Mining Common Syntactic Patterns Used by Java Programmers

Alvaro Losada, Guillermo Facundo, Miguel Garcia, and Francisco Ortin

***Abstract*— Open source code repositories provide massive data as programs that have been used to develop different tools. These kinds of works have been included in the active Big Code and Mining Software Repositories research fields. Although different machine learning works already classify the syntactic constructs used by programmers, there are no reports about the most common syntactic patterns used by Java programmers. In this article, we describe a system we build to provide such a report. Our system retrieves the syntactic patterns used by Java programmers, distinguishing those utilized by experts and beginners. We also present the anomalies found in the usage of different syntactic constructs. We modify the OpenJDK compiler to double the syntactic information included in its Abstract Syntax Tree (AST), define a mechanism to translate ASTs into *n*-dimensional vectors, combine the information of different syntax constructs to build heterogeneous patterns, and apply the Frequent Pattern Growth algorithm to mine the syntactic patterns as association rules. The mined patterns allow expressing hierarchical subpatterns connected to one another, providing a high level of expressiveness.**

***Index Terms*—Syntactic patterns, rule mining, Abstract Syntax Trees, association rules, Java.**

I. Introducción

Con la aparición de plataformas de desarrollo colaborativo tales como GitHub, Bitbucket o SourceForge, la disponibilidad de repositorios de código abierto ha crecido de forma significativa [1]. Este elevado número de programas se ha utilizado para crear distintas herramientas orientadas a mejorar el desarrollo de software, tales como traducción entre lenguajes [2], documentación automática de código [3], autocompletado de código [4] o corrección automática de errores [5]. Estas herramientas utilizan técnicas de procesamiento del lenguaje natural y aprendizaje automático, recibiendo como entrada textual el código fuente de los programas.

En los lenguajes de programación textuales, los programas están compuestos de archivos de código fuente escritos como texto, junto con otros recursos adicionales. No obstante, el código textual en realidad encierra información sintáctica y semántica que es comúnmente representada con determinadas estructuras de datos, tales como árboles y grafos [6]. La principal estructura de datos para representar sintácticamente un programa es el árbol de sintaxis abstracta (*Abstract Syntax Tree, AST)*. Cada nodo de un AST representa una construcción sintáctica de un programa como una definición de un tipo o una variable, la invocación a un método o una expresión aritmética.

Las estructuras sintácticas de los programas obtenida a través de sus ASTs ofrecen información adicional al mero texto de su código fuente. La utilización de la información sintáctica de los programas ha permitido el desarrollo de herramientas avanzadas tales como desofuscadores [1], detectores de vulnerabilidades de seguridad [7], decompiladores [8] o clasificadores de programadores [9].

Adicionalmente a la creación de herramientas, las estructuras sintácticas de los programas se pueden emplear para minar, analizar y documentar los patrones sintácticos recurrentes utilizados por los programadores. Así, se podrían detectar fracciones de código altamente repetidas (*idioms*), asociar patrones al nivel de experiencia de los programadores, detectar patrones utilizados que pudiesen dar lugar a errores o reconocer los patrones de uso de una nueva característica añadida a una nueva versión de un lenguaje (por ejemplo, los *records* añadidos a Java 15). Ésta es una información valiosa para mejorar el desarrollo de IDEs (*Integrated Development Environment*), compiladores y decompiladores. Asimismo, los patrones propensos a errores detectados en los programadores inexpertos pueden ser utilizados en cursos de programación para enseñar por qué no utilizarlos, de forma similar a cómo se enseñan patrones comunes de programadores expertos [10]. Tales patrones también pueden ser empleados para implementar un sistema inteligente de tutorización (*Intelligent Tutoring System*, ITS) que evalúe cómo el estudiante mejora, tras analizar las construcciones sintácticas que es capaz de utilizar, ofreciéndole tareas más avanzadas conforme evolucione y reforzando conocimientos cuando la evolución no sea tan patente [11].

Lamentablemente, la mayor parte de los algoritmos de minería de datos requieren que la información a procesar se encuentre en tablas en las que cada instancia o individuo sea representado como vectores de *n* dimensiones de tamaño fijo. No obstante, tal y como hemos comentado, los programas se representan como árboles (ASTs), dificultando así la utilización de los algoritmos clásicos de minería de datos. Asimismo, los ASTs constituyen estructuras heterogéneas (tales como expresiones, definiciones, sentencias y tipos), dificultando así el cumplimiento del requisito del tamaño fijo de los vectores.

La principal contribución de este trabajo es la extracción de patrones sintácticos de código Java de programadores expertos y novatos, definiendo técnicas de transformación de ASTs heterogéneos a tablas de tamaño fijo que nos permitan aplicar los algoritmos clásicos de minado de datos. Los patrones extraídos son expresados mediante reglas que permiten la composición jerárquica de subpatrones heterogéneos conectados entre sí, proporcionando un nivel de expresividad superior al de los trabajos existentes. Para ello, 1) diseñamos un AST con un grano de detalle de sus entidades mucho más fino que el utilizado por

All the authors are with the Computer Science Department, University of Oviedo, c/ Federico Garcia Lorca 18, 33007, Oviedo, Spain. E-mail: {uo272296, uo236856, garciarmiguel, ortin}@uniovi.es.

el compilador del OpenJDK [12]; 2) modificamos el análisis sintáctico que realiza dicho compilador para crear estos nuevos árboles; 3) definimos un mecanismo de traducción de los ASTs heterogéneos a información tabular; 4) analizamos y detectamos valores anómalos; y 5) extraemos y documentamos los patrones sintácticos de los programadores Java mediante reglas de asociación.

Este artículo se encuentra organizado de la siguiente manera. La siguiente sección describe el trabajo relacionado. La Sección III presenta el sistema propuesto y la Sección IV la metodología seguida. En la sección V se explican los resultados y la Sección VI detalla las conclusiones y el trabajo futuro.

II. Trabajo Relacionado

Allamanis y Sutton proponen un método para extraer automáticamente patrones del código fuente (*idioms*), encontrando construcciones sintácticas recurrentes [13]. Desarrollaron el sistema Haggis que analiza fragmentos sintácticos que se repiten con frecuencia, utilizando gramáticas probabilísticas de sustitución de árboles. Tras aplicar su sistema a distintos proyectos de código abierto, encontraron patrones para la creación de objetos, tratamiento de excepciones y gestión de recursos.

Iyer y Zilles realizaron un estudio de 12 asignaturas de programación de primer año en titulaciones de informática de 9 universidades distintas, buscando los patrones sintácticos que los alumnos deben manejar para aprobar los exámenes y las prácticas [10]. Encontraron que se necesitaban 15 patrones para resolver todos los problemas, de los que 9 de ellos se enseñaban en 9 de las 12 asignaturas y 5 patrones sólo se abordaban en 3 de las asignaturas.

