Universitat Oberta de Catalunya www.uoc.edu	Máster oficial de software libre.
	Proyecto de investigación básica o aplicada.
	PEC 3 Artículo conclusiones.
	Enero 2011

Comparación de analizadores estáticos para código java

Isabel Lamas Codesido
Tutor: Robert Clarisó Viladrosa

Contenido

1. Resumen.

El **objetivo** principal de este artículo es la selección y comparación de dos herramientas de análisis estático para java, esta tarea necesita de estudiar previamente el estado del arte de estos analizadores, ver que características son deseables para este tipo de analizadores y finalmente compararlas en ejecución sobre los dos proyectos de software libre elegidos argoUML y openProj. Se compara FindBugs con PMD dos analizadores que pueden utilizarse con la versión 1.6. de JDK. Los resultados de la comparación nos permiten deducir que los analizadores se complementan en cuanto a bugs detectados, hay pocos solapamientos. Como conclusiones podemos decir que la búsqueda de bugs necesita de más de una herramienta de análisis estático.

Palabras clave. Herramientas de análisis estático, FindBugs, PMD, plugins Eclipse, style checker, bug checker, análisis flujo de datos.

2. Introducción.

Uno de los mayores retos a los que se enfrenta la ingeniería del software en los últimos años es el creciente tamaño, seguridad y protección necesaria para los proyectos del software que se desarrollan. Esta situación necesita de herramientas que ayuden a realizar análisis de código en diferentes etapas de desarrollo. Para esto tenemos dos tipos de herramientas, las estáticas y las dinámicas. Las primeras permiten realizar un análisis de fallos (*bugs*) sin ejecutar el código y se pueden utilizar desde las primeras fases del ciclo de vida. Están enfocadas a optimizar el código y a su validación. Las segundas, conocidas como herramientas dinámicas o de testing se utilizan en las últimas fases del ciclo de vida. El conjunto de herramientas trata de prevenir errores lo que resulta fundamental de cara a la obtención de un producto de calidad.

En este artículo vamos a centrarnos en las herramientas estáticas libres para código java. Para seleccionar las herramientas partimos de lo que hemos considerado las características deseables de una herramienta de análisis estático, usabilidad, documentación, técnicas de análisis, etc.

Una vez descritas las características y los tipos de bugs deseables la investigación se centra en ver el estado del arte de las herramientas estáticas para java, que artículos e investigaciones pueden ayudar a esta investigación.

Los analizadores que hemos elegido después de ver el estado del arte son FindBugs y PMD, se ha descartado ESC/Java 2 por no poder utilizarse en la versión 1.6 de JDK.

FindBugs que es un analizador que realiza análisis del flujo de datos intraprocedural utilizando la librería BCEL y PMD que utiliza la técnica de análisis style checkers.

Para poder comparar estas herramientas utilizamos dos proyectos de software libre argoUML de 202395 líneas de código y OpenProj de 48169 líneas de código.

3. Posibles criterios de valoración en una herramienta de análisis estático.

La mayoría de los documentos estudiados muestran muchas características deseables para una herramienta estática, sin embargo, siguiendo el campo de investigación del technical report de *Nicole et al*[5] citaremos aquellas cualidades o criterios que pueden ser de mayor interés para realizar la valoración de cada herramienta estática:

1. **Tipo de licencia.** Para una empresa pequeña, el importe de una licencia es un tema importante para decidirse por un producto u otro.
2. **Usabilidad.** El desarrollador tiene que encontrar en la herramienta facilidad de instalación, de uso y de obtención de resultados. Para valorar este criterio tendremos en cuenta 5 categorías: la ayuda de la herramienta y la documentación, el tiempo de instalación, el tiempo de configuración del entorno, las actualizaciones y la facilidad para interpretar los resultados que aborda la herramienta. Para convertir estos criterios en objetivos, valoramos cada categoría del 1 a 10, siendo 1 el valor más pésimo y el 10 el mejor valor de los posibles.
3. **Eficiencia.** Veremos el tiempo de ejecución y el consumo de recursos de cada herramienta.
4. **Extensibilidad.** En el sentido de que permita añadir reglas fácilmente, si la herramienta puede extenderse podremos adaptarla a nuestro proyecto y en consecuencia no habrá límite en la cantidad de bugs que pueden ser detectados. Se valorará que la extensión de reglas se realice fácilmente con un lenguaje sencillo (XPath, java, JML). Valoraremos también la existencia de editores especiales de reglas propios de cada herramienta.
5. **Precisión.** Compararemos en este punto el número de bugs comunes y los que encuentra cada uno de los dos analizadores que se comparan.
6. La **técnica de análisis** según *Nicole et al*[5]:
   - ***Análisis basado en AST.*** La mayoría de las herramientas que se estudian en este documento, no analizan el código fuente tal y como está escrito, en su lugar transforman el código en una representación en árbol (AST Abstract Syntax Tree, Árbol de Sintaxis Abstracta) que refleja la estructura del fichero. Como se detalla más adelante, Checkstyle utiliza ANTLR, FindBugs la librería BCEL.
   - ***Análisis dataflow***. Es la técnica más utilizada en el análisis estático, la definición que aparece en el libro de *de Aho et al*[1] dice que *un esquema de análisis de flujo de datos define un valor en cada punto en el programa. Las instrucciones del programa tienen funciones de transferencia asociadas que relacionan el valor antes de la instrucción con el valor después de la misma.*

     Dentro de este tipo de análisis podemos encontrar varias categorías:

     - ***Análisis intraprocedural vs interpocedural.*** Un análisis intraprocedural opera a través de un método o procedimiento, mientras que un análisis interprocedural opera en todo el programa entre diferentes métodos o procedimientos.
     - ***Análisis context-sensitive vs context-insensitive.*** En este caso lo que contabilizamos es el número de llamadas a un procedimiento, método o función. En el siguiente ejemplo, un análisis context-sensitive distingue dos llamadas al método Resultado (llamada sum= 11 y llamada sum=16), sin embargo, un análisis context-insensitive contaría una única llamada (sum toma valores variables).

```
public int Resultado(int x) {
 y=6;
 return x=x+y;
}
Public Calcular(){
..
sum = Resultado(5);
…
sum = Resultado (10);
}
```

- ***Análisis path-sensitive vs path-insensitive.*** Los analizadores path-sensitive tienen en cuenta las distintas alternativas que podemos tener en una sentencia de decisión o bifurcación, mientras que los analizadores path-insensitive no tendrían en cuenta las dos alternativas (x no es constante). Evidentemente, los primeros resultan mucho más caros de implementar que los segundos:

- ***Backward dataflow analysis vs Forward dataflow analysis.*** Estas técnicas se utilizan cuando es necesario investigar el flujo de datos en dos sentidos (forward y backward). La técnica forward analiza el código sobre el comportamiento pasado, alcance de las definiciones o expresiones definidas, mientras que en la técnica backward el análisis intenta predecir el comportamiento futuro, expresiones muy ocupadas o variables sin vida. Se pueden utilizar para descubrir redundancias en el código.

