Prácticas de Desarrollo Software: Un Estudio Exploratorio con Herramientas de Análisis Estático

Marisa Cecilia Tumino, Juan Manuel Bournissen, Gisela Müller, Reinhard Schmidlin y Emanuel Irrazábal
Universidad Adventista del Plata
marcetumino@gmail.com, juan.bournissen@gmail.com

***Abstract*—La investigación en ingeniería de software ha demostrado que la prevención de defectos desde el comienzo del desarrollo resulta menos costosa que corregirlos más tarde. En este sentido existen numerosos intentos por proveer herramientas con el objeto de sistematizar los procedimientos de análisis estático de código fuente. En este trabajo se analizó el potencial de cinco de estas herramientas seleccionadas por la actualidad de sus versiones, su disponibilidad y su utilización en investigaciones afines. Para el análisis, las herramientas se ejecutaron en el código fuente del software BlueJ, en sus diez últimas versiones, dado su amplio empleo tanto en el mercado de desarrollo de software como en el ámbito de instituciones educativas. Los resultados obtenidos del análisis, y sus conclusiones, acercan posibles soluciones a esta problemática que varía en alto grado conforme a los objetivos trazados por los equipos de desarrollo. El trabajo contribuye a facilitar la selección de herramientas apropiadas para el análisis estático de código fuente en equipos que están comenzando a incluir aspectos de calidad del producto software en las etapas tempranas del desarrollo.**

***Palabras clave*—Mantenibilidad, Análisis y evaluación de la calidad, Revisión y evaluación, análisis estático, Análisis de error, Herramientas de software.**

I. INTRODUCCIÓN

La calidad es considerada un factor clave en las empresas de desarrollo software. Como elemento diferenciador entre sus competidores y como mecanismo para mejorar la imagen hacia los clientes.

La investigación en ingeniería de software ha demostrado que la prevención de defectos desde el comienzo del desarrollo es menos costosa que su corrección posterior. El mantenimiento de software equivale aproximadamente al 70% del costo total del ciclo de vida. Asimismo, la calidad del software influye en los costos de uso del software, dado que durante el uso se pueden producir variados fallos de software, causando fallas de los sistemas físicos, con las consecuentes pérdidas financieras. Los costos indirectos provocados por la baja calidad del software podrían atribuirse a los clientes insatisfechos, al daño de la reputación de las empresas y a la pérdida de oportunidades de mercado (Lochmann, 2013:11) [1].

En coincidencia con Escalona, Pérez Pérez, González Barroso, Ponce, Correa y Merino (2008: 46) [2], "La verificación y validación del código es una de las actividades más críticas en los desarrollos de software que sirven para verificar la calidad de los productos que se generan."

Para Greiner, Dapozo, Acosta, Domínguez, Chiapello y Estayno (2013: 621) [3], la medida de la calidad del software es una necesidad para las empresas de Software y Servicios Informáticos (SSI), porque representa una ventaja estratégica al proporcionar el conocimiento de los procesos productivos y permitir mejorar las tareas menos eficientes. La calidad del software está estrechamente vinculada con la medición del mismo.

Para evaluar la calidad de un producto software son necesarios dos pasos. El primero, la definición de un modelo de calidad abstracto que relacione diferentes características de calidad. En ese sentido, una posible elección son las normas ISO/IEC 9126 e ISO/IEC 25010, las cuales descomponen los conceptos abstractos de alto nivel en atributos de calidad más manejables. Aun así, estos atributos continúan siendo poco concretos como para poder ser medidos directamente en el producto software.

El segundo paso es la selección de herramientas de análisis de código, con las cuales obtener la información necesaria del código fuente, como por ejemplo métricas de acoplamiento, cohesión, complejidad o violaciones de buenas prácticas al codificar.

Una métrica software **¡Error! No se encuentra el origen de la referencia.**4] [5] es un mapeo numérico de una parte del software que cuantifica uno o más atributos software **¡Error! No se encuentra el origen de la referencia.**6]. Estas medidas son comúnmente considerados como factores importantes que reflejan la naturaleza del producto software, incluyendo, entre otras características, las propiedades de complejidad y mantenibilidad **¡Error! No se encuentra el origen de la referencia.**7] **¡Error! No se encuentra el origen de la referencia.**8] **¡Error! No se encuentra el origen de la referencia.**9]. Las métricas de software tradicionalmente incluyen la cantidad de líneas de código fuente, la media de líneas de código por método o clase, el porcentaje de comentarios, la cantidad de decisiones o la cantidad de operadores o parámetros.