Diversos trabajos han creado modelos clasificadores a partir de ASTs, pero sin tener por objetivo el minado de patrones sintácticos. En el trabajo realizado por Ortin *et al.*, se crean modelos para clasificar programadores por su nivel de experiencia [9]. Se parte de un conjunto de ASTs de igual estructura y se traducen éstos a tablas en función de las características de los nodos del árbol. Con estas tablas, se construyen clasificadores para cada tipo de AST mediante árboles de decisión. Posteriormente se entrenan otros clasificadores para ASTs de programas enteros, combinando los clasificadores obtenidos anteriormente. El sistema es capaz de distinguir programas desarrollados por novatos o expertos con una precisión del 99,6%.

Baxter *et al.* utilizan ASTs para crear una herramienta de detección de fragmentos de código duplicado [14]. Analizan ASTs de más de 400.000 líneas de código, encontrando en torno a 12,7% de código duplicado. Adicionalmente, el sistema es capaz de utilizar los patrones de código duplicado encontrados para sugerir modificaciones al programador, ayudando en la refactorización del código para evitar así duplicidades. La información sintáctica del código fuente ha sido utilizada por diversos autores para detectar clones (código duplicado), incluyendo recientemente clones independientes del lenguaje [15], con un elevado rendimiento mediante aprendizaje profundo [16] y esquemas de agregación de ASTs aprendidos mediante redes neuronales recursivas [17].

Se han utilizado reglas de asociación para analizar cómo el código de pruebas (*testing*) evoluciona conforme evolucionan los proyectos, desde su desarrollo hasta su producción [18]. El sistema es evaluado mediante dos casos de estudio diferentes: un proyecto de código abierto y otro de un software industrial. El resultado es que existe coevolución del desarrollo y las pruebas, pero varía en función del tipo de proyecto, no pudiendo obtener patrones generales a ambos proyectos.

NAR-Miner utiliza reglas de asociación en código fuente para detectar errores de programación [19]. NAR-Miner permite la negación en los consecuentes de las reglas minadas, aumentando así el número de reglas obtenidas. Para reducir los falsos positivos (reglas que realmente no detectan un error), diseñan un algoritmo de minado de reglas con restricciones semánticas. NAR-Miner detectó 17 errores en el *kernel* de Linux y 6 errores en PostgreSQL, OpenSSL y FFmpeg.

III. Descripción del Sistema

La Fig. 1 muestra el diagrama de la arquitectura de nuestro sistema. A continuación, describimos someramente sus elementos para posteriormente profundizar en los mismos. La entrada del sistema es un conjunto de programas Java obtenidos tanto de GitHub como de dos asignaturas de primer año de programación de un Grado en Ingeniería del Software (detallado en la Sección IV.A), para utilizar código escrito por principiantes y expertos. La salida es una colección de los patrones sintácticos más frecuentemente utilizados, su relación con el nivel de experiencia del programador y un informe acerca de los patrones atípicos detectados.

Lo primero que hacemos es modificar el compilador de Java del OpenJDK, aumentando la información sintáctica ofrecida por éste. Para ello rediseñamos su AST, añadiendo nuevos nodos y relaciones, proporcionando así una información mucho más detallada que la ofrecida por el compilador original (Sección III.A).

Establecemos una clasificación de los distintos tipos de subárboles con estructura común, encontrando los siguientes 7 tipos de construcciones sintácticas: programa (compuesto como una colección de tipos definidos en él), definición de tipo (clase, interfaz, enumerado o registro), definición de campo, definición de método, sentencia (o instrucción), expresión y tipo. Se definen 7 algoritmos de recorrido de estos tipos de subárboles, guardando la información de cada construcción sintáctica en 7 tablas distintas (Sección III.C).

Posteriormente, los datos de las tablas son procesados, combinados y filtrados para crear los *datasets* que se utilizarán como entrada de los algoritmos de extracción de reglas y detección de anomalías. Si las anomalías son debidas a errores de medición (entradas que no debían haber sido consideradas) se eliminan; en caso contrario, se incluyen en un informe de anomalías y se consideran en las reglas de asociación (Sección III.G). Los patrones sintácticos son finalmente generados por el sistema.

A. Modificación del Compilador de Java

Hemos utilizado el API que ofrece el compilador del OpenJDK para extender su implementación mediante *plug-ins* [12]. Para cada fichero Java, el compilador realiza un análisis sintáctico del mismo y genera el AST original. Recorremos dicho

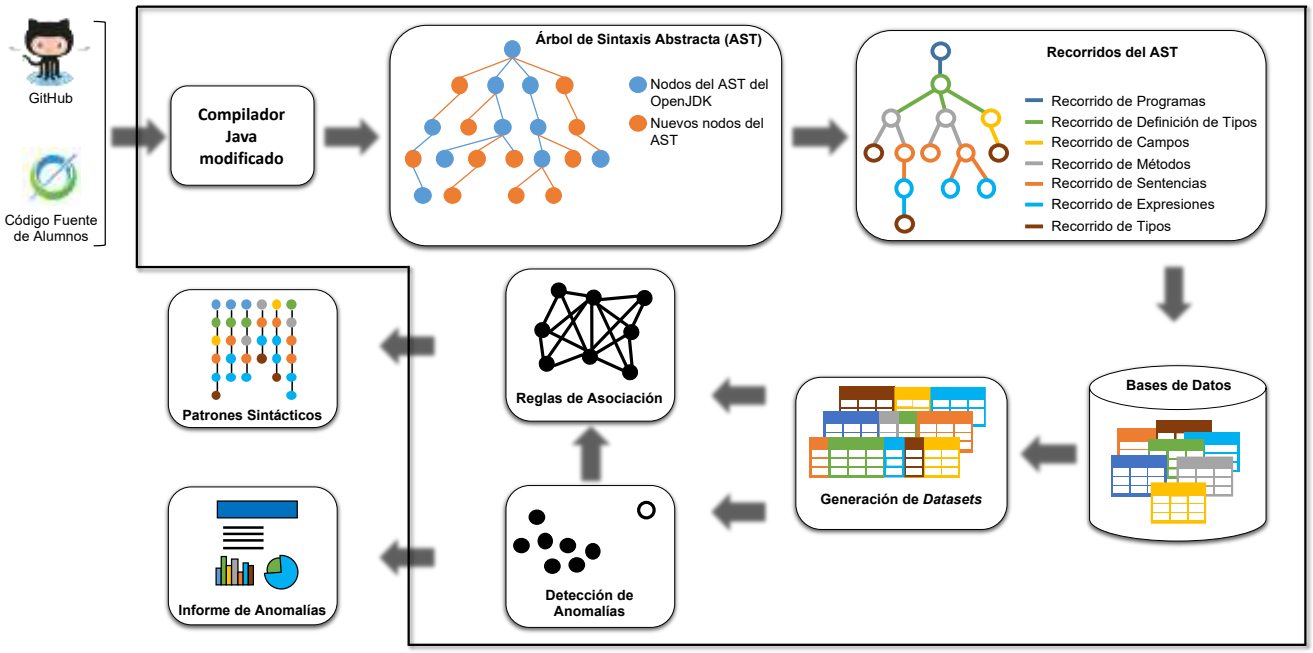

Fig. 1. Arquitectura del sistema propuesto. El sistema se nutre de código de expertos (GitHub) y de alumnos inexpertos para obtener elevado abanico de patrones sintácticos. Se modifica el compilador de Java para aumentar la información de los ASTs de los programas y generar distintas tablas para las distintas construcciones sintácticas mediante distintos recorridos de los ASTs. Estas tablas se combinan en *datasets* heterogéneos para finalmente generar los patrones sintácticos junto con un informe de anomalías o valores atípicos.

