Proyecto Final

Full ANTLR support for PMD

Integrantes

52066 - Soncini, Lucas

52051 - De Lucca, Tomás

53214 - Fraga, Matías

Tutor

Sotuyo Dodero, Juan Martín

Índice

1. Abstract **2**

2. ¿Qué es PMD? **3**

3. Motivación **4**

4. Conceptos **6**

5. ¿Cómo funciona PMD? **8**

¿Cómo funciona CPD? 8

¿Cómo funcionan las reglas de PMD? 9

6. Desarrollo **11**

Análisis 11

CPD 12

PMD 14

7. Logros **20**

CPD 20

PMD 21

Contribuciones 22

8. Conclusión **23**

1. Abstract

La diversidad de lenguajes de programación se encuentra en aumento constante. Es importante poder contar con herramientas para poder verificar la calidad del código que se escribe y que ayuden a encontrar los errores más comunes fácilmente. A su vez, es importante que estas herramientas ofrezcan soporte a gran cantidad de lenguajes y que puedan mantenerse actualizadas frente a los cambios de los mismos. *PMD* es una herramienta de análisis estático de código que soluciona este problema. Sin embargo, la diversidad de lenguajes soportados actualmente es limitada.

El objetivo del siguiente proyecto fue poder incorporar soporte completo en *PMD* a nuevos lenguajes de programación.

Para aumentar la cantidad de lenguajes soportados, se decidió integrar *PMD* con *ANTLR*, una herramienta de generación de gramáticas y recorrido de los árboles *AST* generados. La misma es altamente soportada por la comunidad y tiene un gran repertorio de gramáticas ya existentes listas para ser integradas. Con el trabajo realizado en el siguiente proyecto, se ofrecen las herramientas necesarias para que, con el apoyo de la comunidad, se pueda expandir el universo de lenguajes soportados por *PMD* con un costo muy bajo de implementación. Aumentando así el impacto de *PMD* en la comunidad y reduciendo drásticamente el costo de incorporar soporte a nuevos lenguajes.

2. ¿Qué es PMD?

Antes de comenzar a detallar el proyecto desarrollado, es indispensable entender qué es *PMD*, la herramienta sobre la cual hicimos el desarrollo de este proyecto. *PMD* es una herramienta *open source* de análisis estático de código, multi-lenguaje, muy conocida en el ámbito empresarial, y usada mundialmente en el desarrollo de aplicaciones *Java* y *Android* en empresas como *Mercado Libre*. Cuenta con dos grandes funcionalidades: detección de código duplicado (*CPD*, de las siglas *copy paste detector*) y un motor de reglas sintácticas y semánticas. Dichos lenguajes, se agregan a la herramienta mediante módulos independientes que agrupan las entidades necesarias para dar soporte a cada lenguaje. Es importante destacar que un lenguaje dado no necesariamente debe soportar las dos grandes funcionalidades que *PMD* ofrece, sino que puede soportar, por ejemplo, solamente análisis de código duplicado, y posteriormente ser extendido el soporte a reglas sintácticas. Para entender esto, primero es importante que entendamos cómo funcionan y conviven internamente estas dos funcionalidades.

Por un lado, *PMD* ofrece soporte a análisis de código duplicado (conocido como *CPD*) a través de un algoritmo que analiza secuencias de *tokens* repetidas entre archivos. Para ofrecerlo, necesita de un *lexer*, entidad que para una gramática dada, permite transformar un archivo, en una cadena de *tokens*, que luego será analizada para detectar la existencia, o no, de secuencias repetidas.

Por otro lado, *PMD* también ofrece la posibilidad de evaluar conjuntos de reglas para cada lenguaje. Para poder hacerlo, es necesario que dichos lenguajes ofrezcan un *parser*, es decir, una entidad encargada de proveer el árbol sintáctico sobre el que se aplicarán las reglas definidas para cada lenguaje. Dichas reglas permiten detectar fallas en el código, vulnerabilidades, errores comunes, y marcar buenas prácticas de programación.

3. Motivación

La motivación de este proyecto, por parte de *PMD*, surge por la necesidad de la herramienta de expandir los lenguajes de programación sobre los que puede operar, lo cual acarrea varias ventajas. Por un lado, logra promover una única herramienta de análisis estático de código independiente del lenguaje. Y por el otro, permite expandir el universo de reglas sintácticas y semánticas que ofrece la herramienta, generando una abstracción sobre el uso de gramáticas y *parsers* del lenguaje. Esto último a su vez permite, de forma mucho más simple que hasta el momento, que miembros de la comunidad *open source* puedan contribuir a lograr estos objetivos sin necesidad de comprender el proyecto de *PMD* en su totalidad.

Para poder cumplir sus dos propósitos principales de forma agnóstica al lenguaje, *PMD* tiene abstracciones sobre el *AST* (árbol de sintaxis abstracto) que le permiten ofrecer diversas funcionalidades en su módulo `pmd-core` para todos ellos. Sin embargo, dicho *AST* es necesariamente dependiente de cada lenguaje en cuestión y se requiere de una herramienta que permita generar el mismo a partir de la gramática. Para ello, PMD solo soportaba una única herramienta llamada *JavaCC*, cuyo proyecto se encuentra sin soporte hace un tiempo. Ésta se encarga de proveer los *parser* y árboles sintácticos sobre los que *PMD* opera, pero debido a su falta de mantenimiento en los últimos años y el hecho de no contar con demasiados lenguajes soportados, *PMD* requería expandir la oferta de gramáticas aceptadas y abrir las puertas a otra herramienta para esto.

Aparece en este punto *ANTLR*, una herramienta *open source* que puede cumplir el mismo rol que *JavaCC*, pero cuyo proyecto es muy activo y cuenta con una gran comunidad que contribuye constantemente. Además, ofrece más de 60 gramáticas de múltiples lenguajes de programación, entidades como *parsers*, *visitors* y *listeners*, y facilidades para realizar *queries* sobre grandes volúmenes de información. Por ejemplo *Twitter* utiliza esta herramienta para realizar más de dos mil millones de *queries* por día. Debido a su comunidad así como a su gran cantidad de gramáticas ya existentes y listas para ser utilizadas, incorporar esta herramienta en *PMD* le permite saber que existirá soporte para los lenguajes más modernos y novedosos, y que de acá al futuro continuará siendo de esta manera. De esta forma, el esfuerzo a realizar nos permitiría llegar con *PMD* a diversos tipos de programadores que antes no eran alcanzados debido a la falta de soporte para gramáticas de lenguajes más nuevos por parte de *JavaCC*.

