Universidad de Valladolid

E.T.S Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Mención Ingeniería de Software

Estudio y ampliación de SonarQube

Autor:

Héctor Arranz Orejudo

Tutora:

Yania Crespo González-Carvajal

Resumen

Con este proyecto se pretende, por un lado, estudiar el funcionamiento de SonarQube a nivel usuario, aprendiendo a analizar proyectos y descubrir qué significan las métricas que este software nos ofrece. Se pretende ver cómo se puede integrar con otras herramientas mediante plugins para extender la funcionalidad que ofrece la herramienta.

Por otro lado, una vez tengamos claro cómo se utiliza SonarQube, el siguiente objetivo es estudiar los mecanismos de generación de nuevas reglas para analizar lenguaje Java, y crear una nueva con cada uno de los mecanismos de extensión disponibles.

Tabla de contenido

Índice de Ilustraciones

Índice de Tablas

1. INTRODUCCIÓN

Introducción

Una de las fases que posee la mayoría de proyectos software es la de mantenimiento del producto. A lo largo de la vida útil del software hay que corregir nuevos *bugs* encontrados, desarrollar nuevas funcionalidades necesarias, modificar las existentes para adaptarse a las necesidades de los usuarios o mejorar el rendimiento. Esta fase supone un porcentaje muy alto del presupuesto del proyecto, superando en muchas ocasiones al presupuesto de desarrollo.

SonarQube es un software que nos ayuda a ir generando un código lo más sencillo y legible posible, y que apenas contenga los fallos más habituales, eliminando errores potenciales y evitando así futuros *bugs*. De esta manera haremos que la fase de mantenimiento requiera menos horas y, por consiguiente, menos dinero.

Motivación

Este proyecto fue propuesto por mi tutora Yania Crespo, en las propuestas de TFG que salen a principio de curso para los estudiantes que estén cursando la asignatura y no tengan alguna idea propia.

Me interesó bastante debido a que está centrado en una herramienta que ayuda a mantener una buena calidad del código, y es un campo en el que me gustaría desarrollarme junto con el diseño de software. De esta manera todo lo aprendido aquí podré usarlo en mi día a día y ayudarme a ser más profesional.

Objetivos

Este proyecto tiene dos objetivos diferenciados:

- Investigar y aprender cómo funciona la herramienta SonarQube, para qué se utiliza y en qué casos, y descubrir cómo se calculan algunas de las métricas más relevantes que ofrece los informes de SonarQube.
- Extender la herramienta creando nuevas reglas mediante los dos mecanismos de extensibilidad disponibles para generar nuevas reglas utilizadas en el análisis del lenguaje Java: XPath y Java.

Organización de la memoria

La memoria se organiza en varios capítulos que van en orden cronológico a como se ha investigado.

- **Capítulo 2**, donde se presenta la planificación y seguimiento del proyecto.
- **Capítulo 3**, breve explicación de las características de SonarQube.
- **Capítulo 4**, guía de instalación de todo los necesario.
- **Capítulo 5**, se explica cómo analizar proyectos de varias maneras.

- **Capitulo 6**, se empieza a mostrar cómo se usa SonarQube, concretamente la parte de instalación de plugins, las reglas, y los perfiles que se usan al analizar un proyecto.
- **Capítulos 7 y 8**, dos ejemplos de análisis de proyectos OpenSource, en los que se va explicando apartado por apartado las métricas que nos ofrece SonarQube.
- **Capítulo 9**, se explica y analiza de forma práctica la creación de nuevas reglas usando XPath.
- **Capitulo 10**, se explica y analiza de forma práctica la creación de nuevas reglas mediante Java.
- **Capitulo 11**, con las conclusiones que he podido sacar de este proyecto.
- **Capítulo 12**, referencias bibliográficas consultadas.
- **Capitulo 13**, listado con el contenido del CD.

2. PLANIFICACIÓN

Después de un análisis, se ha dividido el proyecto en varias tareas y subtareas para facilitar su realización y seguimiento. A cada tarea se le ha asignado un porcentaje del tiempo total del proyecto, que es lo que se estima que durará dicha tarea.

Debido a que solo hay una persona realizando el proyecto, todas las tareas son secuenciales. Hasta que no se acabe una tarea, no se podrá empezar la siguiente. Solo existe una excepción, la tarea de 'Documentación', que es una tarea paralela que se realizará durante todo el proyecto, simultáneamente con el resto de tareas. El objetivo es ir documentando todos los resultados a medida que se van obteniendo.

Una vez definidas todas las tareas, y ordenadas cronológicamente, se calcula el tiempo en horas que habrá que dedicar a cada una, partiendo del porcentaje definido en el análisis, y las horas totales del proyecto, que en este caso serán 300 horas. Introducimos toda esta información en MS Project Professional 2013, el cual, aportándole una fecha de inicio, un horario y un calendario laboral, calcula automáticamente las fechas de inicio y fin de cada tarea.

En mi caso he optado por realizar 15 horas semanales, que se reflejan en la planificación como 3 horas diarias de lunes a viernes. Aun sabiendo que no podré dedicarle 3 horas al día, he decidido dejarlo así y dedicarle todo lo posible entre semana, y reservando los fines de semana para realizar todas las horas restantes.