AST mediante el patrón de diseño *Visitor* para crear otro AST con información más detallada (siguiente subsección). Por ejemplo, todas las expresiones binarias en Java son representadas en el OpenJDK como instancias de `BinaryTree`, dificultando el conocimiento del tipo concreto de expresión que representa. Para dar más información acerca de la expresión, hemos añadido nodos que representan expresiones aritméticas (`Arithmetic`), de comparación (`Comparison`), lógicas (`Logical`) o a nivel de bits (`Bitwise`), entre otras [20].

B. *Nueva Estructura del AST*

El diseño del AST de OpenJDK está formado por un total de 55 clases. Nosotros añadimos 56 nuevas clases y campos [20] para proporcionar más información sintáctica de los programas y así facilitar el minado de patrones sintácticos.

Nuestro diseño crea nuevas generalizaciones de árboles para definiciones (tales como, variable, clase, interfaz y método) y tipos (entero, *array*, referencia y cadena de caracteres). También especializa varias clases pertenecientes a las construcciones sintácticas de sentencia y expresión. A modo de ejemplo, OpenJDK modela cualquier expresión utilizada como una sentencia mediante una instancia de su clase `ExpressionStatement`. Esta clase no distingue construcciones sintácticas como "`a=a+1`" y "`++a`". No obstante, la primera suele ser más habitual entre los programadores noveles, pudiéndose detectar como patrón común a este tipo de programadores. Por ello, nuestro diseño del AST incluye los siguientes tipos de nodos para las expresiones que también pueden utilizarse como sentencias: `Assignment`, `{Post,Pre}fix{De,In}crementUnary`, `MethodInvocation`, `CompoundAssignment` y `NewClass`.

Además de nuevas clases, nuestro diseño también añade nuevos campos para almacenar información sintáctica de interés. Por ejemplo, nuestros nodos del AST incorporan información relativa al rol que éstos juegan en su construcción sintáctica padre. Así, una asignación puede jugar dos roles distintos si su padre es una sentencia `while`: ser la condición del `while` o una sentencia de su cuerpo. La información de saber cuándo una asignación se utiliza como condición es significativa, por ejemplo, para detectar programadores no principiantes (éstos rara vez usan asignaciones en las condiciones de `while`).

Las 111 clases utilizadas, junto con las 56 originales, están disponibles para su consulta en [20].

C. *Generación de Tablas*

Tal y como hemos mencionado, los algoritmos clásicos de minería de datos se basan en datos tabulares por lo que, para utilizar dichos algoritmos, necesitamos convertir los ASTs a tablas. Lo primero es definir los distintos tipos de subASTs que puedan tener una estructura homogénea, para poder almacenarlos en la misma tabla. Identificamos las siguientes 7 construcción sintácticas (tipos de subAST): programa (colección de sus tipos definidos), definición de tipo (clase, interfaz, enumerado o registro), definición de campo, definición de método, sentencia (o instrucción), expresión y tipo.

A modo de ejemplo, la Tabla I muestra la información almacenada para las sentencias (el resto de las tablas puede consultarse en [20]). Además del nombre de la clase de la sentencia (su categoría sintáctica), guardamos las categorías sintácticas de sus tres primeros nodos hijos (si alguno de ellos no existe, se le asigna `None`) y la categoría sintáctica de su nodo padre. Como se comentó con anterioridad, también almacenamos el rol que la sentencia juega en el nodo padre. También almacenamos la distancia desde el nodo raíz al actual (altura) y el número de arcos desde el actual al nodo hoja más distante (profundidad).

El último valor almacenado para todos los nodos es el nivel de experiencia del programador.

La traducción de los AST a tablas es realizada por 7 distintos recorridos de cada uno de los 7 subASTs, utilizando el patrón de diseño *Visitor*. Cada recorrido obtiene la información de una construcción sintáctica y la almacena en una tabla de la base de datos.

TABLA I
CARACTERÍSTICAS O ATRIBUTOS DEFINIDOS PARA LAS SENTENCIAS

Nombre	Descripción
Categoría sintáctica	Categoría sintáctica del nodo (nombre de la clase del nodo AST).
Primer, segundo y tercer hijo	Categoría sintáctica del hijo correspondiente.
Nodo padre	Categoría sintáctica del nodo padre.
Rol	Rol del nodo actual en el nodo padre.
Altura	Distancia en arcos desde el nodo actual hasta el nodo raíz .
Profundidad	Distancia en arcos desde el nodo actual hasta el nodo hoja más distante.
Nivel de experiencia	Nivel de experiencia del programador: principiante o experto.

D. Generación de los Datasets

Las 7 tablas creadas describen información sobre construcciones sintácticas homogéneas, permitiéndonos obtener patrones comunes de expresiones o sentencias. No obstante, un programa posee múltiples subárboles de distintas construcciones, representado mediante información heterogénea (p. ej., un programa tiene una clase que define un método, donde se usa una sentencia que incluye varias expresiones distintas). Por ello, además de utilizar la información homogénea, debemos crear *datasets* que incluyan la información heterogénea y así poder minar patrones sintácticos heterogéneos y más expresivos.

La creación de los *datasets* heterogéneos se realiza mediante operaciones de disgregación de datos (*drill down*) que combinan la información de las distintas tablas [21]. Así creamos los 5 siguientes *datasets* heterogéneos:

(i) Una instancia por cada tipo definido, incluyendo la información de su programa.
(ii) Una instancia por cada campo de un tipo definido, incluyendo la información del programa y el tipo del campo.
(iii) Una instancia por cada método de un tipo definido, incluyendo la información del programa y los tipos de retorno y de los tres primeros parámetros (`None` si no los hubiere).
(iv) Una instancia por cada sentencia, incluyendo la información del programa, tipo y método en el que se ha definido.
(v) Una instancia por cada expresión, incluyendo la información del programa, tipo, método y sentencia en el que se ha definido.

Por lo tanto, nuestro sistema crea 12 conjuntos de datos para el minado de patrones sintácticos: 7 homogéneos y 5 heterogéneos.