- ***Análisis style-checkers***. Se centran exclusivamente en la estructura léxica y sintáctica. Con este tipo de analizadores se detectan espacios en blanco, código demasiado extenso, nombres de variables que no siguen las convenciones indicadas, líneas de código mal indentadas.

- ***Análisis bug-checker***. Método basado en búsqueda de patrones o reglas (pattern matching) predefinidos por la herramienta o diseñados por el usuario de la misma. Se utilizan más que los style-checkers. Ejemplos que veremos son: FindBugs, PMD y JLint.

- ***Theorem proving***. Demostración del teorema. Construye una prueba de los requerimientos mediante inducción lógica sobre la estructura del programa. ESC/Java2 es un ejemplo de analizador de este tipo.

7. **Frameworks disponibles e integración en IDEs.** A mayor número de plugins más posibilidades de utilizarla en distintas plataformas.

4. Estado del arte. Lo que otros escribieron.

4.1.Findbugs.

Esta herramienta, desarrollada inicialmente en la universidad de Maryland por Bill Pugh y David Hovemeyer autores del artículo [7], se ha convertido en un proyecto de SourceForge.net protegida con licencia *LGPL* (Lesser GNU Public License). Es uno de los analizadores más utilizados y mejor documentados de las que se han estudiado[15], en agosto del 2009 se ha bajado 287.580 veces.

Utiliza la librería BCEL(Apache Byte Code Engineering Library) para generar el grafo de control de flujo CFG (Control Flow Graph) y realizar el *análisis de flujo de datos intraprocedural,* así consigue encontrar fallos del tipo null pointer, bucles infinitos, sentencias IF que pueden apuntar a un puntero nulo, tipos que son garantía de una excepción ClassCastException, valores de retorno ignorados[4]. El análisis que realiza está basado en el concepto de *bug pattern* por lo que también se puede clasificar como *bug checker.*

FindBugs [15] analiza el bytecode para encontrar posibles problemas de eficiencia y malas prácticas de codificación en aplicaciones Java. Cada *bug pattern* se puede agrupar en uno de los siguientes grupos:

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

Además, organiza el informe de incidencias en varias categorías según su gravedad[15], a cada informe de bug se le asigna una prioridad: *alta, media y baja*. En condiciones normales FindBugs no reporta los fallos de prioridad baja. Estos listados de fallos pueden exportarse a ficheros XML para su uso en bases de datos.

La herramienta ha evolucionado mucho desde que la lanzaron sus autores[3], inicialmente tenía implementadas 45 reglas frente a los 300 bugs que tiene predefinidas en sus últimas versiones.

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, en el artículo de *Cole et* al[4] tenemos la descripción de los pasos necesarios para realizar esta tarea:

1. Crear un detector. El detector crea un análisis dataflow y lo usa como si fuese un AST.
2. Crear un análisis dataflow (forward and backward analysis).
3. Crear un **fact** sino existe.
4. Instanciar un dataflow en el detector.

Para ejecutarla podemos utilizar Ant, Maven, swing GUI[4], desde la línea de comandos o integrado en Eclipse o en Netbeans.

Los autores de los documentos de *Nicole et al*[5] *Meng*[8] *Nick Rutar et al*[9], concluyen que esta herramienta resulta ser más completa en cuanto a número de fallos, tiempo de ejecución, usabilidad e IDEs disponibles que otras como Crystal, PMD, Soot, Bandera, Coverity, Jlint y ESC/Java 2.

Limitaciones Findbugs.

Algunas de las limitaciones más importantes de Findbugs están documentadas en *Nicole et al*[5]:

- La herramienta no proporciona **path-sensitive analysis** aunque sería posible implementarlo.
- Tampoco proporciona un mecanismo estándar de **análisis meta-datos** que permite obtener mejores resultados.
- En lo referente al tipo de análisis que realiza, este analizador no realiza un análisis sensitivo de contexto interprocedural.
- Otra limitación a tener en cuenta es que el análisis que realiza no incluye la documentación javadoc.

4.2.PMD.

Tal y como aparece en la página oficial de PMD[17], es una herramienta de código libre, bien documentada, con licencia *BSD* (Berkeley Software Distribution), que pertenece al conjunto de proyectos de SourceForge.net. Es un analizador que utiliza la técnica de *análisis style checkers.*

PMD no analiza el código fuente del aplicativo de manera directa, en su lugar, parsea dicho código para generar un AST (Abstract Syntax Tree). Su funcionamiento se basa en detectar patrones, los cuales son posibles errores que pueden aparecer en tiempo de ejecución.

PMD escanea el código fuente Java y busca fallos potenciales como:

- Posibles defectos. Sentencias try/catch/finally/switch vacías.
- Código muerto. Variables, parámetros y métodos no utilizados.
- Código no óptimo. Uso ineficiente del StringBuffer, etc.
- Expresiones innecesarias. Sentencias *if* innecesarias, bucles *for* que pueden ser de tipo *while*.
- Código duplicado.

El analizador trae definidos un conjunto de reglas agrupado en *ruleset* temáticos, que pueden extenderse con el módulo *PMD Rule Designer* siguiendo este procedimiento*:*

1. Escribir un trozo de código que incumple la regla X que queremos definir.
2. Generar con *PMD Rule Designer* el árbol AST.
3. Escribir la expresión que encuentre los nodos del árbol AST que incumplen la regla. Para esto podemos utilizar *XPath* o *métodos java*.
4. Generar con la herramienta el código XML para poder incluirla dentro de las reglas *ruleset.* El XML incluye campos como nombre de la regla, clase y sus propiedades.