Para obtener las métricas que permiten estimar la calidad de los productos software, lo más común y eficiente es utilizar herramientas que permiten realizar estas mediciones y ofrecen informes con los resultados. Estas herramientas pueden clasificarse en dos tipos:

1. Herramientas de análisis dinámico: aquellas herramientas que realizan el análisis del software ejecutando el código fuente de dicho software.
2. Herramientas de análisis estático: aquellas herramientas que llevan a cabo el análisis sin necesidad de ejecutar el software bajo estudio.

El desafío actual es el aprovechamiento de los resultados heterogéneos de herramientas de análisis del código fuente junto con un modelo de evaluación de la calidad del producto software.

En este trabajo se han explorado las principales herramientas de análisis del código fuente en Java que ayudan a los equipos de desarrollo software en las etapas tempranas de la codificación. El artículo está estructurado en 4 secciones, además de la introducción. En la sección 2 se describen los

trabajos relacionados, en la sección 3 se indica el procedimiento tenido en cuenta al seleccionar el lenguaje de programación, las herramientas y el código fuente analizado. En la sección 4 se sintetizan los resultados al aplicar las herramientas al proyecto BlueJ, se describe la experiencia y se analiza la evolución de BlueJ. Finalmente en la sección 5 se enumeran las conclusiones de este trabajo.

II. Trabajos relacionados

Fontana, Braione y Zanoni (2011) [10] revisaron el panorama actual de las herramientas para la detección automática de code smells. Definen las preguntas de investigación sobre la consistencia de sus respuestas, su capacidad para exponer las partes de código más afectadas por problemas estructurales y la relevancia de las respuestas en la evolución del software. Los análisis efectuados mediante la aplicación de cuatro detectores de code smells, a seis versiones diferentes de la aplicación GanttProject, un sistema de código abierto escrito en Java, arrojaron como respuesta una salida cuyo análisis permitió detectar los code smells e identificar los sectores que necesitaban mayor mantenimiento. La metodología aplicada por estos autores fue adaptada con el propósito de desarrollar el presente trabajo.

Por su parte, Lamas Codesido [11] intenta demostrar que la elección de una herramienta no es suficiente para poder detectar todos los errores que contiene un programa o aplicación desarrollada. Una de sus conclusiones afirma que en muchos casos será necesario elegir más de un analizador de distinto tipo para poder abarcar más tipos de bugs.

Escalona, Pérez Pérez, González Barroso, Ponce, Correa y Merino [2] presentan la herramienta PMD para ofrecer una visión de su adaptación a diferentes proyectos, como herramienta adecuada para la verificación y validación de la calidad de los productos software, reconociendo que su implementación no encarece los proyectos de desarrollo.

Ayewah, Pugh, Morgenthaler, Penix. y Zhou [12] entienden que una de las razones por las que el análisis estático a veces informa sobre problemas verdaderos pero triviales es el desconocimiento de la funcionalidad del código fuente. Muchas técnicas de análisis estático buscan código inusual, tales como cómputos dudosos o declaraciones que podrían ser en realidad código muerto, casos que idealmente deben ser limpiados pero podrían no afectar a la funcionalidad del software. Por el contrario, las pruebas generalmente se dirigen a probar cuestiones que el desarrollador cree importantes, y por lo tanto los errores encontrados en las pruebas con mayor frecuencia corresponden a los deterioros de la funcionalidad. El análisis estático impulsado por metas más específicas con anotaciones de los desarrolladores, acerca del comportamiento previsto, puede proporcionar mejores formas de identificar aspectos de mayor impacto.

En este sentido Johnson, Song, Murphy-Hill y Bowdidge [13] confirman que las herramientas de análisis estático están infrautilizadas, aunque los desarrolladores reconozcan que su uso es beneficioso para los ingenieros de software y las herramientas pueden aligerar y abaratar la búsqueda de errores o defectos de software. Mediante entrevistas con desarrolladores encontraron que la forma en que se presentan las advertencias dificultan su uso.