E. Detección de Valores Atípicos

En los *datasets* utilizados, es posible que puedan encontrarse instancias con valores atípicos, representando patrones sintácticos significativamente distintos al resto de la población. La detección de los mismos es importante para poder documentar dichos patrones, al constituir información valiosa relacionada con la contribución de nuestro estudio. Adicionalmente, determinadas anomalías pueden deberse a errores en los datos, que deben ser eliminadas para evitar conclusiones erróneas.

Nuestros conjuntos de datos poseen información numérica y categórica. Para detectar valores anómalos en los datos numéricos, empleamos el test de Tukey basado en comparaciones con el rango intercuartil, definido como la diferencia entre el tercer y primer cuartil de una distribución ($IQR = Q_3 - Q_1$) [22]. De este modo, se define un valor atípico como aquél que no pertenece a alguno de los dos siguientes intervalos:

$$[Q_1 - 1{,}5 \times IQR,\ Q_3 + 1{,}5 \times IQR] \quad (1)$$
$$[Q_1 - 3 \times IQR,\ Q_3 + 3 \times IQR] \quad (2)$$

La primera expresión define un intervalo en el que se encontrarán la mayoría de los valores; en caso contrario, se considerará como un valor atípico leve. La segunda expresión define un intervalo aún mayor, por lo que los valores fuera del mismo son valores atípicos extremos. En base a la distribución de los valores de la variable, consideraremos (1) o (2) como anomalías, eligiendo (2) siempre que haya valores atípicos extremos y (1) en caso contrario.

Para el caso de las variables categóricas, empleamos un análisis de frecuencia en el que consideramos que un valor categórico es atípico cuando su número de ocurrencias es menor que 0,2% / *número posibles valores*. Por ejemplo, una variable booleana tendrá un valor anómalo cuando éste ocurra menos del 0,1% del total.

Una vez detectados los datos anómalos, se analizan si se deben a errores de medición (por ejemplo, haber considerado un código de un alumno incompleto). En dicho caso, la instancia anómala se elimina del *dataset* para evitar sesgos en la información minada. Adicionalmente, los atributos utilizados miden dispersión de datos mediante percentiles y no utilizan promedios, evitando así que los valores anómalos impacten significativamente en la información minada.

F. Preparación de los Datos

Los algoritmos de minería de reglas de asociación utilizados para la extracción de patrones sintácticos requieren que los atributos del *dataset* sean binarios o dicotómicos (cierto o falso) [23], por lo que es necesario un procesamiento previo de los datos [24]. En el caso de un atributo categórico que posea *n* posibles valores, sustituimos dicho atributo por *n* atributos binarios (representación *one-hot*). Por ejemplo, el atributo que representa el nivel de ocultación de un método (su visibilidad) se traduce a los cuatro siguientes atributos binarios: público, protegido, privado y paquete.

La binarización de los atributos numéricos continuos es llevada a cabo mediante una discretización de los mismos, dado que es la aproximación más común a la hora de convertir datos continuos en discretos [25]. Para la discretización, se tiene en cuenta la distribución de los datos y se dividen éstos en distintos intervalos. Para distribuciones normales y uniformes, discretizamos los atributos continuos en cinco intervalos en los que todos los intervalos tienen el mismo número de instancias (igual frecuencia). Para el resto de distribuciones, la discretización se basó en los cinco agrupamientos obtenidos

mediante el algoritmo de aprendizaje automático no supervisado K-means.

G. Reglas de Asociación

Las reglas de asociación se utilizan en problemas de minería de datos con el objetivo de obtener asociaciones, patrones frecuentes y correlaciones entres los valores de atributos que forman un conjunto de datos [24]. Si un *dataset* está compuesto por *n* atributos binarios o *items* $I=\{i_1, i_2, \dots i_n\}$, una regla de asociación se define como la siguiente implicación:

$$X \Rightarrow Y \quad donde\ X, Y \subseteq I \ \ y \ \ X \cap Y = \emptyset \tag{3}$$

De esta forma, una regla típica podría tener la forma "`IF` *atributo*$_1$ `AND` *atributo*$_2$ `THEN` *atributo*$_3$ `AND` *atributo*$_4$", siendo *atributo*$_1$ y *atributo*$_2$ los predecesores o antecedentes de la regla y *atributo*$_3$ y *atributo*$_4$ los sucesores o consecuentes.

Dada una regla de asociación, la cobertura o soporte (*support*) y confianza o precisión (*confidence*) son dos medidas de su calidad, definiéndose como:

$$cobertura(X \Rightarrow Y) = \frac{nº\ de\ instancias\ que\ contienen\ X\ e\ Y}{nº\ total\ de\ instancias} \tag{4}$$

$$confianza(X \Rightarrow Y) = \frac{nº\ de\ instancias\ que\ contienen\ X\ e\ Y}{nº\ de\ instancias\ que\ contienen\ X} \tag{5}$$

La cobertura mide la proporción de instancias que la regla cubre, mientras que la confianza indica la proporción de veces que la regla se cumple cuando ésta se puede aplicar (cuántas veces el consecuente es cierto, tras serlo el antecedente).

Un conjunto de atributos o *itemset* es cualquier combinación de atributos o *items* del conjunto de datos ($\{i_1, i_2, \dots i_m\}$). Los algoritmos de minería de reglas de asociación se basan en la búsqueda de atributos que se dan comúnmente (*frequent itemsets*) para luego crear las reglas de asociación, siendo el algoritmo *apriori* uno de los más conocidos. Un problema de este algoritmo es que transforma el conjunto de datos en un conjunto muy elevado de *itemsets* candidatos, requiriendo muchos recursos de memoria y tiempo de computación [25]. Por este motivo utilizamos el algoritmo *FP-Growth (Frequent Pattern Growth)*, que representa los *itemsets* como un árbol de patrones frecuentes o *FP-tree*, reduciendo el tiempo y la memoria de ejecución [25].

IV. Metodología

La Fig. 2 describe los pasos de la metodología aplicada. Tras crear los *datasets* homogéneos y heterogéneos (Sección IV.A), realizamos la detección, análisis y documentación de los valores anómalos (Sección III.E), eliminando aquéllos que representen errores metodológicos. Tras discretizar los datos (Sección III.F), definimos una cobertura y confianza mínima (Sección IV.B). Para cada uno de los *datasets*, generamos las reglas de asociación. Si éstas asocian un patrón sintáctico a la experiencia del programador, se aplica selección de características para reducir el número de sus atributos (Sección IV.C); en caso contrario, se aumenta paulatinamente la cobertura mínima.