PMD incluye además, otro módulo conocido como CPD *Copy Paste Detector*, el cual permite detectar código duplicado existente en el programa. Al copiar y pegar código debemos tener en cuenta que también se copian y pegan los defectos de este, el módulo CPD nos permite detectar:

- Número de líneas y tokens repetidos.
- Archivos que contienen el fragmento repetido y línea en la que comienza el código repetido dentro de cada uno de los archivos.

Los fallos están clasificados en 5 niveles: high error, error, high warning[11], warning e información. El formato del informe de salida depende de la librerías utilizada, si utilizamos por ejemplo, iText podemos obtener un archivo RTF con el fallo, línea, mensaje y nivel de criticidad.

PMD [17] tiene plugins para Ant, Maven, Eclipse, JDeveloper, JEdit, JBuilder, BlueJ, CodeGuide, NetBeans/Sun Java Studio Enterprise/Creator, IntelliJ IDEA, TextPad, Gel, JCreator, y Emacs.

Limitaciones.

Tres limitaciones a destacar:

- No realiza análisis de flujo de datos.
- No soporta análisis meta-data.
- No puede ejecutarse en más de un fichero de cada vez.

4.3. ChekStyle.

CheckStyle es una herramienta con licencia *GPL*[13] que analiza el código fuente y que al igual que PMD y FindBugs pertenecen al conjunto de proyectos de software libre de SourceForge.net. CheckStyle automatiza el proceso de chequear el código Java frente a un estándar de codificación como el de Sun Code Conventions, liberando al desarrollador de esta tediosa pero importante tarea.

La herramienta es altamente configurable y puede soportar casi cualquier estándar de codificación. Es una herramienta que permite seleccionar sobre qué convenciones se quiere generar la alarma y sobre cuáles no.

Los chequeos estándar de CheckStyle se aplican al estilo general de codificación Java y no necesitan de librerías externas, según el autor de *Seelhofer*[10], pertenece al grupo de herramientas *style checkers.* Sin embargo, existen además chequeos opcionales por ejemplo para J2EE.

Si las reglas de validación proporcionadas por este analizador no cubren nuestras necesidades, podemos crear nuestras propias reglas de validación. Para ello, proporciona una herramienta que nos muestra la estructura en árbol de una clase Java permitiéndonos identificar cada uno de sus nodos y utilizarlos para describir la nueva regla.

Al igual que PMD, no analiza el código fuente del programa de manera directa, sino que transforma el código fuente en una representación en árbol (AST) que refleja la estructura del fichero. Para realizar esta función utiliza un parseador/generador llamado *ANTLR* que generará el árbol AST (Abstract Syntax Tree).

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

- Comentarios Javadoc.
- Convenciones en nombres y cabecera de los archivos.
- Sentencias import.
- Tamaño del código.
- Espacios en blanco y bloques.
- Diseño de clases.
- Código duplicado.
- Chequeo de métricas.

Los plugins más conocidos son: Eclipse-CS, Checklipse (licencia Mozilla Public License MPL), Checkstyle-idea, JetStyle, Checkstyle Beans, nbCheckStyle, JBCS y jbCheckStyle.

Es una herramienta bien documentada, con página propia en [13].

Una ventaja de CheckStyle frente a PMD es que la primera encuentra fallos en la documentación javadoc, mientras que la segunda los ignora.

Limitaciones.

- No hace un análisis del flujo de datos (dataflow).

4.4. JLint.

JLint es una herramienta de licencia *LGPL* disponble en la página [16], realiza *análisis de flujo de datos, de sincronización de threads y de herencia*. Al igual que pasaba con FindBugs, analiza el bytecode de java para realizar el análisis, sin embargo, tal y como se menciona en el artículo de *Rutar et* al[9] JLint también incluye un componente que permite realizar el análisis interprocedural[9].

El analizador está compuesto por dos programas:

- **AntiC** que realiza la verificación sintáctica. Es un programa que realiza análisis léxico y sintáctico de ficheros Java (también de C y C++).
- **JLint** que efectúa la verificación semántica analizando ficheros .class. Puede detectar errores de sincronización de threads, problemas que pueden darse en la jerarquía de herencia, errores en el flujo de datos.

Para realizar el análisis hace *dos pasadas*: en la primera se tratan los métodos de forma modular con interpretación abstracta y en la segunda se realiza el análisis de interbloqueos (deadlock).

Entre los fallos que puede detectar podemos distinguir:

- Errores de sintaxis con mensajes como carácter octal esperado (System.out.println("\128")), secuencia de escape incorrecta (System.out.println("\Uabcde:")).
- Errores de sincronización de threads. Deadlocks y race conditions. El primer tipo se da cuando se bloquean demasiados recursos y todos los procesos quedan bloqueados, el segundo caso aparece cuando no se utiliza un bloqueo donde es necesario, los threads pueden interferirse entre sí y el resultado depende del orden de ejecución de cada thread.
- Errores de herencia. Métodos no sobreescritos en las clases derivadas, campos en la clase derivada con el mismo nombre que en la clase base.

En las comparaciones realizadas con FindBugs, ESC/Java y PMD se puede concluir según *Rutar et al* [11] que JLint es extremadamente rápido, incluso con proyectos grandes.

Limitaciones.

- Los autores *Rutar et al*[9] comentan que este analizador es difícil de extender.

4.5. ESC/Java.

ESC/Java fue desarrollado en el centro de investigación de los sistemas de Compaq (SRC). SRC puso en marcha el proyecto en 1997, extendiendo el analizador ESC/Modula-3. En 2002, SRC lanzó el código fuente para ESC/Java y relacionó las 2 herramientas.

La versión ESC/Java2 está bajo licencia *Java Programming Toolkit Source Release* (SRC) *pseudo-open source.*

La herramienta utiliza para el análisis el método *Theorem prover* donde los desarrolladores pueden controlar los tipos de fallos que detecta la herramienta utilizando anotaciones/aserciones en el código java (diseño por contrato en java) que son codificadas en lenguaje *JML (Java Modeling Language)* al estilo de Hoare, con precondiciones, postcondiciones e invariantes. Estas anotaciones se conocen como *pragmas* y pueden expresarse:

- En una única línea: `//@ requires a != null [13].`
- En múltiples líneas. `/*@ .. @*/.`
- Anotaciones embebidas en javadoc.