Por su parte, Rutar, Almazan y Foster [14], centraron su trabajo en comparar la salida de cinco herramientas: Bandera, ESC/Java, FindBugs, Jlint y PMD. La principal dificultad que encontraron en su utilización fue la cantidad de salidas que producen, señalando que el programador debe tener la posibilidad de añadir una anotación o un comentario especial en el código para suprimir las advertencias que son falsos positivos. Sin anotaciones del usuario, una herramienta como ESC/Java que aún es poco sólida, produce incluso más advertencias que Jlint, PMD, y FindBugs. En última instancia, hay una amplia área de la investigación abierta para la comprensión de las compensaciones adecuadas de las herramientas evaluadoras de código fuente. Reafirmando esta última idea, Wagner, Jürjens, Koller y Trischberger [15] presentan un estudio de caso dando los primeros indicios de cómo los defectos encontrados por tres herramientas analizadoras de código fuente, FindBugs, PMD y QJ Pro, se relacionan con otras técnicas de detección de errores. La principal conclusión es que las herramientas encuentran diferentes defectos. Sin embargo, la utilización combinada de herramientas de búsqueda de errores, junto con revisiones y pruebas, podría ser más conveniente mientras el número de falsos positivos fuera manejable.

Las pruebas que se realizaron en el presente trabajo estuvieron encaminadas a evaluar las herramientas seleccionadas por su utilización en investigaciones afines, bajo los criterios que se establecen en la siguiente sección.

III. Metodología

A los efectos de ejecutar las pruebas necesarias para la investigación, en primera instancia se tomó la decisión del lenguaje en el cual estuviera construido el código fuente a evaluar. Dado que, conforme a la experiencia académica y profesional de los investigadores, Java constituye un lenguaje muy utilizado en la actualidad, de uso libre, además de ser muy difundido y enseñado en las instituciones educativas, con buena cantidad de herramientas libres que analizan el código fuente, fácil de analizar, compilar, visualizar y ejecutar y con una comunidad significativa de desarrolladores, se optó por este lenguaje multiplataforma. Es importante señalar también que Java es soportado por una cantidad significativa de entornos de desarrollo.

Una vez seleccionado el lenguaje, fue necesario seleccionar la versión del Java Development KIT (JDK), como paquete de herramientas de desarrollo. La versión elegida fue JDK 8 por ser la versión más actualizada al momento de llevar a cabo las pruebas.

El paso siguiente consistió en seleccionar la plataforma de desarrollo a emplear. Si bien las primeras pruebas fueron realizadas con Netbeans, por ser un entorno de desarrollo sugerido por Oracle, la necesidad de probar una cantidad mayor de herramientas en un entorno libre, llevó al equipo investigador a optar por Eclipse, por ser libre y configurable con una mayor cantidad de plug-ins y programas adecuados para el análisis estático del código fuente.

Eclipse fue concebido como un Integration Development Environment (IDE) genérico que goza de popularidad entre la comunidad de desarrolladores del lenguaje Java.

La versión del IDE elegido fue Eclipse Luna que incluye el soporte oficial para JDK 8 y ofrece la compatibilidad con los paquetes de desarrollo que se utilizaron al efectuar las pruebas.

Se seleccionó el código fuente del software BlueJ en diez versiones, en función del versionado apropiado para la interpretación de las pruebas. El motivo de esta selección radicó en su amplio empleo tanto en el mercado de desarrollo de software como en el ámbito de instituciones educativas, en la cantidad de líneas de código que componen el software, en la cantidad y el tamaño de módulos y paquetes que conforman su arquitectura y, especialmente, en la disponibilidad de una

cantidad considerable de versiones que permiten identificar la evolución del código.

El siguiente paso consistió en la elección de las herramientas de código abierto que se aplicarían, en función de la actualización de sus versiones, la disponibilidad, su rendimiento reconocido y popularidad. Las experiencias de otros investigadores como Fontana, Braione y Zanoni [10] y Lamas Codesido [11] han facilitado la elección. Las herramientas plug-ins de Eclipse, elegidas para probar el código BlueJ, fueron: PMD, FindBugs, CheckStyle, UCDetector y IclEmma Java Code Coverage, aunque esta última no comparte indicadores de comparación con las anteriores, se ha visto su amplia utilización como herramienta de análisis en los equipos de desarrollo software. A su vez se utilizó la herramienta Project USUS, un software propietario con licencia libre durante el tiempo de prueba.