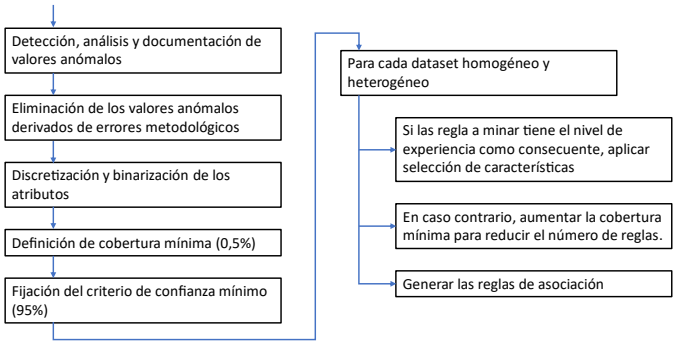

Fig. 2. Pasos de la metodología aplicada.

A. Conjunto de Datos

El conjunto de datos utilizado incluye programas escritos tanto por programadores principiantes como por expertos. Los programas de principiantes han sido obtenidos de estudiantes de primer año del Grado en Ingeniería de Software de la Universidad de Oviedo, durante los años 2018/19 y 2019/20, para las asignaturas de Introducción a la Programación y Metodología de la Programación. Cada programa representa un examen o una práctica realizada por un alumno. Así, tal y como se muestra en la Tabla II, obtuvimos un total de 4.515 programas con 37.701 ficheros Java.

TABLA II
NÚMERO DE NODOS DE LOS ASTS

	Principiante	Experto	Total
Programas	4.515	136	4.651
Definición de tipos	38.452	58.504	96.956
Definición de campos	104.579	135.419	239.998
Definición de métodos	256.562	369.074	625.636
Sentencias	1.173.327	1.963.408	3.136.735
Expresiones	3.939.626	7.296.366	11.235.992
Tipos	628.100	1.131.536	1.759.636
Total	6.145.161	10.954.443	17.099.604

El código fuente de los programas escritos por expertos fue obtenido de diferentes repositorios de código abierto en GitHub, con al menos 1.000 contribuyentes. Los proyectos elegidos fueron *Chromium*, *LibreOffice*, *MySQL* y 133 proyectos diferentes pertenecientes a *Amazon Web Services*. Esto hace un total de 136 programas con 43.765 ficheros de código fuente.

A partir del código fuente, se generan los ASTs y se recorren éstos para para rellenar las tablas de los 7 modelos homogéneos definidos en la Sección III.C. La Tabla II muestra el número de nodos AST obtenidos para cada una de las tablas, observándose cómo la base de datos tiene un total de más de 17 millones de entradas.

B. Filtrado de Reglas de Asociación

Como hemos comentado, la minería de reglas de asociación se basa en obtener las combinaciones de atributos (*itemsets*) más frecuentes, para luego extraer de éstos las reglas de asociación. Para ello, es necesario indicar la cobertura mínima de los *itemsets*, evitando así minar un número muy elevado de reglas que además cubran muy pocas instancias. El valor de cobertura utilizado ha sido 0,5%, representando en nuestro caso un buen equilibrio entre el número de reglas obtenidas y su cobertura.

Otro criterio de filtrado de las reglas es su confianza mínima. Si establecemos una confianza del 100%, podríamos desechar

TABLA III
REGLAS QUE ASOCIAN CONSTRUCCIONES SINTÁCTICAS A NIVELES DE EXPERIENCIA DE PROGRAMACIÓN

		Número de atributos	*Itemsets* generados	Reglas originales (principiante)	Reglas tras la reducción (principiante)	Reglas originales (experto)	Reglas tras la reducción (experto)
Homogéneos	Programas	29	3.095	9.735	181	648	5
	Definición de tipos	24*	38.923	30.995	1.011	173.408	2.550
	Definición de campos	70	44.295	58.734	333	61.743	924
	Definición de métodos	40*	433.927	209.132	5.298	1.357.117	26.283
	Sentencias	262	17.692	273	15	11.718	680
	Expresiones	283	18.186	739	4	3.769	304
	Tipos	43	5.491	22.451	114	540	16
Heterogéneos	Def. tipos + programas	32*	277.047	1.610.619	2.413	2.469.629	5.633
	Def. campos + tipos + def. tipos + programas	12*	835	42	10	423	69
	Def. métodos + tipos + def. tipos + programas	34*	322.015	4.748.636	4.902	1.180.268	14.307
	Sentencias + def. métodos + def. tipos + programas	27*	92.240	14.377	663	80.146	2.580
	Expresiones + sentencias + def. métodos + def. tipos + programas	31*	266.329	870.013	3.886	671.584	9.007

Soporte mínimo de 0,5%. y confianza mínima del 95% en todos los casos. * Se han aplicado un algoritmo de selección de características.

reglas con una elevada cobertura (importantes) que pudiesen tener algún pequeño contraejemplo (p. ej., algún patrón de principiante muy extendido que en algún caso haya podido utilizar algún experto). Para evitar este problema, fijamos la confianza mínima en 95%, permitiendo un margen de error del 5%.

Una peculiaridad de nuestro conjunto de datos es que tenemos una etiqueta que indica, para cada programa, el nivel de experiencia de su programador. Esto nos permite crear reglas en el que el consecuente sea, precisamente, esta etiqueta. Así podemos minar los patrones sintácticos utilizados por los programadores expertos y por los principiantes, permitiendo aplicar éstos en distintos escenarios como los descritos en la Sección I. Un ejemplo de este tipo de regla es el siguiente[1]:

IF $p \in$ *Programas* **AND**
!*códigoCreadoEnPaquetes (p)* **AND**
tiposDefinidosComoClases(p) = 100%
THEN *nivel del programador* = principiante

La regla indica que los programas que sólo definen clases (no implementan nuevos enumerados, interfaces o registros) y todas están ubicadas en el paquete por omisión, son desarrolladas por principiantes. Este tipo de reglas asocian patrones sintácticos al nivel experiencia de sus programadores. Para el resto de las reglas, no añadimos el atributo de experiencia en la generación de los *itemsets*, obteniendo así los patrones sintácticos seguidos por cualquier programador.

C. Reducción del Número de Atributos

El proceso de discretización y binarización de los atributos descrito en la Sección III.F incrementa el número de atributos de forma considerable. Por ejemplo, los atributos de sentencias mostrados en la Tabla I pasan de 9 a 260 atributos. Los algoritmos de extracción de reglas de asociación poseen limitaciones de rendimiento conforme crece el número de atributos del conjunto de datos [23]. Por ello, reducimos el número de atributos del *dataset*.

Para aquellas reglas que asocian los patrones de código al nivel de experiencia de los programadores, buscamos reducir el número de atributos sin reducir la información necesaria para realizar dicha clasificación. Bajo estas circunstancias, los algoritmos de selección de características o atributos (*feature selection*) ofrecen una buena alternativa. Utilizamos una selección de atributos mediante una técnica de envoltorio (*wrapper*), creando un clasificador *Random Forest* con 100 árboles de decisión y quedándonos con los atributos seleccionados por dicho clasificador (Tabla III).