Dentro del análisis estático las anotaciones más conocidas son:

- `@requires P.` Introduce una notación para especificar una precondición.
- `@ensures Q.` Especifica una postcondición que debe cumplirse después de ejecutar el código.
- `@invariant Z.` Invariante.
- `@assignable.` Atributos que pueden cambiar de valor.
- `@signals.` Permite especificar las excepciones que pueden darse. Los comentarios de Java se interpretan como anotaciones de JML cuando comienzan con @.

Una posible clasificación de los fallos que puede encontrar el analizador aparecen en el artículo de *Selfhofer*[10] sería:

- Posibles excepciones en tiempo de ejecución. Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore.
- Violaciones de la especificación JML. Precondiones, postcondiciones incorrectas.
- Violaciones non null. NonNull, NonNullInit. Aplicable a campos, parámetros, valores de retorno, variables locales..
- Especificaciones de bucles y flujo. Ejemplos de estos warning son: assert, reachable, loopInv, DecreasesBound.
- Avisos de clases invariantes. Invariant, Constraint, Initially.
- Avisos de excepciones. Exception.

Es una herramienta fácil de utilizar, que se puede ejecutar desde la línea de comandos o bien integrada en el IDE Eclipse o Netbeans.

Convierte la salida del theorem prover en un mensaje comprensible que los programadores pueden usar para corregir el problema.

Limitaciones.
Dos limitaciones a destacar:

- Al igual que pasa con Coverity Prevent, los mensajes de descripción de fallos pueden resultar complicados de entender por los desarrolladores *Peck*[2].
- Un fallo puede referirse a varios ficheros de código ESC/Java.

5. Metodología.

Una vez seleccionados los dos analizadores hemos utilizado argoUML y openProj para compararlos:

- **OpenProj**, una herramienta alternativa al conocido Microsoft Project que permite gestión y mantenimiento de proyectos. Se distribuye gratuitamente bajo licencia Common Public Attribution License Version (CPAL). La herramienta permite controlar todos los aspectos referentes a la gestión de proyectos, como la planificación, programación y gestión de recursos. Se trata de un proyecto sólido de **48169 líneas de código** en su versión 1.4 disponible en [19].

- **ArgoUML**, una herramienta utilizada para desarrollar diagramas UML publicada bajo licencia BSD. El proyecto se inició en el 2003 y cuenta en su versión actual 0.29.3.201001020415 con **202395 líneas de código** disponible en [20].

Los principales objetivos que se persiguen con esta evaluación son: fundamentar las decisiones de elección de una herramienta u otra y demostrar que la elección de una herramienta no es suficiente para poder detectar todos los bugs que tenemos dentro de un programa o aplicación desarrollada.

Las pruebas que se realizan están encaminadas a evaluar la eficiencia de los dos analizadores bajo los criterios establecidos en la sección 3 de este documento.

5.1. Plataforma pruebas.

Para realizar las pruebas se parte de lo siguiente:

- PC HP Compaq Pentium IV CPU 3,20 GHz y 3,24 GB de RAM.
- El software base para la prueba es un Debian GNU/Linux 5.0 con Kernel 2.6-26-2-686.
- Eclipse. IDE de pruebas es la versión 3.4.2 bajado de http://www.eclipse.org/platform.
- ESC/Java versión plugin 0.1.2 de Michael Peck. Se descartó por no ser compatible con la versión 1.6. de JDK.
- FindBugs 1.3.9.20090821 de edu.umd.cs.findbugs.plugin.eclipse.
- PMD plugin eclipse versión 4.2.5.
- XQuery. Motor Qizx/Open licencia MPL versión 4.1.
- Creamos dos proyectos dentro de Eclipse, argoUML que necesita de varias librerías para poder compilarse y ejecutarse disponibles en [20] y openProj para alojar el código fuente de openProj.

5.2. Instalaciones.

Se instala java siguiendo las instrucciones que aparecen en la página oficial de sun, instalamos Eclipse descomprimiendo el fichero .tar.gz descargado del sitio de Eclipse.org. Además se instala como plugins en Eclipse los dos analizadores elegidos para el artículo, los dos analizadorse explican en sus páginas el proceso de instalación:

- Para instalar FindBugs en Eclipse solo tenemos que ir al menú `Help -> Software Updates -> Available Software -> Add Site` añadir nuevo sitio http://findbugs.cs.umd.edu/eclipse, seleccionamos FindBugs e Instalamos.

- Para instalar el plugin PMD hacemos lo mismo pero con el sitio oficial de PMD http://pmd.sourceforge.net/eclipse, en el cuadro de diálogo *Help -> Software updates -> Available Software -> Add Site,* marcamos el componente correspondiente dependiendo de la versión de Eclipse y pinchamos en Install.

Tras la instalación tenemos dos nuevas perspectivas para poder ver los resultados de su ejecución disponibles en `Window -> Open perspective -> Other (FindBugs o PMD).`

Los dos analizadores generan sus estadísticas en ficheros XML. Para generar estas estadísticas se utiliza el menú de contexto de FindBugs (Save XML) y el de PMD (Generate Reports). Para manipular los ficheros XML generados utilizaremos el motor QIZX/OPEN desarrollado en java, con licencia MPL que permite ejecutar sentencias de XQuery. Los ficheros se descargan de la página [21] y la instalación solo necesita que se descomprima el fichero en una carpeta del equipo.

5.3. Usabilidad de FindBugs y PMD en Eclipse.

Para valorar la usabilidad de las dos herramientas según el punto 3 de este documento tenemos en cuenta los criterios que indicamos a continuación:

5.3.1. Documentación y ayuda disponible.

En los dos casos tenemos toda la documentación en la página oficial de cada herramienta. Los dos analizadores tienen la documentación actualizada [15] [17].

5.3.2. Tiempo de instalación y actualizaciones posteriores.

El tiempo de instalación es el mismo para las dos herramientas, las dos aprovechan la funcionalidad de Eclipse que permite indicar la web del sitio de donde bajarse las instalaciones y posteriores actualizaciones.