Los criterios de análisis que fueron considerados en el estudio de las herramientas empleadas han sido tomados del trabajo de Lamas Codesido [11] y se enumeran a continuación:

1. Extensibilidad: se compara el grado en el que la herramienta permite añadir reglas fácilmente y adaptarlas a nuevos proyectos.

2. Precisión: se indica el número de violaciones de buenas prácticas que encuentra cada uno de los analizadores.

A continuación se describen las características principales de las herramientas seleccionadas, obtenidas de sus respectivos sitios web oficiales.

PMD[1]: es un analizador de código fuente. Se trata de un analizador de código basado en reglas configurables. Posee un conjunto (ruleset) de reglas (rules) básicas y permite implementar reglas adaptadas a cada necesidad, asignándoles la prioridad según su importancia. Esta herramienta es extensible y configurable ya que se pueden añadir nuevas reglas o configurar las que ya se incluyen en caso de que esto fuera necesario. Es totalmente integrable en entornos como NetBeans o Eclipse. Posee la capacidad de analizar código desde la ejecución y detectar código que no se usa, código no óptimo, expresiones redundantes, códigos duplicados y otros posibles errores. Está integrado con los principales IDEs como JDeveloper, Eclipse, jEdit, JBuilder, BlueJ, CodeGuide, NetBeans / Sun Java Studio Enterprise / Creador e IntelliJ IDEA, entre otros [11]. La herramienta contempla 29 ruleset y 344 rules, pudiéndose crear nuevas reglas gracias a la extensibilidad que la caracteriza.

FindBugs[2]: un programa que utiliza el análisis estático para buscar errores en el código Java. Analiza el bytecode para encontrar posibles problemas de eficiencia y malas prácticas. Organiza el informe de incidencias en varias categorías según su gravedad, a cada informe de bug se le asigna una prioridad: alta, media y baja. Una de las mejores características de este analizador es la extensibilidad que permite añadir nuevas reglas para detectar más bugs de los que trae por defecto. Permite crear reglas e incluye 11 categorías y 147 detectors (detectores de errores). A su vez, clasifica los errores en Scariest, Scary, Troubling y Of concern. FindBugs es una herramienta de análisis estático de código abierto que analiza archivos de clase Java en busca de defectos de programación. El motor de análisis informas casi 300 diferentes patrones de errores [12].

CheckStyle[3] es una herramienta de desarrollo para ayudar a los programadores a escribir código Java ajustado a un estándar de codificación. Automatiza el proceso de comprobación de código Java, lo que lo hace ideal para proyectos que pretenden mantener un estándar de codificación. Permite analizar el código en base a reglas predefinidas tales como documentación del código, convenciones de nombres de clases, métodos y ficheros, tamaños máximos de clases, duplicaciones de código, complejidad ciclomática, líneas de código sin comentar, dependencia entre clases y cantidad máxima de parámetros utilizados por métodos, entre otras. Asimismo permite a los usuarios crear otras reglas adaptadas a sus necesidades. Identifica hasta 50 categorías de errores, de las cuales el evaluador selecciona las que considere más apropiadas, y suma 140 tipos de errores. Al igual que las dos anteriores esta herramienta permite crear nuevas reglas.

UCDetector[4]: un detector de código innecesario. Detecta aquellas visibilidades públicas que por su confidencialidad deberían ser privadas. Detecta código muerto y, si bien permite crear reglas, el procedimiento resulta complejo.

IclEmma Java Code Coverage[5]: es una herramienta gratuita de testeo por cobertura para el entorno de desarrollo Eclipse. Se utiliza como complemento de JUnit. Analiza las líneas o fragmentos de código, detectando si fueron o no total o parcialmente cubiertos por las pruebas unitarias. Para ello discrimina las instrucciones cubiertas Cover Instructions y aquellas que no han sido analizadas Miss Instructions.

A fin de otorgar validez a la metodología, todas las herramientas consideradas en el estudio se ejecutaron bajo las mismas condiciones de hardware y software, a fin de garantizar que las distinciones observadas entre los resultados obtenidos se deben exclusivamente a las diferencias en las características propias de las herramientas.