Con todo lo mencionado anteriormente, hemos obtenido la planificación que se muestra en la página siguiente:

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
Inicio	0 horas	lun 25/01/16	lun 25/01/16	
Documentación	300 horas	lun 25/01/16	vie 10/06/16	1
Estudio del uso de SonarQube	**60 horas**	**lun 25/01/16**	**vie 19/02/16**	**1**
Buscar información de que es SonarQube y para que se utiliza	6 horas	lun 25/01/16	mar 26/01/16	
Instalación y configuración de SonarQube	3 hora	mié 27/01/16	mié 27/01/16	4
Primeros análisis de proyectos locales	6 horas	jue 28/01/16	vie 29/01/16	5
Probar integraciones con otras herramientas como Bitbucket o Github	12 horas	lun 26/09/16	vie 30/09/16	6
Buscar integración con SourceForge	3 horas	vie 30/09/16	vie 07/10/16	7
Caso práctico con JHotDraw	15 horas	lun 03/10/16	vie 14/10/16	8
Caso práctico con JFreeChart	15 horas	lun 10/10/16	vie 06/01/17	9
Estudio de la extensibilidad	**180 horas**	**lun 17/10/16**	**mié 19/10/16**	**3**
Estudio de XPath	9 horas	lun 17/10/16	jue 27/10/16	
Estudio del mecanismo de extensibilidad basado en Xpath	18 horas	jue 20/10/16	mié 30/11/16	12
Desarrollar regla basada en XPath	**72 horas**	**vie 28/10/16**	**mar 01/11/16**	**13**
Definición de la regla	9 horas	vie 28/10/16	mié 23/11/16	
Desarrollo	**48 horas**	**mié 02/11/16**	**mié 09/11/16**	**15**
Diseño	18 horas	mié 02/11/16	lun 21/11/16	
Implementación	24 horas	jue 10/11/16	mié 23/11/16	17
Pruebas	6 horas	mar 22/11/16	mié 30/11/16	18
Uso de SonarQube buscando proyectos a los que aplicar la nueva regla	15 horas	jue 24/11/16	jue 08/12/16	16
Estudio del mecanismo de extensibilidad basado en Java	18 horas	jue 01/12/16	vie 06/01/17	14
Desarrollar reglaba basada en Java	**63 horas**	**vie 09/12/16**	**mar 13/12/16**	**21**
Definición de la regla	9 horas	vie 09/12/16	lun 02/01/17	
Desarrollo	**42 horas**	**mié 14/12/16**	**mié 21/12/16**	**23**
Diseño	18 horas	mié 14/12/16	jue 29/12/16	
Implementación	18 horas	jue 22/12/16	lun 02/01/17	25
Pruebas	6 horas	vie 30/12/16	vie 06/01/17	26
Uso de SonarQube buscando proyectos a los que aplicar la nueva regla	12 horas	mar 03/01/17	mar 31/01/17	24
Redacción final de la memoria	50 horas	lun 09/01/17	vie 03/02/17	11
Revisión de la memoria y correcciones	10 horas	mar 31/01/17	vie 03/02/17	29
Fin	0 horas	vie 03/02/17	vie 03/02/17	2;30

Tabla 1. Planificación

Riesgos

Los posibles riesgos que pueden surgir durante el proyecto son los siguientes:

<table>
<tr><th colspan="4">Formulario de Gestión de Riesgos 1</th></tr>
<tr><td>Fecha :
21/12/2015</td><td colspan="2">Creador : Héctor Arranz</td><td>Categoría : riesgo de duración</td></tr>
<tr><td>Fase: Desarrollo</td><td>Probabilidad: 60%</td><td>Consecuencia: alta</td><td>Proyecto: SonarQube</td></tr>
<tr><td>Título del riesgo:</td><td colspan="3">No poder dedicarle las horas necesarias a la semana</td></tr>
<tr><th colspan="4">Valoración del riesgo</th></tr>
<tr><td colspan="4">Enunciado del riesgo:</td></tr>
<tr><td colspan="4">Debido a que se compatibiliza el desarrollo del Trabajo fin de Grado, con las prácticas en empresa, es posible que no se puedan dedicar las horas semanales estimadas. Esto también podría ocurrir por causas de enfermedad y/o vacaciones.</td></tr>
<tr><td colspan="4">Contexto del riesgo:</td></tr>
<tr><td colspan="4">Este riesgo podría ocurrir en cualquier fase del proyecto, acentuándose a partir de la Semana Santa, ya que se podría pasar de jornada parcial a jornada completa, lo que reduciría las horas disponibles para poder dedicarlas al TFG.</td></tr>
<tr><td colspan="4">Análisis del riesgo:</td></tr>
<tr><td colspan="4">Podría provocar un retraso en la entrega final, ya que una demora en una tarea provocaría dilaciones en cascada en el resto de tareas.</td></tr>
<tr><th colspan="4">Plan de riesgo:</th></tr>
<tr><td rowspan="2">Estrategia:
O Prevención
O Protección
X Reducción
O Investigación</td><td colspan="3">Plan de acción para el riesgo:</td></tr>
<tr><td colspan="3">Para que el riesgo afecte lo menos posible, evitando así alargar la duración del proyecto, si surgiera (ya que el proyecto está planificado para realizarle entre semana, dejando los fines de semana libres), se dedicarán las horas necesarias durante los fines de semana.</td></tr>
<tr><td colspan="4">Seguimiento del riesgo:</td></tr>
<tr><td colspan="2">Punto de comprobación:</td><td colspan="2">Comentarios:</td></tr>
<tr><td colspan="2">Indicador:</td><td colspan="2" rowspan="6"></td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">Umbral:</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">Disparador:</td></tr>
<tr><td colspan="2"></td></tr>
</table>

Tabla 2. Riesgo 1

Formulario de Gestión de Riesgos 2			
Fecha : 21/12/2015	**Creador :** Héctor Arranz	**Categoría** : riesgo de duración	
Fase: Desarrollo	**Probabilidad:** 80%	**Consecuencia:** alta	**Proyecto:** SonarQube
Título del riesgo:	Tarea más complicada de lo planificado		
Valoración del riesgo			
Enunciado del riesgo:			
Debido al desconocimiento de la complejidad de las tareas planificadas, puede asumirse que una tarea es más sencilla de lo que realmente es, dedicando así más horas de las estimadas y, por tanto, descuadrando la planificación.			
Contexto del riesgo:			
Este riesgo podría ocurrir en la fase de desarrollo, especialmente en la parte de Estudio de la extensibilidad que es la más compleja y desconocida.			
Análisis del riesgo:			
Podría provocar un retraso en la entrega final, ya que una demora en una tarea provocaría dilaciones en cascada en el resto de tareas.			
Plan de riesgo:			
Estrategia: O Prevención X Protección O Reducción O Investigación	**Plan de acción para el riesgo:** Para reducir el impacto de este riesgo, se estimará un tiempo ligeramente superior a las tareas que puedan ser más complejas con el fin de tener un cierto margen de error.		
Seguimiento del riesgo:			
Punto de comprobación:	**Comentarios:**		
Indicador:			
Umbral:			
Disparador:			