Previo al desarrollo del proyecto, *PMD* contaba con soporte únicamente para *Java* y otros 9 lenguajes de programación, por lo que nuestro objetivo se centró, en esencia, en proveer las mismas

funcionalidades que existían para estos 10 lenguajes, a un universo de lenguajes mucho más amplio. Para ello, debíamos lograr aprovechar el *runtime* de *ANTLR* y las abstracciones propias de *PMD*, para que los *lexers* y *parsers* que *ANTLR* genera sean usados de forma similar a cómo se usan los propios de *JavaCC*, permitiendo al desarrollador, mediante las abstracciones necesarias, sumar reglas en una forma idéntica a como lo hacía sobre gramáticas basadas en *JavaCC*. De esta forma, se debía dar soporte tanto a *CPD* como a la evaluación de reglas tanto con *XPath* como con el patrón *visitor*.

Por nuestra parte, consideramos que contribuir con un proyecto *open source* era indispensable para concluir el ciclo académico que comenzamos en la universidad, teniendo la posibilidad de realizar un aporte a la comunidad de desarrolladores. Creemos que realizar dicho aporte sobre una herramienta como *PMD*, que busca mejorar los estándares de calidad de los proyectos informáticos, estaba completamente alineado con los valores que rigen la carrera. Al mismo tiempo, el participar en un proyecto *open source* ya establecido y usado mundialmente como *PMD* nos ayudó a entender la dinámica de proyectos que transcienden un simple trabajo práctico y serán utilizados por varios años a venir.

Si bien inicialmente se posicionaba como un gran desafío, creemos que la dificultad de participar en un proyecto de tal envergadura también fue un factor motivante. Sabíamos que colaborar en un proyecto *open source* como *PMD* tendría repercusiones en la gran comunidad de programadores, así como también en el ámbito profesional debido a que *PMD* es usado por diversas empresas en el mundo. Coincidimos en que los aportes realizados y el tiempo invertido serían altamente retribuidos en forma de experiencia y reconocimiento.

4. Conceptos

A lo largo de varias materias de la carrera, pero por sobre todo en *Autómatas, Teoría de Lenguajes y Compiladores* (*TLA*), adquirimos diversos conceptos fundamentales para comprender las herramientas a utilizar en este proyecto. En este apartado haremos una breve descripción de aquellos conceptos que resultan de utilidad para comprender cómo funcionan *PMD*, *JavaCC* y *ANTLR*.

En primer lugar, hablaremos del concepto de *gramática*, a la cual describimos como un conjunto de símbolos terminales y no terminales, y producciones que definen cómo se pueden combinar los mismos. Al conjunto de todas las posibles cadenas definidas por una gramática se lo denomina *lenguaje*. A lo largo del proyecto encontramos tanto gramáticas de *ANTLR* como de *JavaCC* para definir, en este caso particular, los lenguajes de programación a analizar por *PMD*.

Un *lexer* es el encargado de *tokenizar* una cadena de caracteres, es decir, interpretar una cadena de símbolos y transformarla en *tokens* propios del lenguaje. Para ello debe comprender el lenguaje en cuestión y agrupar caracteres, por ejemplo, de palabras reservadas del lenguaje. Para el proceso de *CPD* es suficiente con este paso, ya que con únicamente comparar *tokens* alcanza para detectar cadenas de código repetido. Para un mayor análisis del lenguaje evaluando diversas estructuras presentes en el mismo, se requiere además de un *parser*.

Un *parser* tiene el trabajo de tomar una cadena de caracteres y generar un árbol de sintaxis que englobe la estructura de la cadena analizada, teniendo en cuenta las producciones de la gramática que definen el lenguaje en cuestión, es decir, los distintos bloques que pueden haber en el mismo. A dicho árbol de sintaxis, se lo denomina *AST* (*Abstract Syntax Tree*) y éste tiene una estructura jerárquica. A la hora de evaluar reglas en un lenguaje, importa la estructura del código a analizar, y por ello es requerido el *AST* que define el mismo.

Tanto *JavaCC* como *ANTLR* nos permiten generar automáticamente un *lexer* y un *parser* a partir de una *gramática*, pero nuevamente estos conceptos son de gran utilidad a la hora de comprender qué elementos de los provistos por las herramientas son requeridos en cada etapa del procesamiento propio de *PMD*.

A estos conceptos restaría agregar el propio de la entidad *nodo*, para lo cual debemos volver a los árboles mencionados anteriormente. Un *nodo* es una estructura de datos que apunta a su vez a otra serie de *nodos*. Un árbol jerárquico se caracteriza por tener un *nodo raíz* desde el cual se desprenden el resto de los *nodos*. Cada uno de estos *nodos* en un árbol *AST* representa un elemento representativo del lenguaje, como por ejemplo, la definición de una función, y a su vez, dicho elemento se compone de otros *nodos* o elementos como ser el nombre de la función y sus parámetros. Toda esta estructura propia del lenguaje es la representada por un *AST* en sus *nodos* durante el procesamiento de un fragmento de código propio de *PMD*.

Cabe destacar en este punto, que ambas herramientas utilizadas por *PMD* generan, además de un *lexer* y un *parser*, todos los *nodos* propios del lenguaje a partir de su gramática, para luego, al evaluar un fragmento de código, poder generar el *AST* que representa dicha cadena particular.

Ahora bien, una vez conseguido dicho *AST* ya sea por medio de *ANTLR* o bien a través de *JavaCC*, es necesaria una forma de recorrer el mismo para evaluar las reglas en cuestión, que en fín es el propósito de *PMD*. Para ello hay dos aproximaciones principales. Una de ellas es usando un patrón denominado *visitor* y la otra se basa en utilizar el patrón *listener*. *ANTLR* contaba con ambas posibilidades, siendo el *listener* su configuración default. El patrón *listener* se basa en dejar a la herramienta recorrer libremente el árbol y pedirle a la misma que avise al entrar o salir de determinado tipo de nodos; mientras un *visitor* permite definir cómo visitar a un tipo de nodos, pudiendo por ejemplo no recorrer a sus hijos si no es requerido.

5. ¿Cómo funciona PMD?

Habiendo entendido qué hace *PMD* y para qué requiere éste de una herramienta como *JavaCC* o *ANTLR*, procederemos a explicar cómo funciona *PMD*, explicando las entidades involucradas en el proceso de detección de código duplicado y ejecución de reglas de *PMD*, las cuales resultan indispensables para entender los aportes realizados en este proyecto, cabe destacar además, que gran parte de las entidades que se describirán a continuación, poseen abstracciones realizadas como parte de este proyecto, para ser reutilizadas en el futuro. Esto es posible gracias a que la mayoría de estas entidades son independientes de para qué lenguaje se utilicen.

Como mencionamos anteriormente, *PMD* tiene dos principales niveles de soporte para un lenguaje de programación. En primera instancia, puede soportarlo únicamente para detección de código duplicado. Y en una segunda etapa puede a su vez permitir evaluar el cumplimiento de conjuntos de reglas definidas para ese lenguaje. Cabe mencionar en este punto que *PMD* utiliza *XPath* (un lenguaje pensado para construir expresiones que recorren y procesan documentos *XML* similar a como una expresión regular trabaja sobre una cadena de texto) para escribir reglas sintácticas y semánticas, y provee una alternativa a la escritura de reglas que utilizan el patrón *visitor*.