IV. RESULTADOS

Como se ha indicado anteriormente, las herramientas incluidas en el estudio aplican diferentes criterios para la detección de errores y, consecuentemente no son comparables.

Si bien los tipos de errores abarcan diferentes categorías y niveles de gravedad, la cantidad total de errores detectados permiten observar el comportamiento de cada una de las herramientas estudiadas. En la Figura 1 se advierte un amplio rango de detección de errores entre las diferentes herramientas, concluyendo que BlueJ tiene menos errores de FindBug que de PMD. Esto podría deberse a que la herramienta PMD administra una mayor cantidad de reglas, pero la diferencia es aún mayor. Esto implicaría que el equipo de trabajo que desarrolla BlueJ utiliza FindBug como herramienta de análisis estático. Un estudio más detallado requiere una discriminación de los tipos de errores, con el propósito de ponderar su nivel de gravedad y brindar un elemento de juicio de mayor precisión.

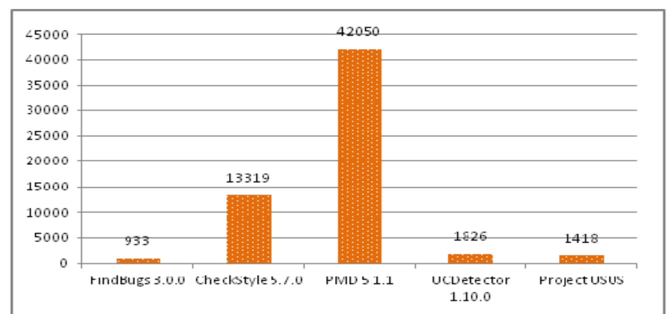

Fig. 1. Comparación de errores detectados en BlueJ, versión 3.1.1, con las distintas herramientas de estudio.

1 http://pmd.sourceforge.net/
2 http://findbugs.sourceforge.net/
3 http://checkstyle.sourceforge.net/
4 http://www.ucdetector.org/
5 http://www.eclemma.org/

La Figura 2 representa las tendencias en el comportamiento de cada una de las cinco herramientas de análisis estático de código fuente, consideradas en el estudio, aplicadas a las diez últimas versiones de BlueJ. En virtud de que el objetivo primordial de toda prueba de software es encontrar los errores, se considera que el hecho de detectarlos resulta ser una evidencia de la eficacia de las herramientas con las que se efectúan dichas pruebas. A partir de este principio, las pautas encontradas reflejan el mismo patrón de comportamiento a lo largo de las diez versiones de BlueJ, lo que confirma el grado de efectividad de las herramientas evaluadas.

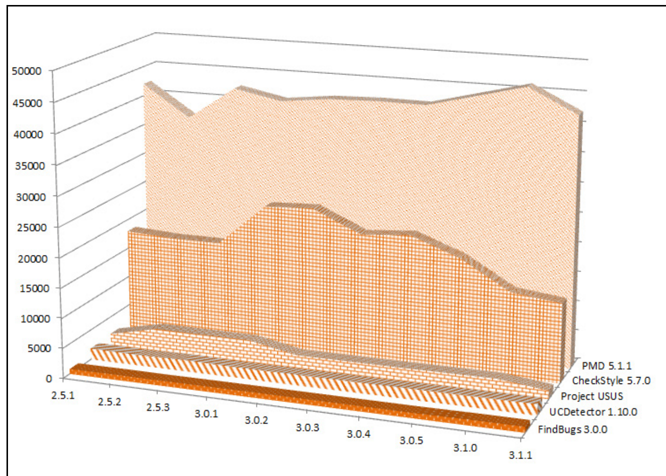

Fig. 2. Errores detectados en diez versiones consecutivas de BlueJ, con las distintas herramientas de estudio.

En función de la discriminación específica de los errores que presenta la herramienta FindBug, es de destacar la evolución que ha mostrado BlueJ a lo largo de los años considerados. Comparando los resultados se puede apreciar diferentes evoluciones en la calidad del desarrollo del BlueJ. Con respecto a las principales violaciones en las reglas de FindBug (Figuras 3, 4 y 5), hubo una disminución de su incidencia o una estabilización a lo largo de las diferentes versiones. Igualmente, existe una tendencia en alza en la versión 3.0.5 (que se extiende, para luego disminuir en el conjunto de reglas Scariest). Como puede apreciarse en la Figura 6, el total de errores de menor prioridad es considerablemente mayor en comparación con los otros tres conjuntos de errores. Por su parte el total de errores muestra una tendencia a incrementarse en las sucesivas versiones, conforme lo muestra la Figura 7.

