¿Qué es el **Análisis Estático del Código**?

por Raúl Expósito

Imagen: http://www.flickr.com/photos/scott_waterman/178367110/

Índice

¿Qué contiene este documento? 03
Sobre el autor 03
Sobre la licencia 03
Teoría acerca del Análisis Estático del Código 04
¿Qué tipos de análisis estático del código existen? 06
¿Cuándo debemos hacer estos análisis? 06
Actividades complementarias al análisis estático del código 09
Limitaciones del análisis estático del código 09
Conclusiones 10
Análisis Estático del Código en Java 12
PMD 13
CPD 14
Checkstyle 15
Findbugs 16
Otras herramientas 16
Análisis Estático del Código en Groovy 18
Codenarc 18

¿Qué contiene este documento?

El texto que vas a poder leer a continuación explica en qué consiste la técnica del **análisis estático del código**, la cual tiene como objetivo mejorar la calidad del código fuente del software.

Te encuentras ante un documento de carácter técnico que no entra en demasiado detalle para conseguir que cualquier persona pueda leerlo, aunque inevitablemente aparecerán algunos términos y vocablos propios de la jerga informática que se explicarán de una manera sencilla según aparezcan.

Como complemento a la teoría acerca de qué es y en qué consiste el análisis estático del código se presentarán varias herramientas para dos lenguajes de programación en particular: java y groovy. De este modo lo que se pretende es que se entienda mejor cuál es el alcance de esta técnica.

La maquetación de este documento está inspirada en la fantástica maquetación del documento *"Eres Productivo. Vol 1"* de Berto Pena, mientras que los iconos son propiedad de Mark James:

http://albertopena.com/descargas/

http://www.famfamfam.com/

Sobre el autor

Raúl Expósito es un ingeniero en informática residente en Getafe (Madrid) a quien le gusta materializar ideas, la creatividad, el diseño, el cuidado de los detalles y la simplicidad, definiéndose por tanto como una persona con vocación técnica pero a la vez creativa y con inquietudes.

Podéis encontrarle en su página web:

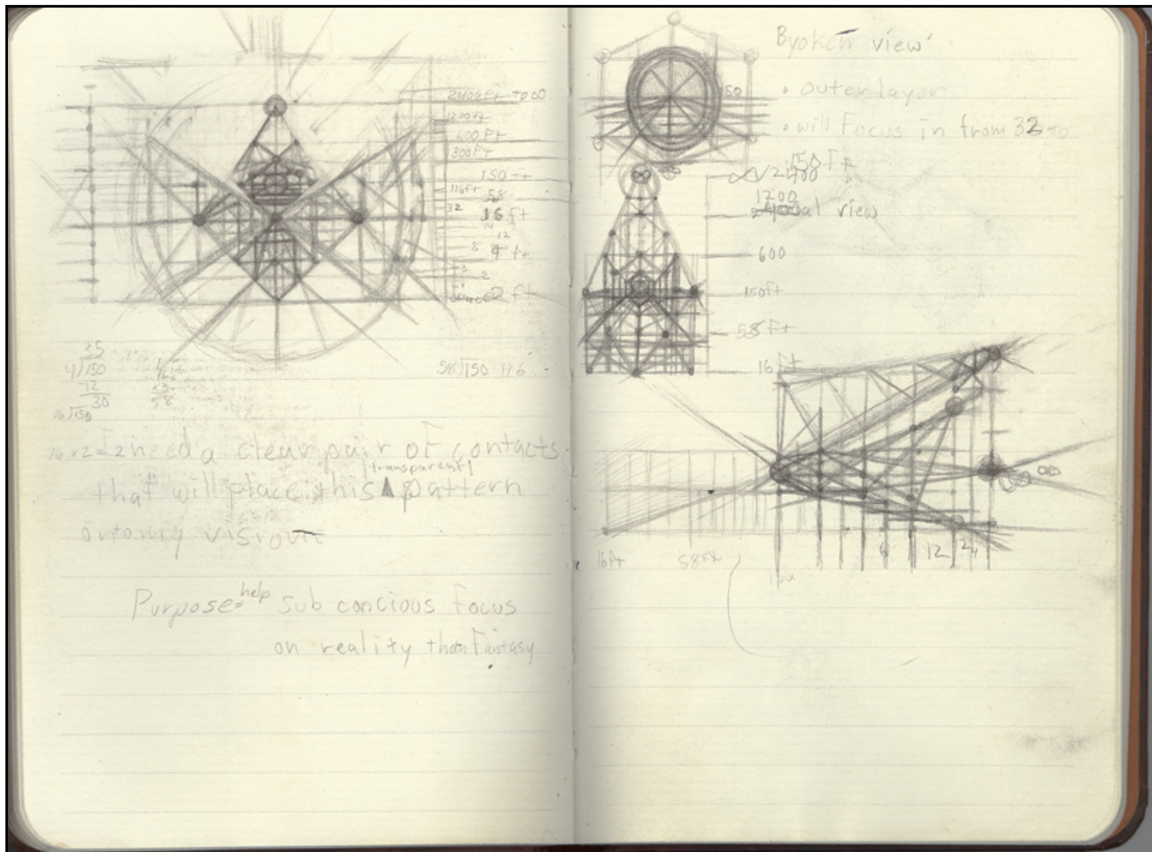
http://raulexposito.com/

Sobre la licencia

Este documento se publica bajo la licencia Creative Commons 3.0 de Reconocimiento-No comercial-Sin obras derivadas, lo que significa que puedes realizar y distribuir copias siempre que:

- reconozcas la autoría de estos textos,
- no haya un beneficio comercial en tu distribución de las copias y
- no modifiques el contenido.

Teoría acerca del Análisis Estático del Código



Imaginemos que somos miembros de un equipo de desarrollo. Nos encontramos creando cierto software y en un momento determinado nos planteamos analizar estáticamente el código. ¿A qué nos referimos con esto?, ¿qué vamos a hacer exactamente?, ¿en qué consiste esta tarea?. Es posible que la definición más breve y concisa de la técnica que vamos a utilizar sea la siguiente:

> *"El análisis estático del código es el proceso de evaluar el software sin ejecutarlo"*

Es, por tanto, una técnica que se aplica **directamente** sobre el código fuente tal cual, sin transformaciones previas ni cambios de ningún tipo. La idea es que, en base a ese código fuente, podamos obtener información que nos permita mejorar la base de código **manteniendo la semántica original**.

El analizador estático de código, para ello, recibirá el código fuente de nuestro programa, lo procesará intentando averiguar qué es lo que queremos que haga y **nos dará sugerencias** con las que poder mejorar ese código.

Pero, ¿cómo hace esto?, ¿qué hace para "saber" qué es lo que queremos hacer y qué podemos hacer para mejorarlo?. Estas herramientas incluyen, por un lado, analizadores léxicos y sintácticos que procesan el código fuente y, por otro, un conjunto de reglas que aplicar sobre determinadas estructuras. Si nuestro código fuente posee una estructura concreta que el analizador considere como "*mejorable*" en base a sus reglas nos lo indicará y nos sugerirá una mejora.

Y todo esto, ¿para qué sirve?, ¿qué salimos ganando realizando estos análisis?. Básicamente **ganamos en facilidad de mantenimiento y de desarrollo** ya que su objetivo es **minimizar la deuda técnica** de nuestros proyectos, y es que algunas de las funciones de los analizadores consisten en encontrar partes del código que puedan:

- reducir el rendimiento,
- provocar errores en el software,
- complicar el flujo de datos,
- tener una excesiva complejidad,
- suponer un problema en la seguridad.

Como podéis ver su misión es la de servirnos de ayuda en nuestros desarrollos, ayudándonos a detectar nuestros propios errores y

RAULEXPOSITO.com

ofreciéndonos posibles soluciones a los mismos. Nos ofrecen mucho y no nos piden nada a cambio, así que, ¿por qué no utilizarlos?