Las actualizaciones se hacen de forma automática al abrir Eclipse y conectarse con la página indicada de actualización de cada analizador.

5.3.3. Configuración del entorno.

La configuración de los bugs que nos interesan se raliza en las dos herramientas en el menú *Window -> Preferences*.

- FindBugs. En *Window -> Preferences – java – FindBugs* podemos ver que FindBugs distingue 3 niveles de criticidad, fast, slow y moderate y 10 categorías de bugs: Malicious code vulnerability, dodgy, Bad practice, Dogus randon noise, Correctness, Internationalization, Performance, Security, Multithreaded correctness y Experimental. En la sección de Reporter configuration del mismo menú podemos seleccionar las categorías de bugs que pasarán al documento XML.

- PMD. PMD agrupa los bugs en RuleSet temáticos, para poder configurarlos tenemos la opción Rules Configuration dentro de *Window -> Preferences – PMD*. Nos permite utilizar un editor propio para poder añadir, modificar, copiar y eliminar reglas de detección de bugs. FindBugs no presenta un editor de reglas de este tipo.

5.3.4. Ejecución de analizadores.

Para ejecutar FindBugs en un proyecto abierto en Eclipse solo tenemos que seleccionar la carpeta del proyecto y con el menú de contexto ejecutar el comando *Find Bugs -> FindBugs*. Para ejecutar PMD utilizaremos el menú de contexto *PMD -> Check code with PMD*. Los ficheros XML desde FindBugs se extraen con el comando *Save XML* y en PMD con el comando *Generate reports*.

5.3.5. Interpretación de resultados.

Para poder interpretar los resultados de cada herramienta, vamos a observar el comportamiento de cada analizador en una línea de código. Hemos elegido la línea 100 del fichero MutableNodeHierarchy.java.

Utilizamos XQuery sobre el fichero XML que genera FindBugs. En ese fichero nos aparecen 5 Bugs, para la línea 100 se detecta un solo bug:

```
declare variable $FBdocs := collection("usr/qiz/qizxopen-
4.1/FindBugs/openProj/Fi*.xml");
for $a in $FBdocs//BugCollection/BugInstance
where $a/SourceLine/@sourcefile="MutableNodeHierarchy.java"
order by $a/@priority
return <BugTipo> {$a/@* except $a/@first except $a/@abbrev, $a/SourceLine)
</BugTipo>
```

```
<Bugs>
<BugTipo type="MS_SHOULD_BE_FINAL" priority="1"
category="MALICIOUS_CODE"><SourceLine
classname="com.projity.grouping.core.hierarchy.MutableNodeHierarchy"
start="100" end="100" sourcefile="MutableNodeHierarchy.java"
sourcepath="com/projity/grouping/core/hierarchy/MutableNodeHierarchy.java"/
></BugTipo>
</Bugs>
```

Este bug se puede localizar por la línea de código, clase y fichero al que pertenece y clasificarse por su tipo ***MS_SHOULD_BE_FINAL*** y categoría ***MALICIOUS_CODE*** con prioridad 1. Lo que viene a decir que la variable declarada debería de ser final (y no lo es). En la descripción que nos muestra FindBugs aparece el texto "*A mutable static field could be changed by malicious code or by accident from another package. The field could be made final to avoid this vulnerability*." Esta descripción no se descarga en el fichero XML que se genera.

La línea 100 de ese fichero corresponde con este código:

```
96 /**
97  * A map that holds the parent-children relationship.  Also implements TreeModel so it can be used to generate
98  * trees, such as in outline cells or popup trees.
99  */
100 public class MutableNodeHierarchy extends AbstractMutableNodeHierarchy{
101     public static int DEFAULT_NB_END_VOID_NODES=50;
102     public static int DEFAULT_NB_MULTIPROJECT_END_VOID_NODES=1;
103
104     private final Node root=NodeFactory.getInstance().createRootNode();
105
106
107
108     protected int nbEndVoidNodes = DEFAULT_NB_END_VOID_NODES;
109     protected int nbMultiprojectEndVoidNodes = DEFAULT_NB_MULTIPROJECT_END_VOID_NODES;
110     public MutableNodeHierarchy() {
111     }
```

PMD con la xquery equivalente a la del punto anterior localiza 144 bugs distribuidos en el mismo fichero. Para la línea 100 tenemos 4 bugs:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<violation beginline="100" endline="100"
rule="LongVariable" ruleset="Naming Rules"
package="com.projity.grouping.core.hierarchy" class="MutableNodeHierarchy"
externalInfoUrl="http://pmd.sourceforge.net/rules/naming.html#LongVariable"
priority="3">
Avoid excessively long variable names like
DEFAULT_NB_MULTIPROJECT_END_VOID_NODES
```

```
</violation>

<violation beginline="100" endline="100" rule="SuspiciousConstantFieldName"
ruleset="Naming Rules" package="com.projity.grouping.core.hierarchy"
class="MutableNodeHierarchy"
externalInfoUrl="http://pmd.sourceforge.net/rules/naming.html#SuspiciousCon
stantFieldName" priority="3">
The field name indicates a constant but its modifiers do not
</violation>

<violation beginline="100" endline="100" rule="VariableNamingConventions"
ruleset="Naming Rules" package="com.projity.grouping.core.hierarchy"
class="MutableNodeHierarchy"
externalInfoUrl="http://pmd.sourceforge.net/rules/naming.html#VariableNamin
gConventions" priority="1">
Variables should start with a lowercase character
</violation>