Tabla 3. Riesgo 2

<table>
<tr><th colspan="4">Formulario de Gestión de Riesgos 3</th></tr>
<tr><td>Fecha : 21/12/2015</td><td>Creador : Héctor Arranz</td><td colspan="2">Categoría : Riesgo de duración</td></tr>
<tr><td>Fase: Todas</td><td>Probabilidad: 20%</td><td>Consecuencia:
alta</td><td>Proyecto:
SonarQube</td></tr>
<tr><td>Título del riesgo:</td><td colspan="3">Equipo no disponible.</td></tr>
<tr><th colspan="4">Valoración del riesgo</th></tr>
<tr><td colspan="4">Enunciado del riesgo:</td></tr>
<tr><td colspan="4">Debido a que el equipo con el que se trabajará se utiliza para otras tareas, y es portátil, puede darse el caso de que en algún momento no se encuentre disponible para trabajar con él, ya que puede haber tenido que ser formateado, puede sufrir alguna caída por el transporte… o cualquier otra eventualidad.</td></tr>
<tr><td colspan="4">Contexto del riesgo:</td></tr>
<tr><td colspan="4">El riesgo se podría dar en cualquier momento del desarrollo del proyecto.</td></tr>
<tr><td colspan="4">Análisis del riesgo:</td></tr>
<tr><td colspan="4">No contar con el equipo de trabajo cuando se le necesita, y sin saber durante cuánto tiempo, puede provocar un retraso en la entrega final.</td></tr>
<tr><th colspan="4">Plan de riesgo:</th></tr>
<tr><td>Estrategia:
X Prevención
O Protección
O Reducción
O Investigación</td><td colspan="3">Plan de acción para el riesgo:

Para prevenir el impacto de este riesgo, se intentará, en la medida de lo posible, no instalar más que el software necesario para evitar problemas de rendimiento, virus y demás que inhabiliten su uso. Además, se evitará transportarle lo menos posible evitando así problemas de hardware roto.</td></tr>
<tr><th colspan="4">Seguimiento del riesgo:</th></tr>
<tr><td colspan="2">Punto de comprobación:</td><td colspan="2">Comentarios:</td></tr>
<tr><td colspan="2">Indicador:</td><td colspan="2" rowspan="6"></td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">Umbral:</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">Disparador:</td></tr>
<tr><td colspan="2"></td></tr>
</table>

Tabla 4. Riesgo 3

Formulario de Gestión de Riesgos 4			
Fecha : 21/12/2015	**Creador :** Héctor Arranz	**Categoría** : Riesgo de proyecto	
Fase: Desarrollo	**Probabilidad:** 60%	**Consecuencia:** muy alta	**Proyecto:** SonarQube
Título del riesgo:	Problemas para resolver una tarea concreta.		

Valoración del riesgo
Enunciado del riesgo:
Al ser un proyecto con un alto contenido de investigación, puede que en alguna de las tareas se pretenda hacer algo demasiado complicado o inviable.
Contexto del riesgo:
Este riesgo se podría dar en la fase de desarrollo, más concretamente en la parte de generar nuevas reglas para SonarQube.
Análisis del riesgo:
Podría provocar tener que volver a replantear la tarea, pudiéndola cambiar completamente, lo que implica empezar la tarea desde el principio y retrasar la entrega final.

Plan de riesgo:	
Estrategia: X Prevención O Protección O Reducción O Investigación	**Plan de acción para el riesgo:** Para prevenir el impacto de este riesgo, a la hora de elegir qué reglas se desarrollarán, habrá que ser lo más realista posibles, eligiendo, de esta manera las que sean más sencillas, teniendo en cuenta que luego se pueden complicar.

Seguimiento del riesgo:	
Punto de comprobación:	**Comentarios:**
Indicador:	
Umbral:	
Disparador:	

Tabla 5. Riesgo 4

Presupuesto

A continuación se desglosa el presupuesto necesario para realizar el proyecto:

- Costes hardware
  - Portátil ASUS F550DP.....21,50€
  - Portátil ASUS X556U.....58,40€
- Costes Software
  - Licencia MS Project Professional 2013.....95,07€
  - Licencia Office 2013.....8,00€
- Costes personales
  - Sueldo programador Junior .....5.360€
- Descuentos
  - Licencia MS Project Professional 2013..... -95,07€
  - Licencia Office 2013..... -8,00€

Subtotal .....5.439,9€

Margen error..... 25%

TOTAL .....6799,86€

Explicación de cómo se obtienen los costes anteriores:

- Portátil ASUS F550DP tiene un coste total de 600€ con una vida útil de 28 meses Lo que supone un coste mensual de 600€/28meses = **21,43€/mes**. El tiempo que duró este ordenador hasta que se rompió fue de 1 mes aproximadamente, por lo que el coste total se puede estimar en **21,50€**
- Portátil ASUS X556U tiene un coste total de 700€ con una vida útil aproximada de 4 años (48 meses). Lo que supone un coste mensual de 700€/48meses = **14,60€/mes**. La duración total del proyecto, como ya se vio anteriormente, se estima en unos 5 meses, y restándole el mes que duró el otro ordenador mencionado anteriormente nos quedan 4 mese, por lo que el coste será de 4meses*14,60€/mes = **58,40€.**
- Project Professional 2013, según la web oficial de Microsoft, tiene un precio para el cliente de escritorio para jefes de proyecto de 1.369,00€ y esperamos que tenga una vida útil de 6 años (72 meses). Prorrateando al tiempo de duración del proyecto, de la misma manera que en el punto anterior, obtenemos un precio total de **95,07€.**
- Office 2013 según la web de amazon.com tiene un coste de 115,29€ con una vida útil de unos 6 años (72 meses). Realizando los cálculos del punto anterior para prorratear el precio a la duración del proyecto obtenemos un total de **8,00€.**
- Sueldo programador Junior. Buscando varias ofertas de programador Junior en España, y observando su sueldo, se estima que este perfil de programador cobra unos 12.000-18.000€ brutos al año. Tomaremos como referencia la media (15.000€). Mediante una web de cálculo online de sueldo neto a partir del sueldo bruto, obtenemos un sueldo mensual neto de 1.072€, en el cual ya están descontados tanto las cuotas a la Seguridad Social como las retenciones del IRPF, con la normativa vigente durante el año 2015. Dicho sueldo, durante 5 meses de duración del proyecto serán **5.360€.**