¿Qué tipos de análisis estático del código existen?

Podríamos decir que existen dos. Por un lado está el análisis **automático** que realiza un programa de ordenador sobre nuestro código y por otro está el análisis **manual** que realiza una persona. Cada uno de estos análisis persigue unos objetivos concretos:

- El análisis realizado por un programa de ordenador, o análisis automático, reduce la complejidad que supone detectar problemas en la base del código ya que los busca utilizando a unas reglas que tiene predefinidas.

- El análisis realizado por una persona, o análisis manual, se centra en apartados propios de **nuestra** aplicación en concreto como, por ejemplo, determinar si las librerías que está utilizando nuestro programa se están utilizando debidamente o si la arquitectura de nuestro software es la correcta.

Ambos, por tanto, **son complementarios**. El análisis automático se centra únicamente en facetas de más bajo nivel como la sintaxis y la semántica del código, funcionando este análisis en **cualquier tipo de aplicación**, mientras que el análisis manual se ocupa de facetas de más alto nivel como, por ejemplo, la estructura de nuestra aplicación o su manera de trabajar con otros elementos externos como las librerías.

La unión de ambos tipos de análisis nos permitirá identificar los potenciales problemas a distintos niveles que generen la deuda técnica de nuestro proyecto. Debemos, por tanto, unificar ambas para poder mejorar la base de nuestro código fuente, lo cual repercutirá en una mejora tanto del desarrollo como del mantenimiento de nuestro software antes incluso de llegar a ejecutarlo.

¿Cuándo debemos hacer estos análisis?

Ahora que conocemos las ventajas del análisis estático del código, que sabemos lo útil que es, que sabemos que existe el análisis manual y el automático y que sabemos que nos va a ayudar a hacer nuestros desarrollos cabe preguntarnos ¿cada cuánto debo realizar este análisis?,

¿todos los días antes de acostarme?, ¿cada vez que escriba una nueva línea de código en mi aplicación?, ¿o una vez al año que no hace daño?

Debemos tener en mente que el análisis estático del código es **un medio** con el que conseguir mejorar nuestro código fuente, **no un fin** en sí mismo. Debemos, por tanto, usarlo únicamente como apoyo.

Una diferencia entre el análisis automático y el análisis manual es el tiempo que éste tarda en realizarse. De este modo un análisis automático se realiza en pocos minutos mientras que uno manual puede tardar varias horas. Esto, como es natural, influye a la hora de establecer una periodicidad.

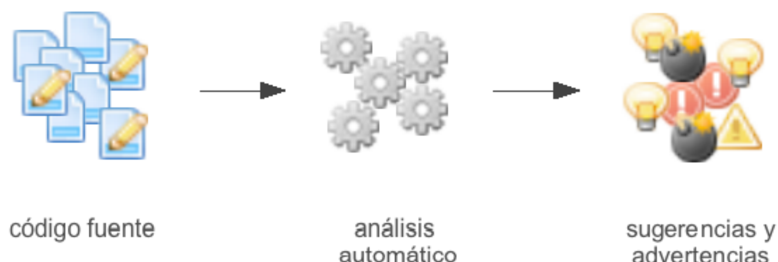
Deberíamos tratar de realizar el análisis manual cada vez que vayamos a crear una nueva funcionalidad en nuestro software, preguntándonos en este caso si la arquitectura actual nos permite implementarla con facilidad, si disponemos de las librerías adecuadas o si podemos modificar la base de nuestro código para facilitar el desarrollo de esta nueva funcionalidad. Es decir, lo reservaremos para situaciones concretas.

Como es natural también podremos hacer este tipo de análisis cuando el proyecto sencillamente va mal, cuando nos cueste mucho esfuerzo desarrollar nuevas funcionalidades o modificar las que ya tenemos o, en definitiva, cuando sintamos que las piezas del software que estamos desarrollando *chirrían*, aunque este último caso no es el deseable y es precisamente lo que queremos evitar con el análisis estático del código.