¿Cómo funciona CPD?

CPD se basa en recibir un archivo propio de un lenguaje (o una serie de archivos) y detectar si hay cadenas de *tokens*, de un largo mínimo configurable, repetidas en el mismo. Para ello, debe tomar la secuencia de caracteres en el archivo fuente y transformar los mismos en un *stream* de *tokens* con significado particular para el lenguaje en cuestión. Este proceso se denomina *tokenizar* y requiere una gramática que, entre otras cuestiones, defina palabras reservadas, caracteres especiales y demás combinaciones de símbolos a tratarse como unidad.

En lo que refiere a este proyecto, *PMD* utiliza los *lexers* generados a partir de la gramática para *tokenizar* la cadena de caracteres; y, para el posterior análisis del *stream* de *tokens*, cuenta con dos entidades principales. Estas entidades son el `TokenManager` y el `TokenFilter`. El `TokenManager` se encarga de procesar el *stream* de `Tokens` y devolver el siguiente `Token` a consumir. Para esto, cada `Token` pasa por una serie de validaciones realizadas por el `TokenFilter`, como por ejemplo evitar contabilizar los *imports*. También permite la entidad `TokenFilter` filtrar porciones de código mediante comentarios `CPD-OFF` y `CPD-ON` así como filtrar cualquier otro token o secuencia de *tokens*

que se desee.

¿Cómo funcionan las reglas de PMD?

El soporte completo de la herramienta viene dado por poder evaluar el cumplimiento de conjuntos de reglas definidas implementadas por *visitors* o bien definidas por un *XPath*. Para ello, *PMD* cuenta con la mayor parte de su funcionalidad completamente abstraída del lenguaje. Simplemente hacen falta definir algunas entidades que permiten a *PMD* utilizar el *lexer*, el *parser* y el *visitor* generados a partir de la gramática.

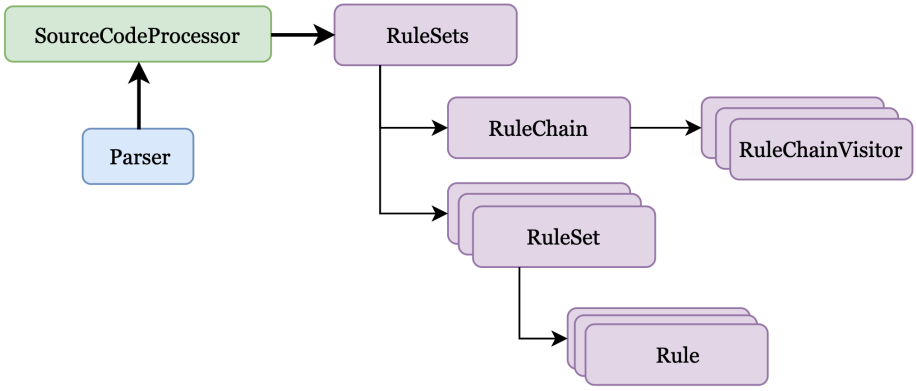

Gráfico 1: *Entidades involucradas en la evaluación de reglas.*

En el gráfico superior se pueden ver las principales entidades involucradas en el proceso de evaluación de reglas de *PMD*. Al ejecutar la herramienta con una serie de parámetros, se carga dicha configuración para generar el *AST* del archivo analizado. A partir del mismo, la entidad `SourceCodeProcessor` pide al *parser* del lenguaje el nodo raíz del *AST* para poder comenzar a evaluar las reglas. Cabe mencionar en este punto que como parte de la configuración de *PMD* se encuentran los *rulesets* a evaluar, los cuales permiten instanciar una clase `RuleSets` que no sólo contiene los distintos `RuleSet`, sino que además tiene una entidad llamada `RuleChain`. La clase `RuleChain` permite optimizar el recorrido del *AST* evitando recorrer todo el árbol de sintaxis y solo visitando aquellos nodos necesarios. Para aprovechar esta optimización, una `Rule` de *PMD* puede optar por definir esta serie de nodos o bien no hacerlo, en cuyo caso se recorrerá todo el árbol mediante un *visitor*.

Resumiendo entonces, al evaluarse un archivo, el `SourceCodeProcessor` detecta el lenguaje del

mismo y pide al `Parser` correspondiente el *root node*. Con este nodo, pide al `RuleSets` que aplique sus reglas sobre el *AST*. Al aplicar dichas reglas, primero se recolectan en la *RuleChain* los nodos indexados por las distintas reglas y luego mediante sus `RuleChainVisitor` se los visita en un mismo recorrido del *AST*. Una vez evaluadas las reglas que aprovechan la `RuleChain`, se procede a evaluar individualmente las reglas restantes, obteniendo todas las violaciones encontradas en un reporte final.

6. Desarrollo

En este apartado haremos un repaso cronológico del camino recorrido para alcanzar el objetivo del proyecto, desde los análisis iniciales, hasta las implementaciones de reglas sintácticas y semánticas que dieron por alcanzado el objetivo del proyecto.

Análisis

Al no ser un proyecto propio y tratarse de una contribución a una herramienta *open source*, a la hora de comenzar el proyecto nos encontramos con la dificultad de entender una herramienta de la magnitud de *PMD*, cuyo repositorio de código cuenta con más de 21.000 archivos. Dicho proceso no fue para nada fácil debido a la gran cantidad de módulos para cada lenguaje en particular (además de aquellos propios del *core* de la herramienta). Por lo tanto, dedicamos gran parte de la primer etapa del proyecto a comprender en profundidad cómo interactuaban todas las entidades dentro de *PMD*, qué roles y responsabilidades tenía cada una, y cómo se conectaban entre sí. Esta etapa de análisis fue crucial debido a que nuestro aporte a la herramienta no sólo consistía en agregar soporte para nuevos lenguajes a través de *ANTLR*, sino que buscábamos encapsular todas las entidades vinculadas a *PMD* en interfaces independientes, para que quienes deseen contribuir posteriormente, no requieran de este análisis, y simplemente puedan seguir estas interfaces para hacer aportes de nuevos lenguajes y reglas.

Decidimos, en primer lugar, concentrarnos en las entidades principales del proyecto, vinculadas a cómo *PMD* se conecta con las entidades específicas de cada lenguaje, para luego observar los módulos de algunos lenguajes como *Java*, a modo de ejemplo de implementación. En este proceso notamos que por la naturaleza de nuestra contribución resultaba indispensable comprender el funcionamiento de la herramienta *JavaCC*, la cual ocupaba el rol que buscamos reemplazar utilizando *ANTLR*. Era indispensable identificar qué puntos tenían en común y en qué se diferenciaban, así como también qué entidades era requerido agregar para lograr la capa de abstracción que buscábamos. Una vez que logramos comprender el proyecto, cómo funcionaba la herramienta *JavaCC* y qué entidades y funcionalidades nos proveía *ANTLR*, comenzamos a establecer conexiones entre *ANTLR* y *PMD*.