- Descuentos. Debido a que el proyecto se realiza como estudiante de la UVa, y esta ofrece de manera gratuita tanto la licencia del Office 2013 como la de MS Project Professional 2013, se descontaran el precio integro de dichas licencias en el cálculo del subtotal.

A estos cálculos se le aplica un margen de error, por si la precisión no es la adecuada, del 25%. Por ello, al total habría que sumarle el 25% del mismo.

Seguimiento

Aunque en un principio estaba planificado que haría unas 15 horas semanales, en la práctica no fue así. Iba avanzando en mis ratos libres, por lo que finalmente han sido unas pocas horas cada semana, incluyendo semanas en el que no realicé nada. Esto es debido a que me gustaba mucho la parte de investigar y descubrir cosas nuevas, pero se me complicaba a la hora de documentar todo lo que he aprendido, por lo que acababa aplazándolo.

En la siguiente tabla muestro para cada tarea las horas planificadas, comparándolas con las reales, obteniendo así la desviación. He omitido las fechas de inicio y fin reales ya que como no ha sido un trabajo constante en el tiempo no ofrecen mucha información.

Nombre de la tarea	Horas estimadas	Horas reales	Diferencia
Buscar información de que es SonarQube y para que se utiliza	6	21	15
Instalación y configuración de SonarQube	3	14	11
Primeros análisis de proyectos locales	6	17	11
Probar integraciones con otras herramientas como Bitbucket o Github	12	35	23
Buscar integración con SourceForge	3	1	-2
Caso práctico con JHotDraw	15	25	10
Caso práctico con JFreeChart	15	13	-2
Estudio de Xpath	9	10	1
Estudio del mecanismo de extensibilidad basado en Xpath	18	19	1
Desarrollar regla basada en Xpath	72	41	-31
Estudio del mecanismo de extensibilidad basado en Java	18	25	7
Desarrollar reglaba basada en Java	63	25	-38
Redacción final de la memoria	50	42	-8
Revisión de la memoria y correcciones	10	13	3
TOTAL	**300**	**301**	

Tabla 6. Desviación en horas de las tareas

Se puede deducir de esta tabla que la parte de estudio fue subestimada, mientras que la parte de desarrollo estaba sobreestimada.

En cuanto a la parte de riesgos, tenía cuatro posibles riesgos que podían suceder, y desgraciadamente se cumplieron los cuatro.

- Riesgo 1 - No poder dedicarle las horas necesarias a la semana: se incumplió casi desde el principio del proyecto. 15 horas semanales eran más de las que podía dedicarle. Viendo que aún tenía tiempo para acabar, opté por ir desarrollando cada vez que tuviera tiempo, sin importar si dedicaba 1, 15 o 30 horas semanales, el objetivo era acabar.
- Riesgo 2 - Tarea más complicada de lo planificado: al igual que el riesgo 1, también se incumplió desde las primeras tareas, ya que como he comentado eran las que estaban subestimadas, pensando que buscar y comprender en qué consistía SonarQube sería más fácil.
- Riesgo 3 - Equipo no disponible: era el riego con menos posibilidades de suceder, y sucedió. No cumplí la estrategia de prevención, lo que provocó estar 3 meses sin ordenador, en los cuales no pude hacer nada.
- Riesgo 4 - Problemas para resolver una tarea concreta: hubo una tarea, concretamente la de aprender a usar los plugins con GitHub y Bitbucket que se complicó en exceso, ya que no era lo que pensaba en un principio. Después de dedicarle un 192% más del tiempo planificado, opte por no continuar, y quedarme con lo que había descubierto. Es una tarea que no se pudo terminar al completo.

Motivo de retraso

Viendo la tabla de planificación, se puede ver claramente que hay 6 meses sin actividad, comprendidos entre finales de Enero de 2016 y finales de Septiembre del mismo año. En un principio comencé el proyecto a principios de Enero de 2016, compaginándolo con las prácticas, las cuales realizaba a media jornada. En ese momento todo iba perfecto, pero el 15 de Febrero de ese año empecé a jornada completa, por lo que después de 8 horas en la oficina trabajando con el ordenador, cuando llegaba a las 19:30h. a casa**,** muchos días era incapaz de avanzar en el proyecto, por lo que el ritmo descendió.

Después, cuando sacaba ratos para avanzar con las tareas, se me ocurrió la idea de usar una máquina virtual, para repetir varias veces las pruebas de instalación, y tener claro cuáles son las dependencias de SonarQube, pero vi que mi memoria RAM era insuficiente, por lo que opté por duplicarla por mí mismo. Por alguna razón quemé la placa base, y estuvo 2 meses en una reparadora que, finalmente, la dio por irreparable. Después de esto, estuve un mes buscando otro ordenador que mereciera la pena gastarse el dinero que tenía pensado. Además de que tuviera unas características que no lo dejasen desfasado en un año. Esta fue la razón por la que estuve 3 meses sin ordenador y sin poder avanzar nada.

Seguidamente, justo cuando compré el nuevo ordenador, me mudé de vivienda, ya en verano, y en la cual estuve 2 meses más sin internet, por lo que tampoco pude avanzar nada en este tiempo. Finalmente, a principios de Septiembre, es cuando tuve vacaciones en la empresa, por lo que desconecté del todo.