El análisis automático, en cambio, puede ser realizado con una mayor periodicidad ya que no requiere de intervención humana y, lo que es mejor, puede ser programado y repetido tantas veces como sea necesario. Además obra con objetividad, siempre nos va a devolver la misma respuesta ante el mismo código fuente de entrada.

Sin embargo no tenemos que creernos literalmente lo que nos diga. Puede que entre sus reglas esté determinada una condición de error concreta que, en nuestro código en particular, no debería ser etiquetada como tal.

El caso ideal sería integrar nuestro analizador automático con alguna opción de integración continua o alguna tecnología que permita generar la aplicación a partir de nuestro código fuente. De este modo cada vez que hagamos algún cambio o que queramos generar nuestra aplicación se ejecutará el analizador automático y este, a su vez, nos hará sugerencias.



Los desarrolladores, por lo general, cuando implementamos por completo alguna funcionalidad en nuestros ordenadores dejamos nuestro código fuente en un repositorio centralizado donde el resto de desarrolladores puedan encontrarlo. Un buen momento para analizar automáticamente el código es el instante en el que ese código pasa de estar únicamente en nuestros ordenadores a formar parte del repositorio, ya que así todo el equipo de desarrollo puede ser consciente de las posibles mejoras que se puedan realizar sobre ese código.

Sin embargo, es posible que queramos que el código ya se encuentre mejorado antes incluso de formar parte del repositorio. Los desarrolladores disponemos, por lo general, de herramientas que transforman nuestro código fuente en aplicaciones. Si utilizamos nuestro analizador automático en nuestro propio ordenador cuando convertimos nuestro código fuente en aplicaciones sólo nosotros recibiremos las sugerencias y podremos seguirlas antes de poner el código a disposición de todo el equipo de desarrollo.

En cualquier caso estas no son reglas de oro ya que el momento en el que se deba realizar el análisis estático del código dependerá del proyecto concreto y de sus circunstancias.

Actividades complementarias al análisis estático del código

Sin embargo, las posibilidades de mejorar nuestra base del código no quedan aquí. Hay otras técnicas que podemos utilizar para conseguir mejorar el código fuente de nuestra aplicación y, con ello, el software que utilizan los usuarios como producto final.

Estas técnicas, al contrario que la que nos ocupa, se centran en analizar el código mientras este se encuentre en ejecución. Explicarlas excede el alcance del presente documento pero, sin embargo, se van a resumir brevemente para poder situar aún mejor el análisis estático del código dentro de su ámbito:

- **Tests**. Son una serie de procesos que permiten verificar y comprobar que el software cumple con los objetivos y con las exigencias para las que fue creado. Su misión es la de encontrar errores antes de que el software final sea utilizado por los usuarios y los hay de varios tipos: unitarios, funcionales, de integración, de carga, de regresión, etc., cada uno centrado en una faceta del software en concreto.

- **Profiling**. Se encarga de analizar el rendimiento del software mientras este se encuentra en ejecución determinando qué recursos son utilizados en cada momento por las distintas partes del software. Con ello lo que se persigue es identificar qué partes del código implican una mayor carga para el sistema para poder obrar en consecuencia aplicando cambios.

Estas técnicas permiten encontrar errores en el software que no es posible detectar cuando se analiza estáticamente el código aunque, sin embargo, es posible que el análisis estático del código nos permita corregir algunos errores antes incluso de detectarlos durante la realización de tests o de profiling.

Limitaciones del análisis estático del código

Sin embargo, a pesar de sus ventajas, los analizadores estáticos del código tienen sus limitaciones:

- No nos permiten saber si el software va a hacer **lo que se espera de él o no**. Podemos analizar el código fuente y saber cómo mejorarlo

muchísimo, pero no sabremos si hace lo que tiene que hacer u otra cosa totalmente distinta e inesperada.