CPD

El desarrollo de la integración con *ANTLR* comenzó dando soporte a la detección de código duplicado *CPD*. En el proyecto existían módulos de lenguajes que ofrecían soporte a *CPD* con *ANTLR*, pero sus implementaciones estaban atadas al lenguaje en cuestión. Uno de estos casos era el de *Swift*. Se optó, por lo tanto, por realizar una abstracción genérica de la implementación de *Swift* para que pueda ser utilizada con las mismas entidades que las otras gramáticas generadas con *ANTLR*.

Luego de un análisis de cómo se encontraba implementada la integración con *Swift* se reconocieron las siguientes entidades principales: El *tokenizer* es la entidad encargada de interactuar con *PMD*. El mismo será el encargado de entregar una lista de *tokens* para que el motor de *CPD* analice. En nuestra implementación, el *token filter* será el encargado de proveerle estos *tokens* al *tokenizer*. El *token filter* es un punto donde se puede hacer supresión de *tokens* no deseados, y al mismo tiempo actúa como punto de extensión donde cada lenguaje puede eliminar *tokens* específicos de cada uno. El *token filter* consumirá estos *tokens* del *token manager*. El rol de este último es obtenerlos secuencialmente del *lexer* de *ANTLR* y convertirlos a un tipo que *PMD* pueda reconocer. El *lexer* recibe el código fuente y utilizando la gramática correspondiente para el lenguaje, genera una cadena de *tokens*.

Para comenzar la integración, se procedió a desacoplar lo ya implementado para la integración de *Swift*. De esta manera, se creó un `AntlrTokenizer`, que cumpliría el rol del *tokenizer* implementando la interfaz `Tokenizer` de *PMD*. Posteriormente, se creó un `AntlrTokenManager`, que cumple el rol de *token manager* e implementa la interfaz `TokenManager`. Todo lo implementado en estas dos entidades se hizo de forma genérica, pensando en que pueda ser utilizado para cualquier *lexer* generado con la herramienta que *ANTLR* nos ofrece para *Java*. Este es el punto en donde la integración con *ANTLR* se hace efectiva. Se obtiene un *token* de *ANTLR* del *lexer* y a partir de esto se construye un `AntlrToken`, una entidad que implementa la interfaz *GenericToken* y hace que pueda ser reconocido por *PMD*. Este *token* después es entregado por el *token manager* al *tokenizer* para que sean expuestos al motor de *CPD* de *PMD*.

Con esta abstracción estábamos en condiciones a utilizarla para el caso de *Swift*. Se modificó la entidad `SwiftTokenizer` que extendía el `AntlrTokenizer` genérico, y bastó asignarle el *lexer* del lenguaje auto generado por *ANTLR* para que funcione. Se aprovecharon los *unit tests* existentes para verificar que el cambio realizado no rompía la funcionalidad existente.

Avanzamos posteriormente con una funcionalidad importante *JavaCC* para el proyecto, la cual permitía filtrar *tokens*. La entidad encargada de esto se denomina *token filter*. Esta funcionalidad no existía para la implementación de *CPD* de *Swift*. Debido a esto, hubo que analizar como implementarla. Se optó por hacerla análoga a la implementación existente para gramáticas de *JavaCC*. Existía en *PMD* una clase llamada `JavaCCTokenFilter` que tenía la mayor parte de la lógica de filtrado de *tokens*. Como los *tokens* obtenidos de gramáticas de *JavaCC* y los obtenidos de *gramáticas* de *ANTLR* tenían la interfaz `GenericToken` en común, descubrimos que se podía aprovechar la misma implementación del *token filter*. De esta forma, se creó un `BaseTokenFilter` y un `AntlrTokenFilter` que extendía de él. Se movió la lógica del `JavaCCTokenFilter` al `BaseTokenFilter` haciendo pocas modificaciones, y se reescribió el `JavaCCTokenFilter` haciéndolo extender del anterior.

El *token filter* nos permite filtrar *tokens* basados en distintas reglas. Uno de los usos más comunes es poder hacer supresiones mediante comentarios. Debido a la forma genérica que estaba implementado para *JavaCC* y debido a que se pudo mantener la funcionalidad en las equivalentes de *ANTLR*, se puede ganar esta funcionalidad en cualquier lenguaje agregado a *CPD* haciendo uso del *token filter default* incluido. Esto es de extrema utilidad ya que se ganó soporte para esta funcionalidad en *Swift* sin necesidad de implementar algo específico para el lenguaje. El poder tener la lógica de filtrado en las clases base permite poder ofrecer la misma sin importar como se *tokeniza* cada uno. A su vez, el *token filter* puede ser usado también para filtrar *tokens* relacionados con cada lenguaje en particular. Para esto, basta con extender el `AntlrTokenFilter` e incluir la lógica específica de cada implementación para poder excluir *tokens* de, por ejemplo, *imports*.

Durante el tiempo que requirió desarrollar el `TokenFilter`, otro contribuidor de *PMD* aprovechó lo hecho por nosotros para *Swift* y el soporte *ANTLR* para *CPD* que desarrollamos, para dar soporte *CPD* a *Kotlin*. Por lo que una vez funcional el `TokenFilter`, realizamos un *refactor* sobre el módulo *pmd-kotlin* para que éste aproveche las ventajas de esta entidad al igual que los demás módulos de *PMD*.

Por último, y para probar que la abstracción realizada estaba a la altura de los objetivos planteados inicialmente, se decidió agregar soporte a *CPD* a un lenguaje que no existía en *PMD*: *Golang*. Para esto, se creó el `GoTokenizer` haciéndolo extender de `AntlrTokenizer`, y se incluyó la gramática de *Golang* obtenida del sitio de gramáticas de *ANTLR*. Configurando *ANTLR* para que pueda auto generar el *lexer* de *Golang* en el momento de la compilación, se creó el proyecto `pmd-go` y se lo registró con *PMD*. Se pudo completar la integración de un nuevo lenguaje para *CPD* en

menos de diez lineas de código, dando así por cumplido el objetivo.