En las reglas de asociación que no tienen por consecuente el nivel de experiencia del programador, no es posible aplicar la selección de características. En estos casos, aumentamos la cobertura mínima para así poder ejecutar los algoritmos que extraen reglas de asociación (Tabla IV).

D. Entorno de Ejecución

Nuestro sistema se ha implementado modificando el compilador de Java del OpenJDK 11.0.3, y desarrollando el resto de los módulos en Python 3.7.4 con scikit-learn 0.21.3 y mlxtend 0.18.0. Para almacenar los *datasets* utilizamos PostgreSQL 10.15. Todo el código fue ejecutado en un servidor Dell PowerEdge R530 con dos procesadores Intel Xeon E5-2620 v4 2.1 GHz (32 núcleos) con 128GB DDR4 2400 MHz de memoria RAM, ejecutando un sistema operativo CentOS 7.9–2009.1 de 64 bits.

V. RESULTADOS Y DISCUSIÓN

En esta sección mostramos los resultados obtenidos con nuestro sistema. Describiremos un resumen de las construcciones sintácticas anómalas encontradas, así como alguno de los patrones sintácticos minados. El listado completo de los mismos está disponible en [20].

A. Valores Anómalos

Hemos realizado el análisis de valores anómalos, siguiendo el método descrito en la Sección III.E. La siguiente información es un resumen de algunos resultados destacables:

[1] Las reglas obtenidas tienen atributos binarizados, pero cambiamos éstos en este artículo para facilitar su comprensión.

TABLA IV
REGLAS QUE NO ASOCIAN CONSTRUCCIONES SINTÁCTICAS A NIVELES DE EXPERIENCIA DE PROGRAMACIÓN

		Número de atributos	Cobertura mínima	*Itemsets* generados	Reglas originales	Reglas tras la reducción
Homogéneos	Programas	27	10%	335	2.266	1.026
	Definición de tipos	71	20%	5.092	14.014	12.081
	Definición de campos	65	10%	323	735	702
	Definición de métodos	107	20%	9.145	110.681	44.397
	Sentencias	260	10%	262	696	376
	Expresiones	280	10%	139	353	162
	Tipos	41	10%	373	773	490
Heterogéneos	Definición de tipos + programas	98	30%	18.156	223.245	85.071
	Definición de campos + tipos + definición de tipos + programas	173	30%	8.302	82.050	34.329
	Definición de métodos + tipos + definición de tipos + programas	205	50%	33.755	339.780	181.113
	Sentencias + definición de métodos + definición de tipos + programas	403	40%	21.828	130.300	58.014
	Expresiones + sentencias + definición de métodos + definición de tipos + programas	567	40%	51.628	802.203	330.837

Confianza mínima del 95% en todos los casos.

(i) Se detectaron como programas anómalos aquéllos que implementan 17,4% o más tipos que no sean clases (enumerados, interfaces o registros) o aquéllos que poseen más de 12 clases en el paquete por omisión. También detectamos algún programa de alumnos con el 100% de interfaces, debido a que no realizaron la tarea de implementarlos. Al no poder considerarse como programas reales, fueron descartados.

(ii) Las clases reconocidas como atípicas tienen más de 25 métodos, 12 campos, 4 anotaciones o 4 interfaces implementados. La utilización de bloques estáticos y clases anidadas son detectadas como anomalías, no siendo por tanto utilizadas de forma común por los programadores Java (ambas características son empleadas únicamente por programadores expertos). Ninguna clase fue definida como `strictfp`.

(iii) La definición de campos `volatile` y `transient` es atípica y nunca utilizada por principiantes. Lo mismo sucede con la utilización de anotaciones para campos.

(iv) En la definición de métodos, sólo algún programador experto, de forma anómala, utiliza la implementación por defecto de métodos de interfaces de Java 8+. También es anómala la definición de métodos `native`, `synchronized` o con nombres en mayúsculas. Un método es anómalo cuando define más de 17 sentencias, 5 parámetros o variables locales, 4 anotaciones, 1 cláusula `throws` o es sobrecargado en más de una ocasión.

(v) La única sentencia detectada como anómala es el decremento prefijo (`--a`), sólo utilizada en el 0,004% de las ocasiones. Los valores de altura y profundidad anómalos son aquéllos mayores de 6 y 9 respectivamente.

(vi) Las expresiones atípicas son las construidas con operadores a nivel de bit (`|`, `&`, `^` o `~`), asignaciones múltiples como expresiones, decremento prefijo y postfijo, y referencias de miembro (`Tipo::miembro`).

(vii) El único tipo anómalo encontrado es el tipo unión, proporcionado en Java 7+ para capturar excepciones de más de un tipo (`catch(SQLException|IOException e)`).

B. Reglas que Asocian Construcciones Sintácticas a Niveles de Experiencia de Programación

Este tipo de reglas son aquéllas donde el consecuente incluye el atributo del nivel de programación (experto o principiante). Tal y como se muestra en la Tabla III, el algoritmo *FP-Growth* genera un número elevado de reglas que, bajo ciertas circunstancias, pueden ser eliminadas sin perder la información buscada en nuestra investigación.

El primer escenario es cuando tenemos dos reglas "$X \Rightarrow NP$" y "$X \Rightarrow NP, Y$", donde $X, NP, Y \subseteq I$ y *NP* representa el nivel de experiencia del programador. Bajo estas circunstancias, se cumple que la cobertura y confianza de la primera regla no son menores que las de la segunda, por lo que podemos obviar la segunda regla.

FP-Growth también genera reglas de asociación con igual consecuente, en las que los antecedentes de una regla son subconjuntos de otra. Es decir, dos reglas de la forma "$X \Rightarrow Z$" y "$X, Y \Rightarrow Z$", donde $X, Y, Z \subseteq I$. En este caso, la cobertura de la primera no es menor que la de la segunda, por lo que obviamos la segunda. La regla no considerada (la segunda) podría tener una mayor confianza al ser más específica, pero no supondría un problema ya que ambas reglas poseen la confianza mínima deseada no inferior al 95% (en caso contrario no serían generadas por nuestro sistema). La Tabla III muestra cómo estas dos simplificaciones reducen significativamente el número de reglas producidas.

A continuación mostramos dos reglas de ejemplo obtenidas del *dataset* homogéneo de definición de tipos, que asocian patrones sintácticos a programadores expertos:

IF *c* ∈ *Def.Campos* **AND** *tipo(c)* = genérico **AND**
inicialización(*c*) = invocación_método
THEN *nivel del programador* = experto
(cobertura = 8,6%, confianza = 99,9%)

IF *c* ∈ *Def.Campos* **AND** *tipo(c)* = referencia **AND**
es*Constante*(*c*) **AND** *visibilidad*(*c*) = pública **AND**
inicialización(*c*) = llamada_constructor
THEN *nivel del programador* = experto
(cobertura = 3%, confianza = 100%)

La primera regla identifica la definición de un campo cuyo tipo es genérico y su valor se inicializa con una invocación a un método. En el segundo caso, el campo es constante (`final`) y público, y se inicializa con la construcción de un objeto (operador `new`).