<violation beginline="100" endline="100" rule="VariableNamingConventions"
ruleset="Naming Rules" package="com.projity.grouping.core.hierarchy"
class="MutableNodeHierarchy"
externalInfoUrl="http://pmd.sourceforge.net/rules/naming.html#VariableNamin
gConventions" priority="1">
Variables that are not final should not contain underscores (except for
underscores in standard prefix/suffix).
</violation>
```

Los bugs de PMD quedan localizados por la línea de código, clase, fichero y paquete a la que pertenece. Quedan clasificados por la regla **`VariableNamingConventions y`** ruleset `Naming Rules`. Se añaden además una breve descripción del bug y un enlace externo http dentro de la página oficial de PMD con toda la información disponible y actualizada.

La descripción que nos da PMD de cada Bug puede verse desde *`Window -> Preferences – PMD.`*

PMD nos indica que estos bugs están en la línea 100 cuando en realidad están dentro de la clase, pero no específicamente en la línea 100. Desde Eclipse podemos ver señalados los bugs en la línea que corresponde.

Como podemos ver ninguno de los bugs es común a dos analizadores a pesar de estar indicando la misma línea de código. **Los dos analizadores se complementan**.

Línea 100 MutableNodeHierarchy.java	**FindBugs**		**PMD**	
Total Bugs	**1**		**4**	
	Categoría	**Tipo**	**RuleSet**	**Rule**
Prioridad 1	MALICIOUS_CODE	MS_SHOULD_BE_FINAL	Naming Rules	VariableNamingConventions
Prioridad 1			Naming Rules	VariableNamingConventions
Prioridad 3			Naming Rules	SuspiciousConstantFileName
Prioridad 3			Naming Rules	LongVariable

PMD muestra además unas estadísticas en forma de tabla de las que no dispone FindBugs.

5.3.6. Valoración de resultados.

En la siguiente tabla valoramos subjetivamente los 5 criterios vistos para determinar la usabilidad:

Usabilidad	FindBugs	PMD
Documentación y ayuda	8	8
Instalación	8	8
Configuración	8	8
Actualizaciones	10	10
Interpretación de resultados	7	9

En conclusión las dos herramientas están bien documentadas, son fáciles de instalar, configurar y actualizar. La interpretación de resultados resulta más sencilla en PMD que en FindBugs por las facilidades que da la herramienta para localizar los bugs desde distintas perspectivas (ficheros XML, desde Eclipse, desde estadísticas de resultados). En los ficheros XML de PMD se incluyen descripciones de los bugs y enlaces externos a la página oficial con más información [17] (en FindBugs no tenemos estas opciones).

5.4. Eficiencia.

Solo FindBugs nos muestra en sus estadísticas XML los tiempos de ejecución del analizador utilizando los atribugos **cpu_seconds** y **clock_seconds**. Para poder ver los tiempos de ejecución de PMD utilizamos el comando time de Linux descontando los tiempos de apertura y cierre de Eclipse.

FindBugs sobre **OpenProj**: cpu_seconds=308.36 total_size=64158