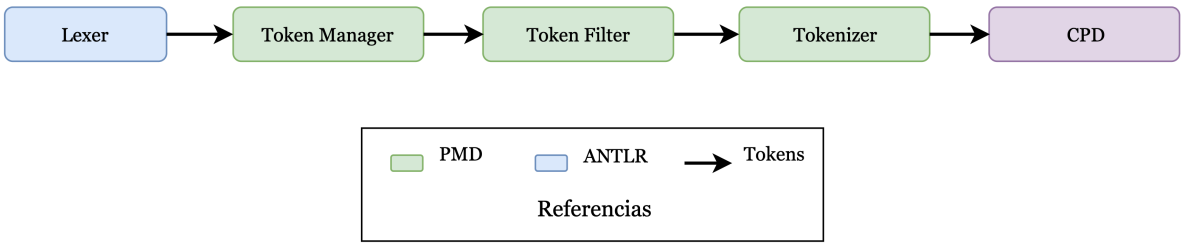

Gráfico 2: *Flujo de los tokens desde el lexer al motor de CPD.*

En el gráfico anterior se puede ver como los distintos componentes interactúan en la integración con *PMD*. Las flechas indican el sentido en que los *tokens* fluyen desde el *lexer* hasta el motor de *CPD*. Este último le pide al *tokenizer* el conjunto de *tokens* a analizar. El *tokenizer* los va obteniendo uno por uno haciéndole el pedido al *token filter* interno. Este es el encargado de pedirle los *tokens* al *token manager* que a su vez se los pide al *lexer*. El *lexer* es la pieza de *ANTLR* dentro del ciclo. El mismo fue auto generado por la herramienta integrada que nos ofrece *ANTLR*. Es importante destacar que el *token filter* en este caso decidirá cuales *tokens* entregará y cuales no. En su implementación base, este último nos permite hacer supresiones mediante comentarios. También, este es el punto de extensión a través del cual se puede incluir lógica propia de cada lenguaje.

De esta forma, para integrar un nuevo lenguaje a *CPD* basta con incluir el *lexer* auto generado por *ANTLR*. Extender las clases `AntlrTokenizer` y, opcionalmente, `AntlrTokenFilter` para incluir la poca lógica dependiente del lenguaje y por último pasarle a *PMD* el nuevo *tokenizer* para que pueda reconocer el lenguaje.

PMD

Luego de dar soporte a *CPD*, quedaba el gran desafío de dar soporte a agregar reglas de *PMD* para lenguajes integrados con *ANTLR*. Analizar cómo atacar este problema fue realmente uno de los puntos que mayor trabajo nos tomó, debido a que existían muchos interrogantes respecto a las entidades involucradas y a cómo hacer viable la interacción entre las entidades requeridas por *PMD* y las ofrecidas por *ANTLR*. Comenzamos este proceso intentando atacar el problema de la forma más genérica posible al igual que para *CPD*, buscando una solución totalmente independiente del lenguaje, siempre buscando minimizar el costo de agregar nuevos lenguajes con gramáticas de

ANTLR.

Empezamos intentando implementar un nodo genérico de *ANTLR* de la misma forma que hicimos con los *tokens*. Observamos que para *JavaCC* se extendía la implementación abstracta de la interfaz `Node` (`AbstractNode`) en cada lenguaje y para cada nodo particular del mismo. Dicho trabajo resultaba muy costoso a la hora de implementar un nuevo lenguaje. Por lo tanto, nosotros intentamos que para la implementación *ANTLR* hubiese un único nodo `AntlrBaseNode` como implementación genérica de un nodo independientemente del lenguaje. Dicho trabajo resultó difícil ya que la interfaz `Node` contaba con métodos muy atados a *JavaCC* y otros métodos cuya utilidad desconocíamos por completo en esa etapa del desarrollo. Respecto al `Parser` y los nodos, es importante marcar que `PMD` ofrece abstracciones sobre los *AST* que ya ofrecen gran parte de las funcionalidades que necesitamos, por lo que el objetivo central de conectar ambas herramientas, era lograr tener todas las funcionalidades que ofrecía el *parser* de *ANTLR* con lo que ya *PMD* nos ofrecía. Para resolver este problema, optamos por inyectarle el código necesario al `Parser` que *ANTLR* provee en runtime, modificando así la clase generada en *runtime*, ir por este camino simplificó mucho el desarrollo, debido a las limitaciones que teníamos por *Java*.

Otra de las limitaciones encontradas fue poder incluir código para satisfacer las implementaciones de las interfaces de *PMD* en los nodos auto generados por la herramienta de *ANTLR*. Los nodos generados ya extendían de la clase `ParserRuleContext` y cómo *Java* no cuenta con herencia múltiple no podíamos hacerlos extender de alguna clase base nuestra. Este fue el punto en donde se tomó la decisión de esperar a la próxima *major release* de *PMD*. *PMD 7.x.x* cuenta soporte para *Java 8* y de esta forma nos permitió incluir implementaciones de métodos *default* en las interfaces y así poder satisfacer los requerimientos de *PMD* y los de *ANTLR* en simultaneo. La estructura resultante se puede ver en el gráfico 3.

Una vez definido un nodo funcional basado en las entidades que *ANTLR* nos ofrecía, y mientras terminábamos de pulir la interfaz que creíamos mejor para los nodos, comenzamos con la implementación de un *Parser* de *PMD* utilizando un *Parser* de *ANTLR* y adaptándolo a la firma que *PMD* necesitaba que cumpla, estos primeros pasos. Para ello ya habíamos decidido implementar el soporte *PMD* completo para *Swift* como primer lenguaje debido al impacto de agregar reglas para el mismo. Aprovechando nuevamente la generalización propia de *ANTLR* y el `AntlrTokenManager` ya implementado, logramos definir un `AntlrBaseParser` genérico, que implementaba la interfaz `Parser` propia de *PMD* utilizando un *parser* de *ANTLR* y los nodos genéricos que ya habíamos implementado (`AntlrBaseNode`). A la hora de implementar dicho `Parser` para un lenguaje solo era

necesario ahora declarar el *lexer*, el *parser* y el nodo raíz de *ANTLR* correspondientes a la gramática de dicho lenguaje, lo cual ya aparentaba ser mucho más simple que lo propio para *JavaCC*.

Sin embargo, nos encontramos con otro problema bastante significativo. Si bien ya habíamos logrado auto generar tanto el *lexer* y como el *parser* para cada lenguaje a partir de la gramática *ANTLR* automáticamente, necesitábamos a su vez que en todos los *parser* auto generados, cada nodo extienda nuestro nodo genérico. *ANTLR* como herramienta provee la funcionalidad para generar un *parser*, un *lexer*, un *visitor* y un *listener* a partir de una gramática y también provee un *plugin maven* para ello, por lo que dicha parte no generaba problema salvo por su configuración para por ejemplo generar *visitors*. Al generarse los archivos automáticamente, teníamos en particular los *parsers* generados. En cada *parser* de *ANTLR* se encuentran todos los nodos propios del lenguaje definido por la gramática que se utilizó. Ahora bien, dichos nodos extienden clases de *ANTLR* y no nuestro nodo base. Por lo tanto, el inconveniente estaba en lograr que al generarse automáticamente esos archivos, los nodos definidos en ellos extiendan `AntlrBaseNode` y que `AntlrBaseNode` luego extienda las clases de *ANTLR* requeridas para aprovechar las funcionalidades implementadas en ellas. Optamos por hacer esto aprovechando los ciclos de vida de *maven* y el `plugin maven-antrun-plugin`, utilizando en particular la fase `process-sources` para procesar los archivos generados antes de utilizarlos. De esta forma, en la fase de generación de fuentes previa se generan estos archivos y luego se realizan estos cambios sobre ellos al procesar el código generado.