Fue ya a mediados de Septiembre de 2016 con la vuelta a la rutina, adaptado a mi nueva casa, con internet y portátil nuevo cuando decidí retomar el proyecto, volviendo a replanificar las fechas de las tareas.

3. QUÉ ES SONARQUBE Y PARA QUÉ SE UTILIZA

SonarQube es una plataforma para gestionar la calidad del código de un proyecto. Es software libre, y se basa en diferentes herramientas de análisis estático de código para obtener métricas sobre el código fuente y poder mejorar la calidad del mismo.

Para ello analiza 7 ejes fundamentales del software, ofreciendo informes con los resultados obtenidos de los siguientes apartados [1]:

- Código duplicado: es capaz de, según el nivel de precisión que le indiquemos, encontrar partes de código duplicado en el proyecto, lo cual ya sabemos que es una mala práctica pues, si en algún futuro hay que cambiar dicho fragmento y éste está replicado por varios sitios del proyecto, tendremos que ir buscando uno por uno y modificarlo.
- Reglas del lenguaje: cada lenguaje tiene un estándar a seguir que, aunque no es obligatorio, deberemos aplicarle. Este estándar define cómo se nombran las variables, los métodos, las clases, cómo deben ser los comentarios, la indentación, entre otras muchas cosas. SonarQube analiza si seguimos el estándar del lenguaje que estemos usando y, si tenemos algún fallo, nos informa de él para que lo solucionemos. Estos fallos no deben suponer ningún problema en el funcionamiento del software que estamos desarrollando, ya que son reglas pensadas para facilitar la vida de los desarrolladores a la hora de entender el código.
- Cobertura de test automáticos: una buena práctica a la hora de desarrollar software es la de crear test de prueba. Estos tests comprueban automáticamente que todo el software funciona como nosotros esperamos que lo haga. Para ello, por ejemplo, comprueba que todos los métodos o funciones devuelven el valor esperado según los parámetros que le pasemos, entre otras cosas. Por eso es bueno tener tests preparados desde el primer día que prueben toda la funcionalidad ya desarrollada, para que así, a la hora de hacer un nuevo cambio para mejorar algo, corregir un error o para implementar una nueva funcionalidad, podamos asegurarnos que todo lo que ya funcionaba sigue funcionando y no hemos roto nada.

  Estos tests tienen que pasar al menos una vez por cada instrucción del código, tienen que recorrer todas las bifurcaciones, tiene que comprobar todas las condiciones booleanas. Entonces llamaremos cobertura al porcentaje de instrucciones sobre el total, que es comprobado por los test automáticos, y tiene que ser lo más alto posible. SonarQube nos ofrece un informe de la cobertura de los test, en el que podemos ver qué cantidad de código tenemos testeada, qué partes del código no son analizadas por ningún test, qué condiciones booleanas son probadas parcialmente, entre otras cosas. Para ello se basa en los reportes sobre cobertura que pueden generar herramientas como JaCoCo.
- Complejidad del código: que un código fuente sea complejo no significa que sea mejor; al contrario, cuanto más sencillo sea de leer y entender, mejor código será, ya que, como se comentaba anteriormente, es más mantenible. La complejidad a la que nos referimos, que es la que calcula SonarQube, es la complejidad ciclomática. Esta se define a grandes rasgos como el número de caminos independientes dentro de un programa o fragmento de código. Es decir, todos los flujos que puede tomar la ejecución del código. Por ejemplo, cuando se está ejecutando un conjunto de instrucciones está siguiendo un camino, y cuando encuentra un if, este camino se divide en dos, la parte que está dentro del if, y la parte del else, ya que no se pueden ejecutar las dos de una sola vez. Por lo tanto, en un código secuencial, si hay un if, habrá dos caminos, la ejecución que pasa por el if y la ejecución que pasa por el else.

SonarQube, a parte del if, también tiene en cuenta como bifurcaciones las siguientes instrucciones a la hora de analizar un proyecto Java: for, while, case, catch, throw, return (si no es la última instrucción de un método), &&, ||, ?. También cuenta cada cabecera de un método como un camino.

SonarQube calcula cuatro tipos de complejidades ciclomática: la complejidad ciclomática de método, la complejidad ciclomática de clase, la de fichero, y la total del proyecto.

- Errores potenciales: Existen otras reglas, que tampoco son obligatorias de cumplir, pero ello podría ocasionar algún problema de funcionamiento del software en un futuro. Un ejemplo claro en Java, es no controlar un posible null en una variable, considerando que siempre tendrá un valor con el que trabajar. Si no se controla dicho null va a compilar, y, posiblemente, cuando se pruebe todo funcione correctamente, pero podría darse el caso de que, por alguna razón, esa variable sea null y, al intentar acceder a algún método suyo, de un error y deje de funcionar. Esto es lo que llamaremos “Errores potenciales”, posibles deficiencias en el código que, en algún momento, podrían causar un mal funcionamiento del software.

- Comentarios: Poner comentarios en el código fuente es necesario, pero tan malo es poner muchos, como no poner ninguno. Los comentarios tienen que ser para explicar algo que no se puede deducir fácilmente del código, no para explicar paso a paso lo que hace un fragmento de código, ya que para eso leemos el fragmento. También hay que comentar de una manera diferente los métodos públicos (denominados APIs), al menos en Java. En estos comentarios se explicará para qué sirve el método, qué parámetros hay que pasarle, y qué retorna. Eso es para que otro programador pueda usar dicho método y poder ver para qué sirve, sin ver el código (ya que a veces no se podrá ver como está implementado, lo único que se podrá ver es esta documentación).

  SonarQube nos muestra la cantidad de líneas de comentarios que hay en el proyecto y el porcentaje que representa sobre el total de líneas. Ya queda en mano de cada uno elegir qué porcentaje es admisible para el proyecto. Un porcentaje del 50% indica que la mitad de líneas son de código y la otra mitad es de comentarios.

  SonarQube también ofrece información sobre las APIs, tal como el número de APIs en el proyecto (suma de clases públicas, métodos públicos y propiedades públicas), el número de estas sin documentar, y el porcentaje de APIs documentadas.