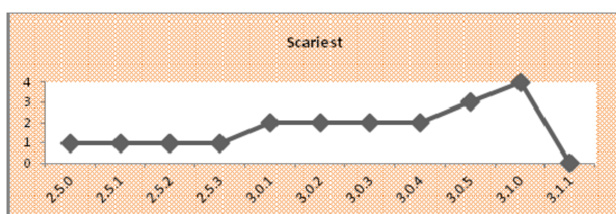

Fig. 3. Evolución de los Scariest de BlueJ detectados por FindBug en las diferentes versiones

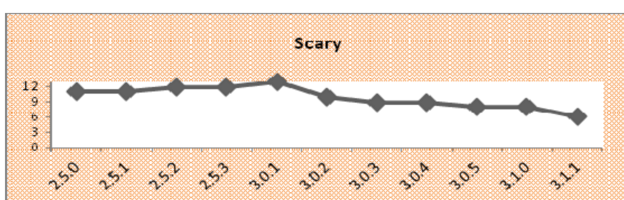

Fig. 4. Evolución de los Scary de BlueJ detectados por FindBug en las diferentes versiones

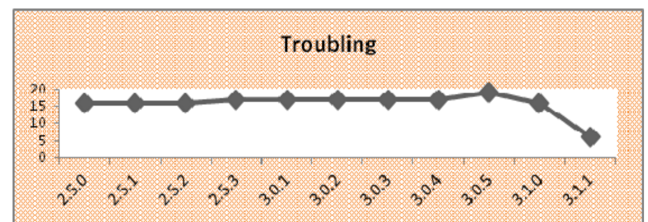

Fig. 5. Evolución de los Troubling de BlueJ detectados por FindBug en las diferentes versiones

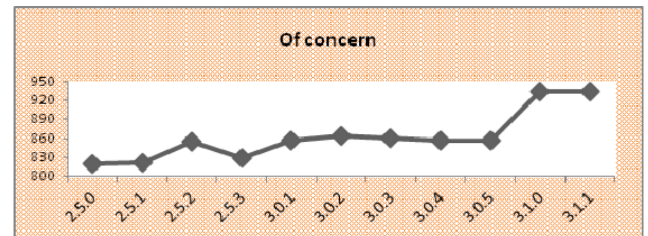

Fig. 6. Evolución de los Of concern de BlueJ detectados por FindBug en las diferentes versiones

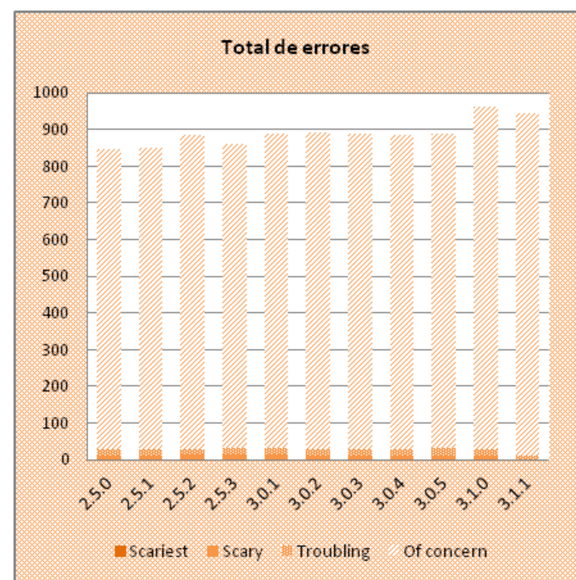

Fig. 7. Evolución en las violaciones de las reglas de FindBug en las diferentes versiones de BlueJ

V. CONCLUSIONES

El análisis desarrollado ha permitido extraer conclusiones tanto en lo referente al comportamiento de las herramientas como en lo vinculado a la metodología aplicada y a la evolución de las violaciones de las reglas de FindBug en BlueJ a lo largo de las versiones analizadas.