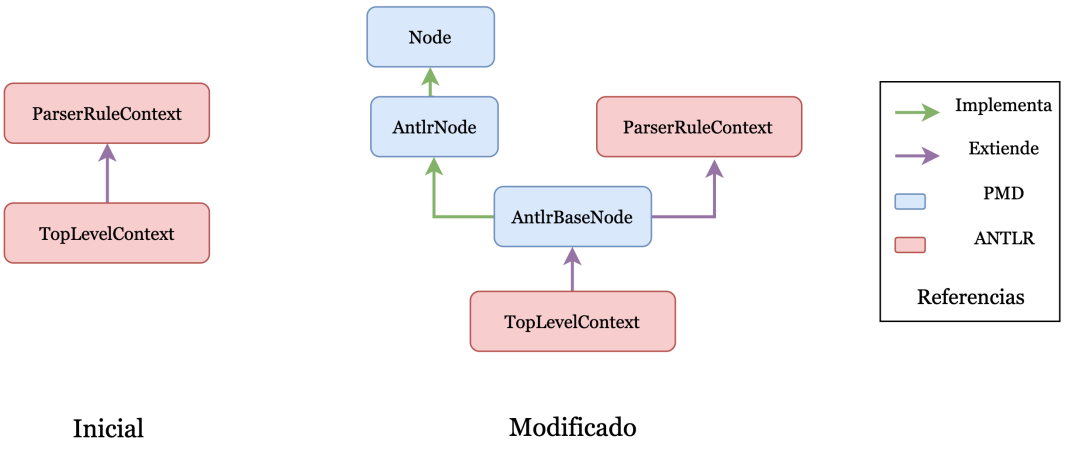

Gráfico 3: *Adaptación de los nodos de ANTLR para que puedan ser utilizados por PMD.*

El próximo paso a seguir era buscar abstraer, para todos los lenguajes que utilizan gramáticas de *ANTLR*, en lo relativo a las reglas de *PMD*. Para poder agregar reglas de nuestro lenguaje era

indispensable allanar previamente el camino generalizando distintas clases existentes en el módulo `pmd-core`. Investigamos nuevamente qué entidades se encontraban involucradas, descubriendo que las principales eran `Rule` y `RuleViolationFactory`. Todas las reglas de *PMD* heredan de una implementación abstracta de `Rule` (`AbstractRule`). `AbstractRule` es una implementación abstracta básica de todos los métodos de la interfaz `Rule` independientes del *parser*. Cada lenguaje debe implementar su clase hija de `AbstractRule` que complete la implementación definiendo cómo se visitan los nodos propios del *parser* de su gramática.

Cabe destacar en este punto que una regla de *PMD* puede ser de dos tipos. Un tipo de regla está definida por una expresión *XPath*, mientras el otro tipo se encuentra implementado mediante un *visitor* que recorre el árbol generado por el *parser* del lenguaje. Por último, `RuleViolationFactory` es una clase con el único propósito de crear violaciones propias de cada lenguaje al no cumplirse las distintas reglas.

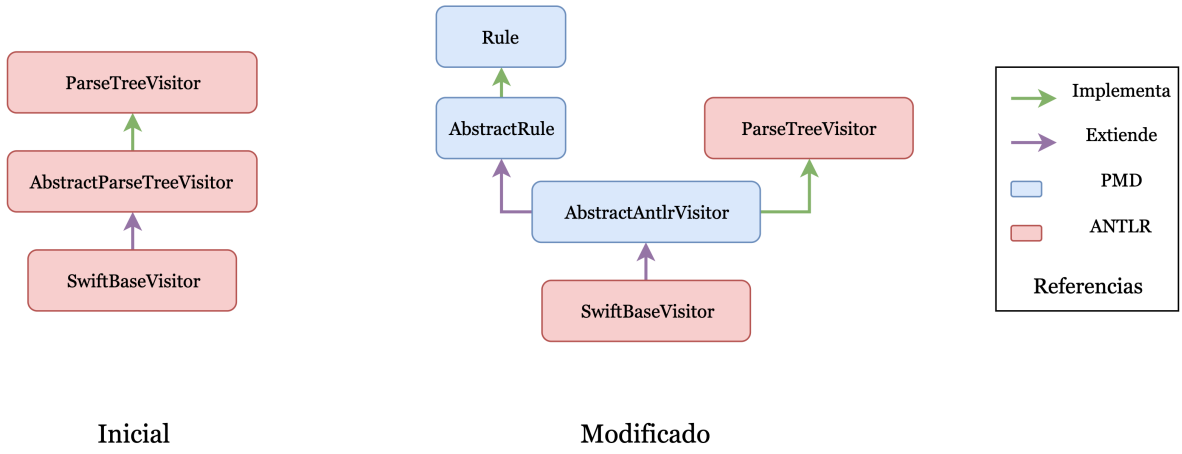

Gráfico 4: *Adaptación de los visitors de ANTLR para que puedan ser utilizados por PMD para el lenguaje de Swift.*

Con estas tres entidades en mente, y considerando como mencionamos ya reiteradas veces la abstracción propia de *ANTLR*, decidimos implementar a nivel *ANTLR* las tres entidades con tres clases nuevas: `AbstractAntlrVisitor` y `AntlrRuleViolationFactory`. La primera es una implementación genérica de `ParseTreeVisitor` (clase de *ANTLR runtime*) que extiende `AbstractRule`. La razón para implementar `ParseTreeVisitor` en esta clase era principalmente que requeríamos que los *visitors* auto generados por *ANTLR* a partir de la gramática extendieran de nuestra re-

gla genérica de *ANTLR*, es decir de `AbstractAntlrVisitor`. Ahora bien, la entidad `Visitor` de *ANTLR* ya de por sí extendía de una clase abstracta que implementaba de manera *default* la interfaz `ParseTreeVisitor`. Al no tener *Java* herencia múltiple, no podíamos heredar de ambas entidades por lo cual nos vimos forzados a implementar en nuestra clase la interfaz `ParseTreeVisitor` de *ANTLR* y así no alterar el funcionamiento normal del *visitor* auto generado. Al igual que para hacer que los nodos auto generados por *ANTLR* hereden de nuestra clase `AntlrNode`, utilizamos el *plugin* `maven-antrun-plugin` para cambiar los fuentes generados en la fase `process-sources` y hacer que el *visitor* de *ANTLR* herede de `AbstractAntlrVisitor`.