```
<FindBugsSummary timestamp="Tue, 16 Nov 2010 23:45:52 +0100"
total_classes="1368" referenced_classes="2439" total_bugs="0"
total_size="64158" num_packages="112" vm_version="17.0-b16"
cpu_seconds="308.36" clock_seconds="341.97" peak_mbytes="323.47"
alloc_mbytes="247.50" gc_seconds="158.30">
```

FindBugs sobre **argoUML**: cpu_seconds = 676 total_size= 138204

Los tiempos de ejecución de PMD que nos da el comando time son:

- ArgoUML 239 segundos.
- openProj 122 segundos.

FindBugs es bastante más lento que PMD en el mismo proyecto. Además consume más recursos, nos vemos obligados a cambiar los parámetros de memoria dentro de Eclipse.ini para conseguir ejecutarlo sin problemas.

5.5. Extensibilidad de FindBugs y PMD.

Si un analizador es fácilmente extensible podremos añadir nuevas reglas para detectar bugs de distintos tipos. Las reglas pueden escribirse en distintos lenguajes de programación dependiendo de las funcionalidades de la herramienta.

Los dos analizadores permiten la extensión de reglas, si bien PMD tiene un editor propio que permite su gestión.

5.5.1. FindBugs.

En la ventana ***Window -> Preferences*** de Eclipse tenemos las opciones de configuración de FindBugs. En la sección de `Filter Files` podemos añadir un fichero XML que nos permitirá filtrar los bugs que mostrará el analizador.

El proceso para añadir una nueva regla utilizando un fichero .XML. El analizador no da más ayuda para la construcción de nuevas reglas.

5.5.2. Extensibilidad de PMD.

PMD trae un editor **Rule Designer** de reglas que facilita añadir nuevas reglas para las distintas versiones de JDK (JDK 1.3, 1.4, 1.5, 1.6, 1.7, JSP). Para utilizarlo solo tenemos que ir al Menú ***Window -> Preferences*** y pinchar en el botón ***Rule Designer.***El editor permite editar, modificar, copiar y añadir nuevas reglas en XPath o en java.

El proceso de crear una regla se resume en escribir un trozo de código que incumple dicha regla, generar con la herramienta el AST (Árbol de Sintaxis Abstracta) y escribir la expresión XPath que encuentre sólo los nodos del árbol que incumplen la regla.

El editor genera el código XML de la regla para poder incluirla dentro del RuleSet.

5.6. Precisión.

Para determinar la precisión hemos comparado los bugs que encuentran los dos analizadores y los que encuentra cada uno de forma independiente.

	FindBugs	PMD
Ficheros con bugs en argoUML	224 10,19% del total (*)	354 16,11% del total (*)
Ficheros con bugs en openProj	235	501
Ficheros con bugs en un solo analizador de argoUML	188	318
Ficheros con bugs en un solo analizador de openProj	120	386
Ficheros con bugs en los dos analizadores de argoUML	36	36
Ficheros con bugs en los dos analizadores de openProj	115	115
Bugs detectados sin contar líneas repetidas en argoUML	614[(2)]	2946[(4)]
Bugs detectados sin contar líneas repetidas en openProj	920[(3)]	7109[(5)]
Bugs detectados en la misma línea de código sin repeticiones en argoUML	27 (4,39 % de [(2)])	27 (0,91% de [(4)])
Bugs detectados en la misma línea de código sin repeticiones en openProj	288 (31,30%de [(3)])	288 (4,05% de [(5)])

Tabla. Resumen bugs por ficheros/líneas de código sin repeticiones.

(*)Los ficheros .java del proyecto argoUML son 2197 y los de openProj son 559.

PMD localiza el doble de ficheros con bugs que FindBugs, Los resultados de los dos analizadores presentan pocos solapamientos.

6. Resultados.

Comparando los dos proyectos elegidos:

	OpenProj	argoUML
Tamaño en Mb.	64158 Mb.	138204 Mb.
Ficheros .java	559 ficheros	2197 ficheros
Tiempo ejecución FindBugs	308.36 s.	676 s.
Tiempo ejecución PMD	122 s.	239 s.

Tabla resumen comparación ambos proyectos de software libre.

Utilizamos xQuery para poder resumir los resultados de ambas herramientas:

FindBugs			PMD		
	OpenProj	argoUML		OpenProj	argoUML
Bugs Prioridad 1	145	50	Bugs Prioridad 1	406	78
Bugs Prioridad 2	612	336	Bugs Prioridad 2	380	23
			Bugs Prioridad 3	15211	5983
			Bugs Prioridad 4	321	163
			Bugs Prioridad 5	2023	917
Totales	757	386		18341	7164

Tabla resumen prioridades de FindBugs y PMD.

Cada analizador distingue en su salida las siguientes **categorías de bugs**:

Bugs por categoria en FindBugs		
Categ. BAD_PRACTICE	137	89
Categ. CORRECTNESS	49	50
Categ. EXPERIMENTAL	2	11
Categ. MALICIOUS_CODE	150	11
Categ. MT_CORRECTNESS	36	15
Categ. PERFORMANCE	236	89
Categ. STYLE	147	121
Bugs totales (*)	**757**	**386**

(*) En estas estadísticas se contabilizan los duplicados en la misma línea. Para la línea 100 del ejemplo visto en 5.3.5. se contabilizará 4 veces en PMD y 1 vez en FindBugs.

Bugs por categoría en PMD		
RuleSet Basic Rules	258	59
RuleSet Braces Rules	1916	69
RuleSet Clone Implementation Rules	51	--
RuleSet Code Size Rules	572	275
RuleSet Controversial Rules	4349	2705
RuleSet Coupling Rules	44	32
RuleSet Design Rules	1425	480
RuleSet Import Statement Rules	--	5
RuleSet Finalizer Rules	6	--
RuleSet J2EE Rules	59	15
RuleSet JUnit Rules	--	26
RuleSet Jakarta Commons Logging Rules	6	--
RuleSet Java Logging Rules	290	7
RuleSet JavaBean Rules	1126	300
RuleSet Migration Rules	115	40
RuleSet Naming Rules	2222	941
RuleSet Optimization Rules	4586	1881
RuleSet Security Code Guidelines	50	3
RuleSet Strict Exception Rules	19	24
RuleSet String and StringBuffer Rules	113	48
RuleSet Type Resolution Rules	919	185
RuleSet Unused Code Rules	215	69
Totales(*)	**18341**	**7164**

Teniendo en cuenta las características deseables de un analizador vistas en otros puntos de este trabajo podemos resumir:

Característica	FindBugs	PMD
Tiempo de ejecución en OpenProj	308,36s	122s
Tiempo de ejecución de ArgoUML	676s	239s
Usabilidad: (*)	8	7
Ayuda y documentación	8	8
Facilidad para instalación	8	8
Facilidad en configuración del entorno	8	8
Facilidad actualizaciones	10	10
Facilidad interpretación de resultados	6	9
Eficiencia: (*)	7	8
Tiempo de ejecución	6	9
Consumo de memoria sin bloquearse	6	9
Extensibilidad: (*)		
Editor propio de reglas	No disponible	Rule Designer. Disponible editor propio sencillo.
Técnica de análisis	Bug Pattern	Style-checkers Bug-checker
Frameworks disponibles	Eclipse entre otros	Eclipse entre otros
Características bugs detectados:		
Categorías de bugs detectados para proyectos	2 prioridades	5 prioridades
Bugs detectados en código java	7 tipos	22 tipos

(*) Criterio subjetivo valarado de 1 a 10, el 1 indicaría poco adecuado (poca usabilidad, eficiencia, extensibilidad) el 10 indicaría eficaz (buena usabilidad, eficiencia, extensibilidad).

6.1. Conclusiones.

Para elegir una herramienta de análisis estático es necesario tener en cuenta características como la usabilidad, la eficiencia, la extensibilidad y la técnica de análisis. En muchos casos será necesario elegir más de un analizador de distinto tipo para poder abarcar más tipos de bugs. De los resultados extraídos en estas pruebas podemos decir que:

- Los dos analizadores son fáciles de usar en cuanto a instalación, configuración, actualización y uso posterior. Hay plugins para varias plataformas e IDEs. Desde Eclipse ambos añaden una perspectiva más que permite la ejecución desde un menú de contexto.
- PMD resulta más rápido que Findbugs. El consumo de recursos es mayor en FindBugs que en PMD. FindBugs añade en los ficheros XML atributos de tiempo de ejecución para cada fichero y proyecto. PMD no dispone de este tipo de información.
