

<Package> **::=** <NameOperation>
| <Contains>
| <Empty>
| <NameType>;
<Class> **::=** <IsGeneric>
| <JavaDoc>
| <Size>
| <Implements>
| <NameOperation>
| <Contains>
| <NameType>
| <Modifiers>
| <IsSuperClass>
| <Empty>

| <IsSubClass>;
<Interface> ::= <IsImplemented>
| <NameType>
| <IsSuperInterface>
| <Size>
| <Modifiers>
| <Empty>
| <IsGeneric>
| <Extends>
| <JavaDoc>
| <Contains>
| <NameOperation>;
<Enumeration> ::= <Size>
| <Contains>
| <JavaDoc>
| <Empty>
| <NameOperation>
| <NameType>
| <Modifiers>
| <Implements>;
<Method> ::= <NameOperation>
| <Size>
| <IsGeneric>
| <Modifiers>
| <Return>
| <Constructor>
| <Parameter>
| <NameType>
| <JavaDoc>
| <Empty>;
<Attribute> ::= <Modifiers>
| <Initialize>
| <AttributeType>
| <JavaDoc>
| <IsGeneric>
| <NameType>
| <NameOperation>;
<IsImplemented> ::= '**is**' ('**not**')? '**implemented**' <Valores>;
<Valores> ::= ('**[**'<INT>'**..**'(<INT> |'*')'**]**')? ('**types{**'<StringProperty>('**,**' <StringProperty>)* '**}**')?;
<IsSuperInterface> ::= '**is**' ('**not**')? '**superinterface**' <Valores>;
<IsSuperClass> ::= '**is**' ('**not**')? '**superclass**' <Valores>;
<IsSubClass> ::= '**is**' ('**not**')? '**subclass**' '**of**' <StringProperty>;
<Implements> ::= '**don't**' '**implements**' <Valores>;
<Extends> ::= '**don't**' '**extends**' <Valores>;
<Size> ::= '**size**' (('='|'<>') <INT>| [<INT> '**..**' (<INT>| '*****')]);
<Parameter> ::= '**parameters**' ('**size**' (('='|'<>') <INT>| [<INT> '**..**' (<INT>| '*****')]))?
('**types**' '=' '**[**' <TypeProperty> ('**,**' <TypeProperty>)* '**]**')?;
<Constructor> ::= '**is**' ('**not**')? '**constructor**';
<Return> ::= '**return**' '**type**' ('=' | '<>') <TypeProperty>;
<AttributeType> ::= '**type**' ('=' | '<>') <TypeProperty>;
<Initialize> ::= '**is**' ('**not**')? '**initialize**';
<Empty> ::= '**is**' ('**not**')? '**empty**';

<IsGeneric> **::=** ‘**is**’ (‘**not**’)? ‘**generic**’;
<NameOperation> **::=** ‘**name**’ <Operator> <StringProperty> (‘**,**’ (‘**English**’| ‘**Spanish**’))?;
<Operator> **::=** ‘=’|’<>’| ‘**contains**’ | ‘**start**’ | ‘**end**’ | ‘**like**’;
<NameType> **::=** ‘**name**’ ‘**type**’ (‘=’| ‘<>’) <NType>;
<NType> **::=** ‘**upper case**’ | ‘**lower case**’ | ‘**upper camel case**’ | ‘**lower camel case**’ | ‘**start upper case**’;
<JavaDoc> **::=** (‘**no**’)? ‘**JavaDoc**’
(‘**@author**’)?(‘**@parameter**’)?(‘**@return**’)?(‘**@version**’)?(‘**@throws**’)?(‘**@see**’)?;
<Contains> **::=** (‘**have**’ | ‘**haven’t**’) ‘**{**’ <Rule>‘**}**’;
<Modifiers> **::=** ‘**is**’ (‘**not**’)? ‘**modified with**’ ‘**[**’ <BlendModifiers> (‘**or**’ <BlendModifiers>)‘**]**’
<BlendModifiers> **::=** <AccessModifier> ('**and static**')? ('**and final**')? ('**and abstract**')? ('**and default**')?
| ‘**static**' ('**and final**')? ('**and abstract**')? ('**and default**')?
| ‘**final**' (('**and abstract**')? ('**and default**')?
| ‘**abstract**' ('**and default**')?
| ‘**default**';
<AccessModifier> **::=** ‘**public**’ | ‘**protected**’ | ‘**private**’;
<TypeProperty> **::=** <StringProperty>
| ‘**isPrimitiva()**’
| ‘**isCollection(**’ (<TypeProperty>)?‘**)**’
| <TypePrimitive>;
<TypePrimitive> **::=** ‘**Primitive.**’<Primitives>;
<Primitives> **::=** '**boolean**' | '**byte**' | '**char**' | '**double**' | '**float**| '**int**' | '**long**'| '**short**' | '**String**';
<StringProperty> **::=** <STRING>
| <VariableName> ‘**.**’(<SubElement> ‘**.**’)? (‘**name**’| ‘**type**’);
<SubElement> **::=** ‘**Class**’
| ‘**Interface**’
| ‘**Enumeration**’
| ‘**Method**’
| ‘**Attribute**’;

C. Restricciones del lenguaje

Para el correcto funcionamiento del lenguaje es necesario crear algunas restricciones.