Las otras dos abstracciones (`AntlrRuleChainVisitor` y `AntlrRuleViolationFactory`) fueron simplemente implementaciones muy similares a las propias de los módulos de cada lenguaje, salvo por el detalle de estar implementadas genéricamente para cualquier lenguaje que esté definido por gramáticas *ANTLR*. A modo de prueba de concepto, agregamos también en esta instancia una implementación para *Swift*: `AbstractSwiftRule`. Cabe destacar que *Swift* ya contaba con soporte para análisis de código duplicado, pero no así para aplicarle reglas. La regla cuenta con tan sólo cuatro líneas de código en las que tan solo se define el lenguaje para la regla y el resto de la implementación es la heredada de `SwiftBaseVisitor` (*visitor* auto generado por *ANTLR* a partir de la gramática de *Swift* que hereda de nuestra implementación base de `AbstractRule`: `AbstractAntlrVisitor`) lo cual demuestra la simpleza a la hora de integrar lenguajes.

Ya teniendo la regla base para implementar las demás, restaban un par de pasos para completar con el objetivo del presente proyecto. En primer lugar, necesitábamos definir aquellas cuestiones requeridas en el módulo `pmd-swift` para que *Swift* se reconozca como lenguaje con soporte para *PMD* y *CPD*. Para ello había que implementar diversas entidades en nuestro módulo. Se requería de un `LanguageModule` para *Swift* (`SwiftLanguageModule`). El mismo ya se encontraba implementado para dar soporte a *CPD*, pero para tener soporte *PMD* se requería agregar un `RuleChainVisitor` (`AntlrRuleChainVisitor` en este caso por tener una gramática *ANTLR*) y un `Handler` (`SwiftHandler`). Este último sirve para integrar en *PMD* toda la lógica específica del lenguaje en cuestión y tiene dos propósitos. En primer lugar, proveer un acceso a la instancia de `RuleViolationFactory` (`AntlrRuleViolationFactory` en este caso); y en segundo lugar, proveer un acceso al *adapter* del *parser*.

Restaba ahora únicamente poder definir reglas particulares de utilidad para el lenguaje *Swift* tanto definidas con una expresión *XPath*, como definidas por un *visitor*. Para esto se investigó un *linter* de *Swift* llamado *SwiftLint* [8] que ya contaba con varias reglas de interés para el lenguaje y se

implementaron en total cuatro reglas para *Swift*, siendo dos de ellas de la categoría `bestpractices` y las otras dos propias de `errorprone`. Ambas reglas de buenas prácticas (`ProhibitedInterfaceBuilder` y `UnavailableFunction`) se implementaron utilizando *visitors*, mientras las otras dos (`ForceTry` y `ForceCast`) fueron desarrolladas mediante expresiones *XPath*. Se utilizaron los ejemplos dados en la documentación de *SwiftLint* para poder validar que la implementación de los distintas reglas era correcta y abarcaba todos los casos.

Además de las reglas en sí se generó todo el *boilerplate* requerido para definir *rulesets* y para las distintas categorías en las que se subdividen las reglas dentro de cada lenguaje. Esto se logró por medio de distintos archivos *XML* de configuración propios de cada *ruleset* y categoría. A su vez, se aportaron *tests* para cada regla evaluando diversos casos representativos positivos y negativos, y validando el correcto funcionamiento de nuestras implementaciones contra todos los casos de prueba que tenía *SwiftLint*.

Finalmente, no podíamos dar por terminado el proyecto sin incorporar además toda la documentación pertinente a nuestros aportes. Entre distintos documentos aportados se encuentran algunos propios de las reglas en sí, donde se explica la utilidad de las mismas, cómo evitarlas y se dan ejemplos de casos positivos y negativos. También reescribimos la documentación que detalla cómo agregar nuevos lenguajes con soporte *PMD* y la documentación que explica cómo agregar reglas a un lenguaje explicando los pasos a seguir para realizar lo mismo con lenguajes *ANTLR*. Nuevamente resulta sumamente satisfactorio ver la simpleza con la que ahora se pueden incorporar, con muy pocas líneas de código y con escaso o nulo conocimiento de la herramienta, tanto lenguajes nuevos como reglas para lenguajes existentes. Simplemente siguiendo una serie de pasos mucho más breve y mucho más simples que los propios de *JavaCC*, en cuestión de minutos se puede tener un módulo para un lenguaje nuevo, dándole soporte tanto para *CPD* como para *PMD*, lo cual claramente disminuye la barrera de entrada para nuevos contribuidores de *PMD*.

7. Logros

En este apartado se describen los logros adquiridos, en este corto plazo, por el proyecto, vinculados a la contribución hecha hacía la herramienta *PMD* y su comunidad, así como la comunidad de desarrolladores que la utiliza.

CPD

En lo que respecta a la funcionalidad de análisis de código duplicado, aportamos no sólo las implementaciones base necesarias para que agregar soporte para nuevos lenguajes sea simple, sino que además proveemos una documentación al pie y concisa, con pasos a seguir para hacerlo, lo cual, en conjunto a la *API* pública y las clases mencionadas anteriormente, permitió que se agregasen con menos de 300 lineas de código, soporte para los siguientes lenguajes:

- Swift
- Go
- Kotlin
- Dart
- Lua

El primero, como mencionamos anteriormente, ya contaba con soporte para *CPD*, pero se lo integró con el *core* de *PMD*, lo cual aportó la posibilidad de realizar supresiones aprovechando el `TokenFilter`. *Go* fue desarrollado desde cero por nosotros mientras implementábamos esta funcionalidad para asegurarnos de que la facilidad de integración cumplía todos los objetivos planteados inicialmente. Por otra parte, los últimos tres fueron agregados por la comunidad, sin necesidad alguna de explicación por parte del equipo que trabaja en la herramienta, simplemente siguiendo la serie de pasos que especificamos en la documentación. Específicamente el desarrollo del soporte para *Kotlin* en *CPD* fue incluido previo a que hayamos agregado soporte para filtro de *tokens* genérico pero posteriormente a haber agregado el soporte básico para la integración. Debido a esto, nosotros nos encargamos de reescribir esta funcionalidad probando la implementación genérica apenas terminada. Al igual que *Kotlin*, *Lua* y *Dart* cuentan con la posibilidad de incorporar filtros de *tokens* por defecto, y la flexibilidad de agregar filtros particulares de cada lenguaje, como *imports*,

con menos de cinco lineas de código. El soporte para esto último en *Dart* fue agregado por un contribuidor externo utilizando el filtro de *tokens* genérico implementado por nosotros.

Sorprendentemente, la comunidad comenzó a hacer su trabajo inmediatamente después de que nosotros hayamos terminado el nuestro. Sin lugar a dudas esto prueba que la funcionalidad agregada era algo que se estaba esperando, que la simpleza de la implementación estuvo a la altura de los objetivos inicialmente planteados y que a partir de ahora el alcance de la herramienta no estará limitado a los lenguajes sobre los que se le permite operar. Por otra parte, queda demostrado que la comunidad puede contribuir al proyecto con una barrera de entrada muy baja, ya que no es necesario comprender demasiadas entidades para hacer contribuciones sobre nuevos lenguajes y nuevas reglas.