- Al utilizar analizadores automáticos estos nos pueden devolver **falsos positivos**. Es posible que el analizador detecte como error algo que nosotros sabemos que está bien y que por alguna buena razón está programado así aposta.

- Están **limitados** al análisis de código sin llegar a ejecutarlo, de tal modo que necesitamos complementarlo con tests o con profiling para poder realizar mejoras en un ámbito mayor.

- Los analizadores estáticos del código están **muy ligados** a lenguajes de programación **concretos**, e incluso entre lenguajes hay diferencias, ya que en lenguajes estáticos como java es más sencillo hacer un análisis estático del código que en otros lenguajes dinámicos como groovy.

Conclusiones

A lo largo de este documento hemos introducido la técnica del análisis estático del código, explicando para ello sus objetivos y sus metas y dividiéndolo entre análisis manual y análisis automático, lo cual nos ha permitido determinar las diferencias de uno con respecto al otro en distintos ámbitos.

También lo hemos englobado con otras técnicas que permiten mejorar la base del código como son los tests y el profiling, los cuales tienen su origen en la ejecución de dicho código.

¿Es realmente importante hacer este tipo de análisis?, ¿deberíamos realizarlo o podríamos seguir como estamos?. Esta técnica es muy sencilla de utilizar, si la automatizamos incluso podemos limitarnos a esperar sus resultados, y como podéis ver nos ayuda sin pedirnos nada a cambio.

En mi opinión es un tipo de análisis que deberíamos hacer. No cuesta trabajo y sólo puede darnos beneficios, tanto a corto plazo mirando en el desarrollo más inmediato como a largo plazo en vistas a un futuro mantenimiento.

Pero, ¿deberíamos quedarnos únicamente ahí?, ¿o podemos hacer algo más para mejorar nuestro código y, con ello, nuestros desarrollos?. Si queremos mejorar la calidad de nuestros productos hacer este tipo de análisis es un buen primer paso, pero quizá deberíamos dar algún paso más completándolo con tests e incluso con profiling.

En cualquier caso, ¿qué podemos perder con probarlo?. El coste es cero y los beneficios palpables, ¿existe una relación calidad/precio mejor?

Imagen: http://www.flickr.com/photos/ninjaguy82/756138395/

Análisis Estático del Código en Java

Una vez hemos terminado de ver qué es el análisis estático del código vamos a centrarnos en aplicarlo sobre código escrito en un lenguaje concreto: **Java**.

Gracias a la amplia difusión de este lenguaje hay infinidad de herramientas, librerías y tecnologías que giran en torno suya. El análisis estático del código no es una excepción y, como veremos, hay varios analizadores automáticos que nos permitirán mejorar el código fuente de nuestras aplicaciones escritas en java.

Finalmente presentaremos un producto que, como veremos, utiliza los resultados de los analizadores automáticos que vamos a mostrar a continuación y los unifica en una única salida que pretende ser mejor que la suma de las salidas de los analizadores automáticos de manera individual.

Pero antes de ello conozcamos los distintos analizadores disponibles:

PMD

http://pmd.sourceforge.net/

Es una herramienta muy conocida, tanto por su facilidad de uso como por su fácil integración con las tecnologías que estemos utilizando para desarrollar nuestro proyecto.

Su funcionamiento se basa en **detectar patrones**, los cuales son posibles errores que pueden aparecer en tiempo de ejecución, código que no se puede ejecutar nunca porque no hay manera de llegar a él, código que puede ser optimizado, expresiones lógicas que puedan ser simplificadas, malos usos del lenguaje, etc.

Estos patrones se encuentran catalogados en distintas categorías, entre las cuales estarían las básicas, las de tamaño, acoplamiento, tratamiento de excepciones, nombrado, complejidad, optimización, seguridad, etc.

http://pmd.sourceforge.net/rules/index.html

Además va más allá de lo que únicamente es la detección de errores en nuestro código ya que también encuentra errores en el uso que hagamos de tecnologías como Android, JSF, JSP o JUnit.