El sistema también obtiene reglas asociadas a principiantes,

en las que se definen campos de tipo simple sin inicialización:

IF $c \in$ *Def.Campos* **AND** *inicialización*(*c*) = None **AND**
(*tipo(campo)* = double **OR** *tipo(campo)* = char)
THEN *nivel del programador* = principiante
(cobertura = 1,7%, confianza = 98,4%)

El sistema también es capaz de encontrar reglas utilizando construcciones sintácticas de distintas tablas (*datasets* heterogéneos). La siguiente regla utiliza información de programas y definición de tipos (clases), identificando un patrón propio de programadores novatos en el que se utiliza el paquete por omisión, no se definen interfaces y una clase implementada no hereda de otra clase, ni implementa ningún interfaz, in es anidada, y no utiliza ninguna anotación ni ningún método estático (de clase):

IF $p \in$ *Programas* **AND** *códigoPaquetePorOmisión*(*p*) **AND**
interfacesDefinidos(p) = 0% **AND**
$c \in$ *Def.Tipos* **AND** *tipo(c)* = *Clase* **AND** $c \in p$ **AND**
métodosEstáticosDefinidos(c) = 0% **AND**
númeroInterfacesImplementados(c) = 0 **AND**
númeroAnotaciones(c) = 0 **AND** !*esClaseAnidada(c)* **AND**
númeroClasesHeredadas(c) = 0 **AND**
THEN *nivel del programador* = principiante
(cobertura = 4,1%, confianza = 100%)

La siguiente regla heterogénea nos indica cómo las clases no definidas en el paquete por omisión, que tienen un campo de tipo no primitivo y además son constantes (`final`) son un patrón muy común entre los programadores expertos:

IF $c \in$ *Def.Tipos* **AND** *tipo(c)* = *Clase* **AND**
$c \notin$ *paquetePorOmisión* **AND**
$f \in$ *Def.Campos* **AND** $f \in c$ **AND** *esConstante(f)* **AND**
$t \in$ *Tipos* **AND** $t \in f$ **AND** !*esPrimitivo(t)*
THEN *nivel del programador* = experto
(cobertura = 21,5%, confianza = 98,6%)

Cabe destacar la expresividad de este tipo de reglas en las que, además de permitir la utilización de diversos tipos de construcciones sintácticas, también se permite controlar la existencia de una dentro de la otra (su jerarquización). Por ejemplo, la regla anterior permite indicar que *t* es el tipo del campo *f* ($t \in f$) y que *f* es un campo de la clase *c* ($f \in c$). Aunque las reglas de asociación son altamente utilizadas para minar información, la composición de información sintáctica, heterogénea y jerarquizada que hemos realizado permite minar reglas con una expresividad no ofrecida por los trabajos relacionados [9], [13], en los que se limitan a combinar la existencia de instancias en distintas tablas, pero sin asociar éstas. Es decir, se puede representar que un programa tiene una clase en el paquete por omisión y un atributo no constante, pero no es posible indicar que dicho atributo pertenezca a esa clase en particular [9].

C. Reglas de Asociación que no Utilizan el Nivel de Experiencia del Programador

En este caso minamos reglas de asociación con cualquier estructura, pero excluyendo el atributo de nivel de experiencia del programador. Al no tener ese atributo como clasificador, no es posible aplicar algoritmos de selección de características para reducir su número (Sección IV.C). Debido al elevado número de atributos, *FP-Growth* se queda sin memoria para generar los *itemsets*, por lo que elevamos la cobertura mínima de los mismos, tal y como se muestra en la Tabla IV. También se aplica la segunda eliminación de reglas descrita en la Sección V.B, en las que se obvian las reglas en las que sus antecedentes son un subconjunto de los antecedentes de otra regla y ambas tienen igual consecuente. La Tabla IV muestra cómo la reducción en el número de reglas.

A modo de ejemplo, la siguiente regla de asociación identifica un código común de las clases de utilidad, en el que la práctica totalidad de sus miembros son estáticos (de clase) y, por tanto, no definen un constructor ni derivan de otras clases:

IF $c \in$ *Def.Tipos* **AND** *tipo(c)* = *Clase* **AND**
visibilidad(c) = pública **AND**
camposEstáticos(c) = 100% **AND**
métodosEstáticos(c) = 100%
THEN *númeroContructores(c)* = 0 **AND**
númeroClasesHeredadas(c) = 0
(cobertura = 13,4%, confianza = 99,8%)

El siguiente patrón sintáctico heterogéneo indica que los campos con tipos genéricos y privados están casi siempre definidos en paquetes distintos al paquete por omisión, puesto que son tipos altamente reutilizables (por eso usan genericidad):

IF $c \in$ *Def.Tipos* **AND** *tipo(c)* = *Clase* **AND**
$f \in$ *Def.Campos* **AND** $f \in c$ **AND**
visibilidad(f) = privada **AND**
$t \in$ *Tipos* **AND** $t \in f$ **AND** *esGenérico(t)*
THEN *paquete(c)* ≠ paquete_por_omisión
(cobertura = 31,3%, confianza = 99,3%)

Las reglas mostradas en esta sección son meros ejemplos de las muchas obtenidas, las cuales pueden consultarse en [20].

VI. CONCLUSIONES

El sistema presentado es capaz de minar reglas que describen los patrones sintácticos más utilizados por los programadores Java, permitiendo la diferenciación de los patrones más utilizados por principiantes y expertos. Asimismo, nuestro sistema genera importante información acerca de las construcciones sintácticas anómalas, permitiendo conocer de forma empírica cómo se utilizan las características sintácticas del lenguaje Java. Las reglas minadas permiten expresar subpatrones jerarquizados heterogéneos conectados entre sí, consiguiendo una expresividad no proporcionada por los trabajos relacionados.

Como trabajo futuro planeamos utilizar los *datasets* generados para detectar agrupamientos de patrones sintácticos mediante aprendizaje no supervisado. También queremos aumentar la información extraída del compilador, añadiendo información semántica [26].

Planeamos aplicar este método a varios lenguajes de programación. Para ello, deberíamos realizar dos tareas: la principal es definir la información sintáctica de forma agnóstica al lenguaje de programación; adicionalmente, sería necesario implementar una analizador sintáctico por cada lenguaje soportado

que genere la información sintáctica. El resto del sistema permanecería sin variación.

Todos los datos utilizados en nuestro trabajo, las reglas extraídas, el informe de valores anómalos, el nuevo diseño del AST y todo el código fuente utilizado para implementar nuestro sistema están disponibles para su descarga en http://www.reflection.uniovi.es/bigcode/download/2021/ieee-lat.