El hecho de que PMD y CheckStyle localizan diferentes problemas de código y mantienen las pautas de análisis, permite reconocer sus bondades y calificarlas como recomendables frente a las iniciativas de desarrollo de software. Es importante aclarar que mientras PMD busca posibles errores, tales como sentencias vacías, códigos muertos, expresiones complejas, códigos no óptimos o códigos duplicados, CheckStyle auxilia a los programadores en la creación de código acorde a los estándares de las buenas prácticas del desarrollo de software. Por su parte, IclEmma Java Code Coverage detecta la cantidad de código accedido por las pruebas de testeo, con la finalidad de identificar los sectores de código sin cobertura. Por esta razón se estima conveniente utilizarla como herramienta complementaria de evaluación de código fuente.

Confirmando la idea de Johnson, Song, Murphy-Hill y Bowdidge [13], se reconoce que la forma en que se presentan las advertencias, mediante la aplicación de las diferentes herramientas consideradas, dificultan su uso; por lo que se considera necesario identificar con mayor precisión el tipo de error detectado e idear un mecanismo interactivo para ayudar a los desarrolladores a solucionar esos defectos.

En cuanto a la extensibilidad de las herramientas evaluadas, es de destacar que tanto PMD como CheckStyle mantienen este atributo, dada la facilidad de crear nuevas reglas adaptadas a cada escenario en particular, hecho que las confirman como herramientas recomendables a los fines de testear el código fuente de un producto software.

A partir del diseño del procedimiento seguido a lo largo del trabajo se advierte la posibilidad de implementar innovaciones metodológicas adaptadas a las expectativas e intereses de los investigadores. La flexibilidad de los procesos por una parte potencia la creatividad pero podría atentar contra la validez del diseño. Si bien las normas ofrecen estándares de comparación, la diversidad de atributos limita las posibilidades y consecuentemente demanda nuevas estrategias metodológicas.

Por último, de acuerdo al análisis de la evolución de BlueJ en los cuatro tipos de violación de las reglas de FinBug, las violaciones de mayor riesgo tienden a disminuir mientras que las de menor riesgo se incrementan, revelando en definitiva una evolución positiva de BlueJ en las sucesivas versiones. Como trabajos futuros se continuará con el estudio de estas herramientas, incluyendo analizadores dinámicos, y se administrarán encuestas a equipos de desarrollo en empresas privadas. De esta manera se profundizará en el conocimiento del uso real que los equipos hacen de estas herramientas y de la evolución de las buenas prácticas en el tiempo.

Reconocimiento

El equipo investigador agradece a la Universidad Adventista del Plata por financiar el proyecto y permitir la concreción de la propuesta.

Referencias

[1] Lochmann, K., *Defining and Assessing Software Quality by Quality Models*. Tesis doctoral no publicada, Institut für Informatik der Technischen Universität München, Alemania, 2013.

[2] Escalona, M. J., Pérez Pérez, M., González Barroso, O., Ponce, J., Correa, M. y Merino, A. I., “Revisiones de código en el contexto del aseguramiento de calidad. Un caso práctico”, Revista Española de Innovación, Calidad e Ingeniería del Software, vol. 4, nº 2, pp. 46-57, 2008.

[3] Greiner, C.; Dapozo, G.; Acosta, J.; Domínguez, M.; Chiapello, J. y Estayno, M., “Persistencia de mediciones como apoyo a la gestión de proyectos de software”. En: XV Workshop de Investigadores en Ciencias de la Computación, 2013. Recuperado el 1 de julio de 2014 de http://sedici.unlp.edu.ar/bitstream/handle/10915/27259/Documento _completo.pdf?sequence=1

[4] Chidamber S.R. y Kemerer C.F., “A metrics suite for object-oriented design”, IEEE Transactions on Software Engineering, vol. 20, nº 6, pp. 476-493, 1994.

[5] Kitchenham B.A., Hughes R.T. and Kinkman S.G., “Modeling Software Measurement Data”, IEEE Transactions on Software Engineering, vol. 27, nº 9, pp. 788-804, 2001.

[6] Fenton N. E. y Kaposi A. A., “Metrics and software structure”, Information and Software Technology, vol. 29, pp. 301-320, 1987.

[7] Poels G. y Dedene G., “Distance-based software measurement: necessary and sufficient properties for software measures”, Information and Software Technology, vol. 42, pp. 35-46, 2000.