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. Además también podemos configurar el formato en el que deseamos que nos devuelva los resultados para que elijamos entre aquello que nos resulte más cómodo: xml, html, txt, etc.

CPD:

http://pmd.sourceforge.net/cpd.html

Forma parte de PMD, aunque funciona de una manera autónoma, y es sin duda uno de los analizadores **más útiles** que podamos encontrar.

CPD son las siglas de "**C**opy **P**aste **D**etector" y con ese nombre queda clara cuál es su misión: **buscar duplicidades en el código**. Todo aquello que esté escrito más de una vez en nuestro código será detectado por CPD.

Pero, ¿qué sentido tiene encontrar código que haya sido copiado y pegado?. Cuando hay código duplicado, triplicado o n-plicado en nuestra aplicación nos encontramos con un **grave** problema: si hay que hacer cambios en código duplicado de la aplicación tendremos que recorrer todo el código de la misma para repercutir ese mismo cambio en todas las copias que hayamos hecho.

Imaginemos un caso. Si en varios puntos de mi aplicación necesito poder comprobar si un año es bisiesto, ¿tendría sentido que el mismo código de la misma comprobación apareciera copiado en varios lugares de la aplicación?, ¿o debería crear una zona específica donde realizarla?

Si al crear el código de la comprobación original he cometido un error y lo he copiado y pegado por todo el código de mi aplicación me encontraré con el problema de que, una vez detectado el error y subsanado en el código original, deberé buscar el código erróneo entre todos los entresijos de su código fuente para poder reemplazarlo por el que resuelve el problema. Pero, ¿quién me asegura que he encontrado todas las copias?, ¿el problema se habrá solucionado en todos los casos o funcionará en unos y fallará en otros?

En cambio, si tengo ese código en un único sitio sólo me tengo que preocupar de solucionar el problema en un único punto y sé que, una vez solucionado, está solucionado en todos los casos.

Otro caso ficticio vendría de la mano de no utilizar el mecanismo de la herencia de Java. Si mi aplicación necesita un tipo 'Jefe' con nombre, DNI y departamento y un tipo 'Empleado' con nombre, DNI y horario, ¿necesito crear los dos tipos con nombre y DNI separados?. Si son lo mismo, ¿por qué debería tenerlo dos veces?, ¿o es que no debería?

Por suerte puedo crear el tipo 'Persona', con nombre y DNI, y crear los tipos 'Jefe' y 'Empleado' que hereden de 'Persona', de tal modo que estos tipos ya tengan nombre y DNI por heredar de 'Persona'. Así sólo tengo que tenerlo en un único sitio, en 'Persona'.

Los problemas derivados de la función "copiar y pegar" tienen su solución en su función hermana: "**cortar y pegar**":

- Si inicialmente sólo necesitábamos comprobar si un año es bisiesto en un punto y, posteriormente, necesitamos hacer esta comprobación en más sitios lo que tenemos que hacer es **mover** la comprobación a un lugar más adecuado. Para ello **cortamos y pegamos** el código en un sitio común a todos los puntos y, desde ellos, invocamos la validación.

- Si inicialmente nuestra aplicación sólo tenia jefes y posteriormente se añadieron los empleados, lo que tenemos que hacer es crear un nuevo tipo y allí **cortamos y pegamos** lo que ambos tengan en común.

Al igual que PMD, CPD permite devolver los resultados de sus análisis en varios formatos, como por ejemplo xml o txt.

Checkstyle

http://checkstyle.sourceforge.net/

Inicialmente se desarrolló con el objetivo de crear una herramienta que permitiese comprobar que el código de nuestras aplicaciones se ajustase a los estándares dictados por Sun Microsystems, empresa creadora del lenguaje Java.

Sin embargo, posteriormente se añadieron nuevas capacidades que han hecho que sea un producto muy similar a PMD. Es por ello que también busca patrones en el código que se ajustan a categorías muy similares a las de este analizador.