- Diseño y arquitectura: A la hora de desarrollar software, una de las buenas prácticas que se debe seguir es la de “bajo acoplamiento”, que consiste en tener las clases del proyecto lo menos ligadas entre sí, para poder tener una alta modularidad. Así, si en un futuro se necesita cambiar una clase por otra similar, pero con una implementación más eficiente, por ejemplo, sea una tarea sencilla al estar poco acopladas.

  Cuando se empieza un desarrollo software, es fácil mantener este bajo acoplamiento, pero, a medida que crece, si no se pone especial atención, puede que el diseño se empiece a complicar, llegando un momento a partir del cual, el cambio de una clase, implique la modificación de otra gran cantidad de clases que dependían de la primera.

  También existen lo que se conoce como ciclos de dependencias entre paquetes, en el que algunas clases de un paquete A dependen de otras clases de otro paquete B que, a su vez, dependen de clases de un paquete C en el que hay dependencias del primer paquete A. Esto es algo que debemos evitar [2], ya que reduce la modularidad del proyecto, hace que este sea menos mantenible y más difícil de testear. SonarQube nos muestra la

cantidad de ciclos de dependencias entre paquetes que hay en nuestro proyecto y el índice de interdependencia entre paquetes, que nos da una idea del grado de acoplamiento general del proyecto. Un valor del 0% indica que no hay ciclos en el proyecto, y un 100% significa que los paquetes están muy acoplados.

SonarQube también nos muestra qué dependencias entre paquetes y archivos hay que eliminar para mejorar el diseño.

Hace ya tiempo SonarQube se convirtió en la herramienta de facto para la mejora de la calidad de un producto software.

SonarQube consiste en una aplicación web, y usa una base de datos para almacenar la información obtenida de cada análisis, pudiendo obtener así, un histórico de la evolución del proyecto. Por defecto viene con el gestor de bases de datos H2, pero solo es recomendable para pruebas. Si queremos usar SonarQube en un entorno de producción, es más aconsejable usar algunos de los gestores compatibles: MySQL, Oracle, PostgreSQL, SQLServer.

Aunque existe la creencia de que solo vale para Java, no es así. También soporta otros lenguajes como JavaScript, PHP, Cobol, C#... y, para ello, utiliza plugins para extender la funcionalidad de SonarQube.

Además de plugins para extender los lenguajes soportados, también se pueden extender otro tipo de funcionalidades. Por ejemplo, para poder analizar un proyecto, no es necesario tenerle en local; existen plugins para poder acceder a un proyecto guardado en un repositorio en la nube. Algunos de estos plugins, son los que permiten a SonarQube analizar un proyecto almacenado en Bitbucket, o en GitHub, entre otros.

Aunque SonarQube puede utilizarse en el momento que se desee, está más pensado para usarlo junto con una herramienta de gestión y construcción de proyectos software, entre los que se encuentra Maven, Ant y Gradle; y herramientas de integración continua como Bamboo, Jenkins y Travis. Para ello, también existen plugins para cada una de estas herramientas.

SonarQube realiza muchas cosas pero ahora vamos a ver algunas ideas equivocadas de lo que no hace SonarQube [3]:

- **SonarQube no es una herramienta de construcción.**

  Ya existen herramientas para la gestión del ciclo de vida del software como Maven. SonarQube ya espera que el proyecto esté compilado, que haya pasado los test. SonarQube se ocupa del análisis de la calidad sin preocuparse cómo se compila, cómo se limpia, etc.

- **SonarQube no es solamente una herramienta de análisis estático de código.**

  SonarQube se puede integrar con herramientas externas que le provean información sobre análisis dinámico de código.

- **SonarQube no es una herramienta de cobertura de código**

  Aunque provee información sobre la cobertura de los test, SonarQube no es capaz de hacer esos cálculos. Para ello se basa en herramientas externas como Jacoco.

- **SonarQube no es una herramienta de formateo de código.**

  SonarQube no modifica el código. Simplemente se limita a analizarlo, y algunos de esos análisis se centran en el formateo del código para ver si cumple con los estándares establecidos.

- **SonarQube no es solo una herramienta de revisión manual de código**

  Se puede integrar con herramientas como Jenkis o Travis para que pueda revisar el código automáticamente cuando suceda algún evento.

- **SonarQube no solo gestiona la calidad de código Java.**

  SonarQube está desarrollado en Java, pero ya es capaz de analizar numerosos lenguajes de programación como C#, PHP, JavaScript…

- **SonarQube no necesita Maven para analizar tu proyecto.**

  Se puede analizar un proyecto usando Maven, pero no es la única manera. También se puede analizar mediante o Ant o a través de una herramienta de SonarQube, entre otras.

- **SonarQube no solo usa reglas para validar la calidad.**

  También realiza cálculos para obtener la complejidad ciclomática, código duplicado, deuda técnica…

- **SonarQube no solo permite ver la calidad desde la aplicación web.**

  Existen extensiones para varios IDEs como Eclipse, IntelliJ, VisualStudio, que permite a los desarrolladores ver en los entornos de desarrollo si van cumpliendo las normas o no, pudiendo ejecutarse así análisis locales para revisar el código antes de subirlo a un repositorio, por ejemplo.

Actualización: cuando empecé el proyecto y buscaba qué era SonarQube y qué es lo que analizaba, en la misma documentación oficial mencionaban la parte de diseño y arquitectura que he descrito anteriormente. A través de la documentación oficial y otras webs aprendí un poco más detalladamente en qué consistía.

Una vez acabado el proyecto me he dado cuenta que en toda la herramienta no hay ninguna medida sobre ese tema. Por lo que investigué qué pasó con ella, y resulta que la eliminaron a partir de la versión 5.2, por lo que ya no podemos obtener esas medidas.

4. INSTALACIÓN Y CONFIGURACIÓN

En este apartado se va a explicar cómo instalar y dejar totalmente configurado SonarQube para un primer análisis de un proyecto en local, tanto con herramientas como Maven o Ant como sin ninguna de estas.