AGRADECIMIENTOS

Este trabajo ha sido financiado por el Ministerio de Ciencia, Innovación y Universidades, bajo el proyecto RTI2018-099235-B-I00, así como por la Universidad de Oviedo (proyecto GR-2011-0040).

REFERENCIAS

[1] F. Ortin, J. Escalada, and O. Rodriguez-Prieto, "Big Code: New Opportunities for Improving Software Construction," *J. Softw*., vol. 11, pp. 1008–1083, Nov. 2016, DOI: 10.17706/JSW.11.11.1083-1088.

[2] K. Aggarwal, M. Salameh, and A. Hindle. "Using machine translation for converting Python 2 to Python 3 code". *PeerJ PrePrints*, Oct. 29, 2015. [Online] DOI: 10.7287/PEERJ.PREPRINTS.1459V1, Accessed on: Jun. 10, 2021.

[3] A. V. M. Barone, and R. Sennrich. "A parallel corpus of Python functions and documentation strings for automated code documentation and code generation," 2017. [Online]. Available: arXiv:1707.02275v1 [cs.CL].

[4] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel. "Learning Python Code Suggestion with a Sparse Pointer Network," 2016. [Online]. Available: arXiv:1611.08307v1 [cs.NE].

[5] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using Recurrent neural networks," 2016. [Online]. Available: arXiv:1603.06129v1 [cs.PL].

[6] A. W. Appel, and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed., Cambridge, UK: Cambridge University Press, 2002.

[7] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized Vulnerability Extrapolation using Abstract Syntax Trees," in *ACM Int. Conf. Proc. Series*, 2012, pp. 359–368, DOI: 10.1145/2420950.2421003.

[8] J. Escalada and F. Ortin, "An Adaptable Infrastructure to Generate Training Datasets for Decompilation Issues," in *Advances in Intelligent Systems and Computing*, Cham: Springer International Publishing, 2014, pp. 85–94. DOI: 10.1007/978-3-319-05948-8_9.

[9] F. Ortin, O. Rodriguez-Prieto, N. Pascual, and M. Garcia, "Heterogeneous tree structure classification to label Java programmers according to their expertise level," *Futur. Gener. Comput. Syst.*, vol. 105, pp. 380–394, 2020, DOI: 10.1016/J.FUTURE.2019.12.016.

[10] V. Iyer, and C. Zilles. "Pattern Census: A Characterization of Pattern Usage in Early Programming Courses," in *Proc. of the 52nd ACM Technical Symp. on Comp. Science Education*, 2021, pp. 45–51.

[11] T. Crow, A. Luxton-Reilly and B. Wuensche. "Intelligent tutoring systems for programming education: a systematic review," in *Proc. of the 20th Australasian Comp. Education Conf.*, 2018, pp. 53–62.

[12] Oracle, *Oracle JDK 9 Documentation - Java Platform, Standard Edition Tools Reference*. Accessed on: Jun. 10, 2021. [Online]. Available: https://docs.oracle.com/javase/9/tools/javac.htm#JSWOR627

[13] M. Allamanis, and C. Sutton, "Mining Idioms from Source Code," in *Proc. of the 22nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, 2014, pp. 472–483, DOI: 10.1145/2635868.2635901.

[14] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. of the Int. Conf. on Software Maintenance*, 1998, pp. 368–377, DOI: 10.1109/ICSM.1998.738528.

[15] D. Perez, and S. Chiba, "Cross-language clone detection by learning over abstract syntax trees," in Int. Conf. on Mining Software Repositories, 2019, pp. 518–528, DOI: 10.1109/MSR.2019.00078.

[16] J. Zeng, K. Ben, X. Li, and X. Zhang, "Fast Code Clone Detection Based on Weighted Recursive Autoencoders," *IEEE Access*, vol. 7, pp. 125062–125078, 2019.

[17] L. Büch, and A. Andrzejak, "Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection," in Proc. of the *IEEE Int. Conf. on Software Analysis, Evolution and Reengineering*, pp. 95–104, DOI: 10.1109/SANER.2019.8668039.

[18] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production test code," in *6th IEEE Int. Working Conf. on Mining Software Repositories*, 2009, pp. 151–154, DOI: 10.1109/MSR.2009.5069493.

[19] P. Bian, B. Liang, W. Shi, J. Huang, and Y. Cai, "NAR-Miner: Discovering Negative Association Rules from Code for Bug Detection," in *Proc. of the 2018 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, 2018, pp. 411–422, DOI: 10.1145/3236024.3236032.

[20] A. Losada, G. Facundo, M. Garcia, and f. Ortin, "Mining Common Syntactic Patterns used by Java Programmers (support material website)," 2021. [Online]. Available: http://www.reflection.uniovi.es/bigcode/download/2021/ieee-lat.

[21] M. Joglekar, H. Garcia-Molina, and A. Parameswaran, "Interactive Data Exploration with Smart Drill-Down," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 1, pp. 46–60, 2019.

[22] J. W. Tukey, *Exploratory Data Analysis*. Upper Saddle River, NJ: Pearson, 1977.

[23] R. Agrawal, and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in *Proc. of the 20th Int. Conf. on Very Large Data Bases*, 1994, pp. 487–499.

[24] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. of the 1993 ACM SIGMOD Int. Conf. on Manage. of Data - SIGMOD '93*, 1993.

[25] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*. Oxford, England: Morgan Kaufmann, 2012.

[26] O. Rodriguez-Prieto, A. Mycroft, and F. Ortin, "An efficient and scalable platform for Java source code analysis using overlaid graph representations," *IEEE Access*, vol. 8, pp. 72239–72260, 2020.

Alvaro Losada es Graduado en Ingeniería Informática por la Universidad de León desde 2018. Actualmente es estudiante del Máster de Investigación en Ingeniería Web en la Universidad de Oviedo. Sus intereses de investigación incluyen *machine learning*, *big data* y razonamiento probabilístico.

Guillermo Facundo se graduó en Ingeniería Informática del Software por la Universidad de Oviedo en 2020. Actualmente es estudiante del Máster de Investigación en Ingeniería Web en la Universidad de Oviedo. Sus intereses de investigación incluyen web semántica, desarrollo de software y lenguajes de programación.

Miguel Garcia es Profesor Contratado Doctor de la Universidad de Oviedo. Ingeniero Técnico en Informática (2005), Máster en Ingeniería Web (2008) y Máster en Investigación en Ciencias de la Computación (2010). Es doctor en Informática por la Universidad de Oviedo desde 2013. Investiga en *big code* y lenguajes de programación.

Francisco Ortin Catedrático de Universidad del Departamento de Informática de la Universidad de Oviedo y Profesor Adjunto del *Munster Technological University* (Irlanda). Director del grupo de investigación oficial *Computational Reflection* centrado en lenguajes de programación y desarrollo de software.