Restricciones generales

- Los proyectos que se quieran analizar deben estar en el *workspace*.

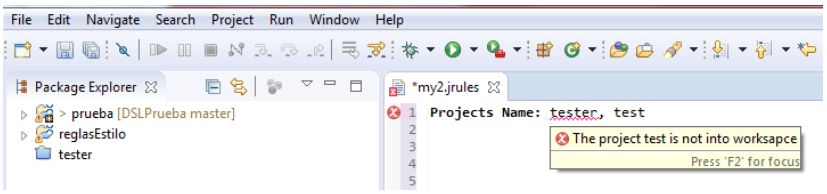

Figura C.1. Restricción 1: Los proyectos que se quieran analizar deben estar en el *workspace*.

- El tipo de todas las propiedades de una sentencia deben ir acorde con elemento, es decir, no se puede poner una propiedad que hereda de *Method* con un elemento que sea *Package*.

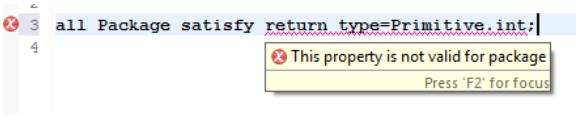

Figura C.2. Restricción 2: La propiedad no concuerda con el elemento.

- Cuando se vaya a usar una variable en una propiedad es necesario que la variable esté puesta en el campo *using*.

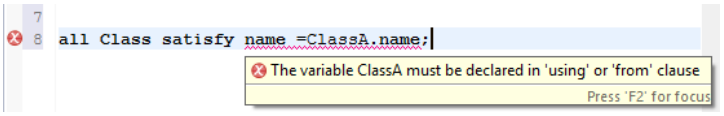

Figura C.3. Restricción 3: La variable *ClassA* debe declararse en la variable *using*.

- No se puede poner en la cláusula *from* una variable cuyo elemento no contenga el elemento de la sentencia. Es decir, no se puede poner una variable que sea de *Attributes* en la cláusula *from* de una sentencia que tiene como elemento *Package* porque los atributos no contienen paquetes.

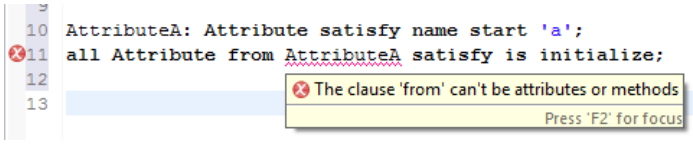

Figura C.4. Restricción 4: Los atributos y no métodos no pueden contener nada.

- La variable de la cláusula *in* de una sentencia debe ser del mismo tipo de elemento que la sentencia.

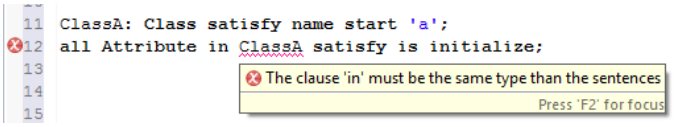

Figura C.5. Restricción 5: La variable de la cláusula *in* debe coincidir con el elemento.

- Todas las variables que se usen en una sentencia deben haberse declarado antes de la sentencia.

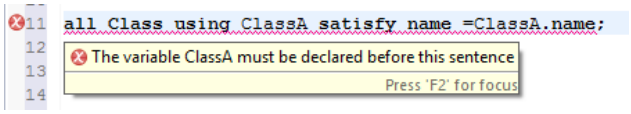

Figura C.6. Restricción 6: La variable *ClassA* debe ser declarada antes de su uso.

- No puede haber dos o más variables con el mismo nombre.

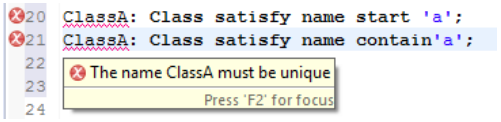

Figura C.7. Restricción 7: El nombre de variable *ClassA* debe ser único.

A continuación se mostrarán las restricciones organizados por propiedades.

NameType:

- Siempre que se use el operador *Like* debe especificarse el idioma en el que buscar sinónimos.

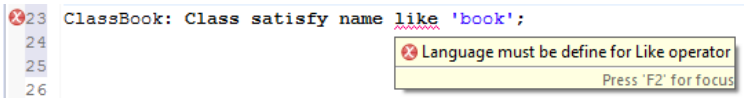

Figura C.8. Restricción 8: Para buscar sinónimos es necesario especificar el idioma.