- En cuanto a extensibilidad PMD resulta más sencillo a la hora de añadir nuevas reglas. Trae un editor propio que permite copiar, modificar, añadir y eliminar reglas de una forma intuitiva y sencilla. No pasa lo mismo con FindBugs que dificulta esta tarea por tener que programar desde cero cada regla que se desee añadir a un proyecto.
- La documentación de bugs que permite interpretar los resultados aparece más documentada en PMD que en FindBugs. PMD incorpora además una funcionalidad para generar estadísticas resumen.
- En los resultados de los dos analizadores, PMD localiza muchos más bugs que FindBugs en los dos proyectos vistos. Los dos muestran una descripción y localización correcta del error,

aunque referida en algunos casos al número de línea donde comienza la clase y no al número de línea del bug. Solo en los ficheros XML de PMD se incluyen descripciones de los bugs detectados.

- Los dos analizadores se complementan en cuanto a los tipos de bugs que localizan. La técnica de análisis es distinta en las dos herramientas, FindBugs está clasificado como Bug Pattern y PMD como Style-checker y Bug-Checker, esta es una de las razones por las que el porcentaje de bugs comunes no supera el 30% en los dos proyectos vistos.
- Con los criterios vistos siempre resultará más eficaz utilizar más de una herramienta para el análisis de código estático. De esta forma aumentamos el número de bugs detectados.

7. Trabajos futuros.

Algunos de los trabajos futuros que pueden ser de interés para la comunidad para mejorar estas herramientas serían:

- Crear un editor de reglas dentro de FindBugs. PMD trae un editor de reglas que permite gestionar las reglas desde Eclipse sin necesidad de utilizar ningún otro editor.
- Mejora de las estadísticas que presenta FindBugs. Se hace mucho más útil que las salidas de bugs se presenten en tablas.
- Añadir tiempo de ejecución en los ficheros XML de PMD. Los tiempos de ejecución y consumo de recursos ayudan a resolver problemas de eficiencia cuando los proyectos son muy grandes.
- Crear un DTD para los informes de salida de todos los analizadores. Los informes de salida XML deberían de cumplir con un DTD que hiciese más fácil combinar los resultados de varios analizadores en uno solo.

8. Bibliografía.

1. Libro. Compiladores: Principios, técnicas y herramientas.
Autor. de Aho, Sethi y Ullman.
Editorial.Addison-Wesley Logman de México. Año. 2008.
Lugar de publicación. México.
2. Tesis. Improving the Usability of the ESC/Java Static Analysis Tool.
Autor. Michael Peck. Año. 2004, abril.
Lugar de publicación. School of Engineering and Applied Science. University of Virginia.
3. Artículo. Evaluation Static Analysis Defect Warnings On Production Software.
Autor. Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, YuQian Zhou.
Año. 2007, junio.
Lugar de publicación. http://findbugs.cs.umd.edu/papers/FindBugsExperiences07.pdf.
4. Artículo. Improving your software using static analysis to Find Bugs.
Autor. Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, Kristin Stephens.
Año publicación. 2006, octubre.
Lugar de publicación. Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, Universidad de Mariland, Oregon. USA. Págs: 673 - 674
5. Artículo. Evaluating Static Analysis Frameworks.
Autor. Ciera Nicole Christopher, Carnegie Mellon University. Año. 2006, mayo.
Lugar publicación. http://www.cs.cmu.edu/~aldrich/courses/754/tools/christopher-analysis-frameworks-06.pdf. Carnegie Mellon University.

6. Artículo. Prevent Vs FindBugs Application and Evaluation.
Autor. Lyle Holsinger, Snehal Fulzele, Smita Ramteke, Ken Tamagawa, Sahawut Wesaratchakit.
Año. 2008, marzo.
Lugar publicación. Universidad de Pensilvania, trabajo de estudiantes. El profesor Jonathan Aldrich lo tiene publicado en su web: http://www.cs.cmu.edu/~aldrich/
7. Artículo. Finding bugs is easy.
Autor. David Hovemeyer, William Pugh. Año. 2004.
Lugar de publicación. ACM SIGPLAN Notices. Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. Pag: 132 - 136.
8. Artículo. An Approach to Merge Results of Multiple Static Analysis Tools.
Autor. Na Meng, Qianxiang Wang, Qian Wu, Hong Mei. Año. 2008.
Lugar de publicación. Págs. 169-174, The Eighth International Conference on Quality Software, 2008. QSIC 2008.
9. Artículo. A comparison of bug finding tools for java.
Autor. Nick Rutar, Christian B. Almazan, Jeffrey S.Foster. Año. 2004. ISSREE-2004.
Lugar de publicación. Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04).
10. Artículo. Introduction to Extended Static Source Code Checking with ESC/Java2.
Autor. Martin Seelhofer. Año. 2009, junio.
Lugar de publicación:
http://wiki.ita.hsr.ch/SemProgAnTr/files/MSE_Seminar_ProgTrans_F09_-_Martin_Seelhofer_-_Final_20090528.pdf es parte del seminario: Program Analysis and Transformation.
11. Artículo. Trabajo alumnos CMU. Mini-Project: Tool or Analysis Practicum.
Autor. Bokuk Seo, Do-Hoon Kim, Yong Gyu Kim, Sang-hyun Lee.
Lugar de publicación. Página del profesor Jonathan Aldrix de la universidad CMU (Carniege Mellon University). Año. 2009, abril.
12. Artículo. Extended Checker for FindBugs.
Autor. Jun Zhang. Año. 2006.
Lugar de publicación. Publicado en la web del autor http://cs.ubc.ca/~ericazhj/papers/pl.pdf y en la web de la Universidad de Colombia (ubc.ca).
13. Herramienta. CheckStyle.
Año. 2010.
Web. http://checkstyle.sourceforge.net/
14. Herramienta. Coverity Prevent.
Año.2010.
Web.http://coverity.com/products/static-analysis.html
15. Herramienta. FindBugs.
Año. 2010.
Web. http://findbugs.sourceforge.net/.
Descripción. Sitio descarga de la herramienta FindBugs.
16. Herramienta. Jlint.
Web. http://www.garret.ru/jlint/ReadMe.htm, http://jlint.sourceforge.net/
Autor. Konstantin Knizhnik, Cyrille Artho.
Año.2003
17. Herramienta. PMD
Web. http://pmd.sourceforge.net/
Año. Fecha última modificación 2009.
18. Herramienta. Sonar.
Año. 2010, marzo (versión 2.1).
Web. http://www.sonarsource.org/

19. OpenProj.
Año. octubre 2008 versión 1.4.
Web. http://www.openproj.org/product-overview
20. argoUML.
Año. 2009 versión 0.29.3.
Web. http://argouml.tigris.org/
21. Qizx/Open.
Web. http://www.xmlmind.com/qizxopen/download.shtml

Revistas técnicas de posible publicación.

- **Revista Novática.**

  Novática, creada en 1975 y de periodicidad bimestral, es la decana de las revistas informáticas españolas y está incluida en numerosos catálogos e índices, nacionales e internacionales, de publicaciones técnicas y científicas.

  En 2006 creó el Premio Novática, destinado al mejor artículo publicado cada año por Novatica.

- **Revista NEX.**

  El contenido de la Revista NEX está dirigido a empresas dedicadas a las tecnologías de la información y/o aquellas que apoyan sus negocios en la tecnología. También está orientado a personas con un particular interés en seguridad y administración de redes o programación, administradores de sistemas, encargados de seguridad informática, empresas de diseño y hosting de páginas web. Para aquellos estudiantes universitarios de las carreras de sistemas y para quienes estén buscando las Certificaciones Internacionales de IT les será también de gran utilidad.