Instalar Java

SonarQube es un software desarrollado en Java, y como todos los programas desarrollados en este lenguaje, necesita una maquina virtual de Java para poder ejecutarse.

En un principio, para poder ejecutar SonarQube sería suficiente con instalar el JRE (Java Runtime Environment). Pero como en un futuro vamos a desarrollar nuevas reglas escritas en Java, y necesitaremos desarrollar con este lenguaje, voy a optar por instalar el JDK (Java Development Kit), que es el que necesitamos para poder desarrollar en Java, el cual incluye el JRE.

Usaré la versión más reciente del JDK que en el momento de la realización de este proyecto es la 8u121.

En el siguiente enlace explican cómo instalar el JDK y cómo añadirlo al Path en Windows, para poder usar así las herramientas del JDK en la consola.

http://elclubdelautodidacta.es/wp/2013/03/instalacion-del-kit-de-desarrollo-java-jdk-en-windows/

Además de añadir la ruta de los binarios del JDK al Path, algunas herramientas como Maven necesitan conocer el directorio del JDK completo y no solo el de los binarios. Por eso vamos a crear una nueva variable de entorno llamada JAVA_HOME, cuyo valor será la ruta al JDK completo. En mi caso sería la siguiente ruta: C:\Program Files\Java\jdk1.8.0_121

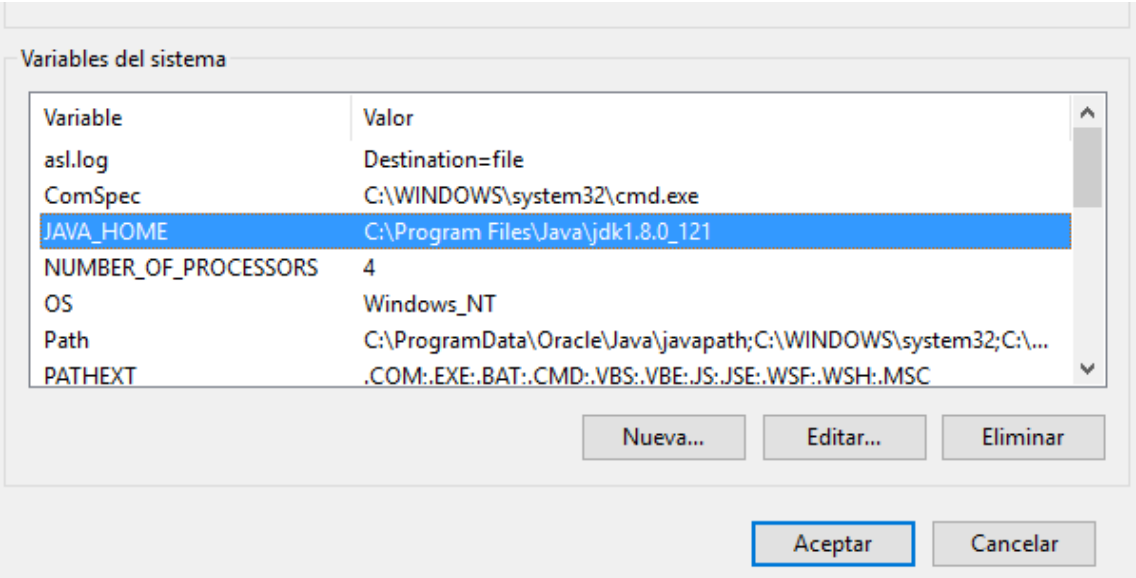


Ilustración 1. Variables de sistema en Windows, JAVA_HOME

Instalar SonarQube

SonarQube es muy sencillo de instalar, y no necesita ningún instalador [4].

Accedemos a la web de SonarQube: https://www.sonarqube.org/ y descargamos la versión 6.3.

Obtendremos un ZIP que contiene una carpeta con el programa. Copiaremos dicha carpeta en nuestro disco duro, donde deseemos. En mi caso la pondré en C:\ de tal manera que el directorio de SonarQube será: C:\sonarqube-6.3

Para poder ejecutar SonarQube desde cualquier ruta en la consola, tendremos que añadir a las variables de entorno la ruta de los binarios. Cada Sistema Operativo tendrá una ruta diferente, y están todas dentro de la carpeta bin del programa. Actualmente existen las siguientes:

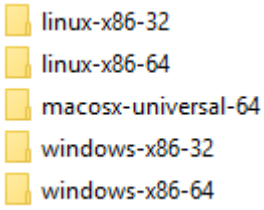


Ilustración 2. Contenido carpeta bin de SonarQube

En mi caso usaré: C:\sonarqube-6.3\bin\windows-x86-64

Debemos añadir esa ruta en la variable Path, dentro de las variables de entorno, al igual que hicimos cuando instalamos Java.

Una vez finalizado ya tendremos instalado SonarQube. Solo nos falta abrir una consola y ejecutar el comando **StartSonar** para arrancar SonarQube.