- Si el operador no es *Like* no se debe definir idioma.

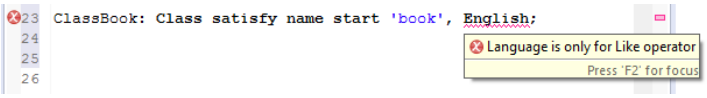

Figura C.9. Restricción 9: Para buscar sinónimos es necesario especificar el idioma.

JavaDoc:

- Las etiquetas *@return @parameters* y *@throw*s son específicas de los métodos. Es una advertencia en lugar de un error.

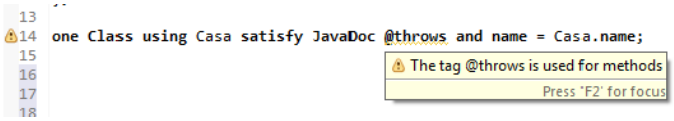

Figura C.10. Restricción 10: La etiqueta *@throws* es solo para métodos.

Modifiers:

- El modificador *abstract* es un modificador implícito en las interfaces y por ello no es necesario declararlo de forma explícita. Es una advertencia.

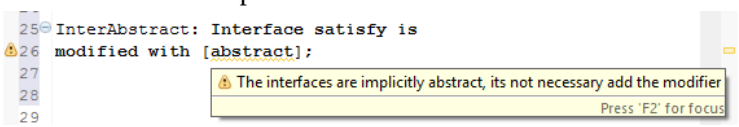

Figura C.11. Restricción 11: Las interfaces son implícitamente *abstract*.

- El modificador *abstract* se usa únicamente para clases y métodos.

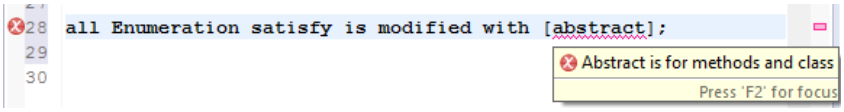

Figura C.12. Restricción 12: El modificador *abstract* se usa únicamente para clases y métodos.

- Los modificadores *abstract* y *final* no pueden estar simultáneamente en un elemento.

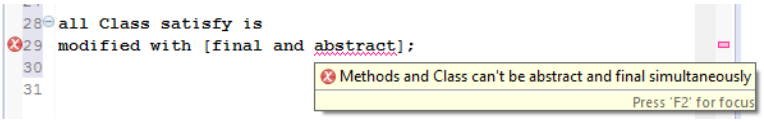

Figura C.13. Restricción 13: Los modificadores *abstract* y *final* no se pueden usar simultáneamente.

- Solo los métodos pueden usar el modificador *default*.

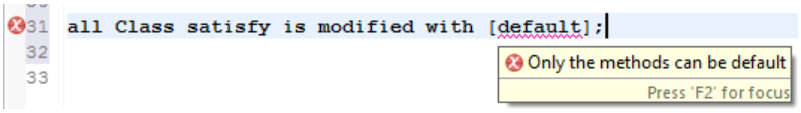

Figura C.14. Restricción 14: Solo los métodos pueden usar el modificador *default*.

- Solo las interfaces tienen métodos con modificadores *default.*

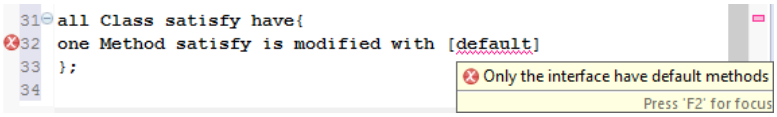

Figura C.15. Restricción 15: Solo las interfaces tienen métodos con modificadores *default.*

- El modificador final es solo para métodos, clases y atributos.

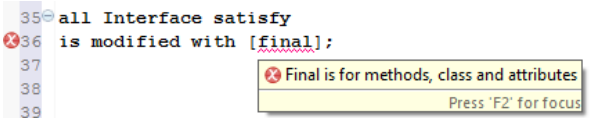

Figura C.16. Restricción 16: El modificador final es solo para métodos, clases y atributos.

- El modificador *synchronized* es solo para métodos.

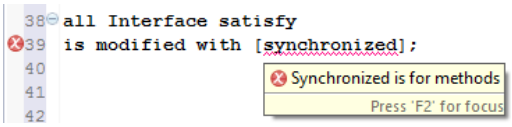

Figura C.17. Restricción 17: El modificador *synchronized* es solo para métodos.

Contains:

- Las reglas dentro *contains* deben ser de un elemento que pueda ser contenido por el elemento de la regla de la que parte.

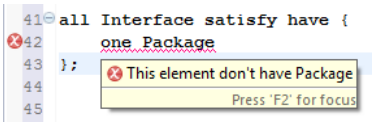

Figura C.18. Restricción 18: Las interfaces no pueden tener paquetes.