[8] Weyuker, E.J., “Evaluating Software Complexity Measures”, IEEE Transactions on Software Engineering, vol. 14, nº 9, pp.1357-1365, 1988.

[9] Reformat M., Pedrycz W. y Pizzi N.J., “Software quality analysis with the use of computational intelligence”, Information and Software Technology, vol. 45, pp. 405-417, 2003.

[10] Fontana, F. A., Braione, P. y Zanoni M., “Automatic detection of bad smells in code: An experimental assessment”, Journal of Object Technology, 2011. Recuperado el 4 de diciembre de 2013 de: http://www.jot.fm

[11] Lamas Codesido, Isabel., Comparación de analizadores estáticos para código java. Proyecto de investigación. Universidad Oberta de Catalunya, 2011. Recuperado el 13 de mayo de 2014 de http://openaccess.uoc.edu/webapps/o2/bitstream/10609/6145/1 /mlamascTFC012011.pdf

[12] Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J. y Zhou, Y. “Evaluating Static Analysis Defect Warnings On Production Software”, In Proceedings of PASTE 07. (San Diego, June 13-14, 2007). pp 1-8. doi 10.1145/1251535.1251536

[13] Johnson, Brittany, Song, Yoonki, Murphy-Hill, Emerson y Bowdidge. Robert. 2013. “Why don't software developers use static analysis tools to find bugs?” In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 672-681.

[14] Rutar, N., Almazan, C. B., y Foster, J. S. A Comparison of Bug finding tools for Java. En: Software Reliability Engineering, 5th IEEE International Symposium on ISSRE, pp. 245-256, 2004.

[15] Wagner, S., Jürjens, J., Koller, C. y Trischberger, P. Comparing bug finding tools with reviews and tests. In Proc. 17th International Conference on Testing of Communicating Systems, pp. 40–55, 2005.

Marisa Cecilia Tumino. Ingeniera en Recursos Hídricos, Analista en Informática Aplicada, Magister en Ciencias Computacionales, Doctora en Educación con énfasis en Administración educativa. Desempeño profesional en (a) Instituto Francisco Ramos Mejía, (b) Universidad Adventista del Plata, (c) Instituto Juan Bautista Alberdi- Misiones, (d) Universidad de Montemorelos, México, (e) Herbert Fletcher University, Puerto Rico y (f) Universidad de Chillán, Chile.

Juan Bournissen. Doctorando en Tecnologías Educativas: E-learning, y Gestión del Conocimiento en la Universidad de Islas Baleares, España. Master en Ingeniería del Software obtenido en la Universidad Politécnica de Madrid. Magíster en Ingeniería del software obtenido en el Instituto Tecnológico de Buenos Aires, ITBA. Especialista en Entornos Virtuales del Aprendizaje. Profesor Universitario en Sistemas de Información. Ingeniero en Sistemas de Información. Analista Universitario en Sistemas. Profesor universitario de grado y posgrado en la modalidad presencial y a distancia. Administrador de plataformas virtuales en instituciones estatales y privadas. Formador de formadores en e-learning en Argentina y en varios países de Latinoamérica. Director de centro de cómputos. Asesor informático en sistemas de información y en tecnologías educativas y e-learning.

Gisela Vanina Müller. Licenciada en Sistemas de información. Magíster en Ingeniería del Software. Directora de la licenciatura en Sistemas de Información en la Universidad Adventista del Plata. Docente en Sistemas de Información. Directora del Instituto de Informática y Sistemas.

Reinhardt Schmidlin. Estudiante de la licenciatura en Sistemas de información en la Universidad Adventista del Plata. Desarrollador web en Inter-European Division. Becario en el Instituto de Informática y Sistemas.

Emanuel Irazábal. Ingeniero Superior en Informática. Master oficial en Tecnologías de la Información y Sistemas Informáticos. Doctor en Informática. Consultor de Calidad KYBELE CONSULTING S.L. Investigador en Universidad Rey Juan Carlos. Escuela Superior de Ciencias Experimentales y Tecnología. Dpto. Lenguajes y Sistemas Informáticos II. Profesor Asociado. Consultor Experto en Pruebas en KOTASOFT S.L. Experto en pruebas y analista Funcional en REDPRINT S.L.