Sin lugar a dudas, esto demuestra la necesidad real que existía por parte de la comunidad, en que las funcionalidades que agregamos, comiencen a tomar parte de *PMD* lo antes posible.

PMD

En lo que respecta a las reglas del lenguaje, también dejamos a disposición de la comunidad, una serie de simples implementaciones y una clara *API*, junto con una serie de pasos requeridos a modo de documentación, para que agregar reglas sintácticas y semánticas sea muy simple tanto usando el patrón *visitor* como a través de *XPath*. A diferencia del soporte para CPD, el soporte para *PMD* aún no se encuentra en el último *release* estable de la herramienta y se espera que salga a producción con la próxima versión *major*. Esto se debe a que *PMD* utiliza versionado semántico [6] para sus *releases*, lo que implicó que desarrollemos esta nueva funcionalidad que acarrea cambios en diversas estructuras, para el siguiente *major release*. Debido a esto no existen aún casos de éxito tan marcados como los de *CPD*, pero consideramos que por la simpleza de la implementación y de los requerimientos, y por la experiencia de utilizar nosotros mismos las herramientas creadas, el abanico de lenguajes con soporte total para *PMD* crecerá drásticamente en el los próximos meses.

En este caso dejamos dos reglas utilizando el patrón *visitor* y otras dos utilizando *XPath* sobre el lenguaje *Swift*, a modo de ejemplo para que en conjunto con la documentación, la comunidad tenga todas las herramientas necesarias para seguir contribuyendo a este proyecto, al menor costo posible. Se optó por realizar estas reglas de ejemplo en *Swift* para darle un arranque formal al soporte de *Swift* en *PMD* ya que la comunidad de este lenguaje es muy activa y creemos que la introducción de esta herramienta en dicha comunidad puede potenciar mucho a *PMD*.

Finalmente consideramos importante destacar que todas las nuevas funcionalidades que ahora

forman parte de *PMD*, tienen asociada una documentación detallada sobre como hacer uso de las mismas, así como también un conjunto de *APIs* pensadas en minimizar los costos de contribución de desarrolladores que quieran participar en este proyecto. Creemos que esta es una de las piezas fundamentales que le permitirán a *PMD* crecer ampliamente.

Contribuciones

Debido a que *PMD* es un proyecto *open source*, se detallan a continuación todos los cambios realizados en forma de *pull request*:

- [core] [CPD] Decouple Antlr Tokenizer implementation from any CPD language supported with Antlr
- [core] [cpd] Generalize *ANTLR* tokens preparing support for *ANTLR* token filter
- [go] [cpd] Add CPD support for Antlr based grammar on Golang
- [kotlin] Kotlin tokenizer refactor
- [core] Antlr token filter
- [core] Add final modifiers into AbstractNode
- [doc] Update CPD documentation
- [core] Node support for Antlr-based languages
- [core] Antlr visitor rules
- [core] [swift] Antlr Base Parser adapter and Swift Implementation
- [swift] Feature/swift rules
- [swift] UnavailableFunction Swift rule
- [doc] Add *ANTLR* documentation

8. Conclusión

A lo largo de este proyecto, todo el equipo en conjunto fue detectando conclusiones personales y enseñanzas que este proyecto nos fué dejando a cada uno, y en esta sección listaremos los más relevantes, aquellas lecciones y conclusiones que cierran un ciclo académico de muchos años, esfuerzos y aprendizajes.

En primer lugar, y muy temprano en el proyecto, quedó en evidencia que contribuir en un proyecto *open source* implica una serie de responsabilidades a la hora de escribir código, como gratificación al ver que uno ayudó a construir una herramienta de uso masivo en el mercado. En cuanto a las responsabilidades, es importante entender varios factores, por un lado, en este caso particular, la dimensión del proyecto, tanto por su tamaño (más de 21000 archivos), como por su comunidad, con más de 10 lenguajes soportados, 160 contribuidores y más de 8 años de evolución, lo cual hace que uno tome dimensión de que el código que uno desarrolla, será mantenido por esta comunidad, servirá de ejemplo y guía para otros contribuidores posteriores y formará parte de una herramienta de la que miles de programadores dependen día a día. Por otro lado, uno es responsable por las contribuciones que realiza y la calidad con la que decide hacerlas, a modo de *curriculum*, en un lugar privilegiado, pero también en un lugar donde toda la comunidad puede ver y juzgar, la calidad y la seriedad con la que uno se toma su trabajo.

Sin lugar a dudas, y al poco tiempo de haber comenzado el proyecto, confirmamos uno de los supuestos que nos habían llevado a la decisión de elegir este, como nuestro proyecto final de la carrera de grado, y es que, resulta imprescindible, a nuestros ojos, que los alumnos en su formación profesional, transiten este camino, no solo por las ventajas de expandir su *curriculum* ni las gratificaciones de contribuir haciendo lo que a uno lo apasiona, sino para comenzar a tomar dimensiones, del impacto que uno puede tener en las personas, como programador.

Si bien tomamos aprendizajes y conclusiones propias al desarrollo técnico del proyecto, y a la herramienta *PMD*, estamos totalmente convencidos, que de caras al futuro, el aporte desde el punto de vista técnico, será anecdótico. Creemos que el proyecto tiene gran relevancia para la comunidad de *PMD* y que realmente le permitirá expandirse y penetrar en muchos lenguajes como la herramienta de análisis estático de código principal gracias al desarrollo de este proyecto. Sin embargo, consideramos que fue un medio para llegar a un objetivo de mucho mayor impacto personal y profesional en la vida de cada uno de los involucrados, permitiéndonos crecer personal y

profesionalmente de una manera en la que no lo podríamos haber hecho sin este proyecto.

Finalmente, luego de haber transitado todo este camino, creemos que este proyecto no es sólo la conclusión de una etapa fundamental en nuestra vida, sino también el comienzo de un nuevo ciclo, uno en el cual el alcance e impacto, como se vió en este proyecto, es de carácter global.

Referencias

[1] Repositorio *git* de *PMD*.

`https://github.com/pmd/pmd`

[2] Documentación oficial de *PMD*.

`https://pmd.github.io/latest/`

[3] Documentación oficial de *ANTLR*.

`https://www.antlr.org/api/`

[4] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf; Edición: 2 (25 de enero de 2013)

[5] Documentación oficial de *JavaCC*.

`https://javacc.org/doc`

[6] Versionado semántico *Semver*.

`https://semver.org/`

[7] Repositorio de gramáticas de *ANTLR*.

`https://github.com/antlr/grammars-v4`

[8] Reglas de *SwiftLint*.

`https://github.com/realm/SwiftLint/blob/master/Rules.md`