Una ventaja que tiene con respecto a PMD es que además **encuentra errores** en la documentación que podamos haber escrito **en el javadoc**. PMD no realiza ningún tipo de validación mientras que Checkstyle es muy riguroso en ese sentido.

Gracias a ello, si tenemos una función recibe dos parámetros y documentamos uno o ninguno, Checkstyle nos avisará de que estamos dejando parte de la función sin documentar.

Findbugs

http://findbugs.sourceforge.net/

Es un producto de la Universidad de Maryland que, como su nombre indica, está especializado en **encontrar errores**.

Al igual que los demás tiene una serie de categorías donde poder catalogar dichos errores, y es que según findbugs estos pueden ser malas prácticas, mal uso del lenguaje, internacionalización, posibles vulnerabilidades, mal uso de multihilo, rendimiento, seguridad o, directamente, **chapuzas**.

http://findbugs.sourceforge.net/bugDescriptions.html

Otras herramientas

En esta lista faltaría por mencionar Crap4J, que es un producto de Agitar Software cuya función es la de tratar de descubrir qué parte de nuestro código es muy difícil de modificar:

http://www.crap4j.org/

http://adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=crap4j

Para ello utiliza, por un lado, el número de bifurcaciones de los fragmentos de código y, por el otro, hasta qué punto éstos están cubiertos por tests unitarios. Lo he dejado apartado ya que los tests deben ejecutar el código y se salen, por tanto, de las técnicas de análisis estático del código.

Una herramienta muy interesante, la cual he introducido al comenzar este apartado dedicado a Java, es Sonar:

http://sonar.codehaus.org/

Su misión es la que aparece en la página web:

“Pon tu deuda técnica bajo control”

En su demo podemos verlo en acción analizando muchos proyectos open source:

http://nemo.sonarsource.org/

Si vemos los resultados para Checkstyle, veremos lo siguiente:

http://nemo.sonarsource.org/project/index/136602

Sonar internamente utiliza las herramientas comentadas anteriormente: PMD, CPD, Findbugs, checkstyle, etc. y unifica sus resultados mostrando, además, más información: complejidad, número de métodos, de clases, de líneas de código, mantenibilidad, etc. Todo de una manera muy visual y muy intuitiva.

Si estás desarrollando proyectos con Java estás ante una herramienta **absolutamente imprescindible**.

Imagen: http://www.flickr.com/photos/dodogoeslr/2858347645/

Análisis Estático del Código en Groovy

Groovy es un lenguaje de programación bastante reciente, dinámico, muy expresivo, productivo y con características similares a otros lenguajes como Ruby o Python que funciona bajo la máquina virtual de Java.

http://groovy.org/

http://www.escueladegroovy.com/

A pesar de su ser un lenguaje nuevo, o posiblemente debido a ello, no hay muchos analizadores estáticos del código para este lenguaje. Al contrario de lo que ocurría con Java, Groovy es un lenguaje dinámico, lo cual dificulta en parte la tarea de analizar el código.

Sin embargo contamos con un analizador: codenarc.

Codenarc

http://codenarc.sourceforge.net/

http://raulexposito.com/blog/2009/11/analiza-tu-codigo-fuente-groovy-con-codenarc/

RAUL EXPOSITO .com

Según cuentan en su página web, codenarc es un analizador inspirado en PMD y en Checkstyle entre otros.

Es por ello que sus reglas son similares a las de los analizadores anteriores, ya que posee reglas dedicadas a encontrar potenciales problemas en cosas básicas del uso del lenguaje, en excepciones, en carga de otras clases, en código sin utilizar, etc.

Su uso es muy sencillo, ya que se puede integrar fácilmente con tecnologías que usan groovy como base como son Grails y Griffon, aunque también puede ser ejecutado directamente desde la consola.

Es un analizador muy útil para un lenguaje muy interesante que está en pleno crecimiento. Espero que crezca y que aparezcan otros analizadores alternativos con los que poder mejorar nuestros desarrollos escritos en groovy.

Imagen: http://www.flickr.com/photos/tussenpozen/31414294/