Una vez arranque todos los servicios nos saldrá lo siguiente en la consola:

```
Fraction=75 -XX:+UseCMSInitiatingOccupancyOnly -XX:+HeapDumpOnOutOfMemoryError -Djava.io.t
mpdir=C:\sonarqube-6.3\temp -javaagent:C:\Program Files\Java\jre1.8.0_101\lib\management-a
gent.jar -cp ./lib/common/*;./lib/search/* org.sonar.search.SearchServer C:\sonarqube-6.3\
temp\sq-process7269794079277594760properties
jvm 1    | 2017.04.01 23:27:50 INFO  app[][o.s.p.m.Monitor] Process[es] is up
jvm 1    | 2017.04.01 23:27:50 INFO  app[][o.s.p.m.JavaProcessLauncher] Launch process[web
]: C:\Program Files\Java\jre1.8.0_101\bin\java -Djava.awt.headless=true -Dfile.encoding=UT
F-8 -Xmx512m -Xms128m -XX:+HeapDumpOnOutOfMemoryError -Djava.io.tmpdir=C:\sonarqube-6.3\te
mp -javaagent:C:\Program Files\Java\jre1.8.0_101\lib\management-agent.jar -cp ./lib/common
/*;./lib/server/*;C:\sonarqube-6.3\lib\jdbc\h2\h2-1.3.176.jar org.sonar.server.app.WebServ
er C:\sonarqube-6.3\temp\sq-process3285987448660427220properties
jvm 1    | 2017.04.01 23:28:05 INFO  app[][o.s.p.m.Monitor] Process[web] is up
jvm 1    | 2017.04.01 23:28:05 INFO  app[][o.s.p.m.JavaProcessLauncher] Launch process[ce]
: C:\Program Files\Java\jre1.8.0_101\bin\java -Djava.awt.headless=true -Dfile.encoding=UTF
-8 -Xmx512m -Xms128m -XX:+HeapDumpOnOutOfMemoryError -Djava.io.tmpdir=C:\sonarqube-6.3\tem
p -javaagent:C:\Program Files\Java\jre1.8.0_101\lib\management-agent.jar -cp ./lib/common/
*;./lib/server/*;./lib/ce/*;C:\sonarqube-6.3\lib\jdbc\h2\h2-1.3.176.jar org.sonar.ce.app.C
eServer C:\sonarqube-6.3\temp\sq-process3984991021774395232properties
jvm 1    | 2017.04.01 23:28:13 INFO  app[][o.s.p.m.Monitor] Process[ce] is up
jvm 1    | 2017.04.01 23:28:13 INFO  app[][o.s.application.App] SonarQube is up
```

Ilustración 3. Consola de arranque de SonarQube

Lo que nos indica que SonarQube ya está listo para usar. Solo tenemos que acceder a la siguiente URL en cualquier navegador: http://localhost:9000/ y veremos la página principal de SonarQube.

Instalar Base de Datos

Aunque ya hemos visto en el apartado anterior que no es necesario que instalemos ninguna base de datos para poder usar SonarQube ya que, por defecto, trae una integrada, desde SonarQube nos recomiendan que esta base de datos integrada se use solo para pruebas, como voy a hacer en este proyecto, No obstante, para entornos reales se debe usar una base de datos externa al programa.

En mi caso voy investigar cómo usar SonarQube con MySQL, por si fuera necesario configurarlo en un entorno de producción, aunque SonarQube también admite otras bases de datos como Oracle, SQLServer, PostgreSQL.

Voy a partir de que ya tenemos un servidor MySQL instalado y debidamente configurado.

Mi servidor lo tendré activo en mi local, escuchando en el puerto 3306. Pero no es obligatorio que esté en el mismo servidor que SonarQube, ni que sea el puerto 3306 para conectarse al mismo.

Para que SonarQube pueda usar MySQL, deberemos crear una base de datos en nuestro servidor MySQL, donde luego se crearán automáticamente las tablas necesarias. También crearemos un usuario específico para SonarQube, con los permisos necesarios.

Tendremos que entrar en una consola de MySQL y ejecutar los siguientes comandos:

```
CREATE DATABASE sonarqube CHARACTER SET utf8 COLLATE utf8_general_ci;
CREATE USER 'sonarqube' IDENTIFIED BY 'sonarqube';
GRANT ALL ON sonarqube.* TO 'sonarqube'@'%' IDENTIFIED BY 'sonarqube';
GRANT ALL ON sonarqube.* TO 'sonarqube'@'localhost' IDENTIFIED BY 'sonarqube';
FLUSH PRIVILEGES;
```

Con el primero creamos una base de datos llamada "*sonarqube*" con una codificación UTF8.

El segundo crea un usuario llamado "*sonarqube*".

El tercero da todos los permisos disponibles al usuario *sonarqube* sobre la base de datos *sonarqube,* sin importar desde qué IP se esté conectando con nuestro servidor MySQL.

El cuarto es idéntico al tercero, pero permitiendo la conexión del usuario sonarqube desde *localhost*.

Por último, forzamos al servidor a que refleje los cambios sobre los usuarios y sus privilegios**,** sin necesidad de reiniciar el servidor.

Ahora que ya tenemos todo configurado en MySQL, debemos configurar SonarQube para que se conecte a la nueva base de datos creada con el usuario que hemos generado para ello.

Tendremos que ir a la carpeta donde tenemos instalado SonarQube y dentro tendrá una carpeta llamada conf.

Buscamos el archivo sonar.properties dentro de dicha carpeta.

Lo abrimos con un editor de texto. Es un archivo muy simple que contiene pares clave-valor. Las líneas que comienzan con el símbolo # se consideran comentarios, y se descartan.

Haremos los siguientes cambios en el archivo:

- Establecemos el usuario y contraseña para conectarnos contra la base de datos. Buscamos las siguientes líneas:

```
#sonar.jdbc.username=
#sonar.jdbc.password=
```

  Las descomentamos, y en el valor ponemos el usuario y contraseña respectivamente que habíamos creado anteriormente en MySQL.

- Después buscamos en una de las líneas comentadas del archivo sonar.properties la clave que activa la base de datos MySQL, la cual tiene un aspecto similar a este:

```
#sonar.jdbc.url=jdbc:mysql://localhost:3306/sonar?useUnicod
e=true&characterEncoding=utf8&rewriteBatchedStatements=true
&useConfigs=maxPerformance
```

  Descomentamos la línea, y vamos a configurar la dirección de conexión. Como la base de datos está en nuestra máquina, dejaremos *localhost*. En otro caso, habría que poner la IP donde responde la base de datos. El puerto dejamos el que venía por defecto, el 3306, por lo que este valor también lo dejamos tal cual. En caso de haber cambiado el número de puerto, tendríamos que indicarlo ahí. Finalmente entre el símbolo / y el ? vemos que aparece la palabra *sonar*. Este es el nombre de la base de datos a la que nos queremos conectar. Si le hubiéramos puesto otro nombre a la base de datos, como en nuestro caso que la hemos llamado '*sonarqube'*, tendremos que ponerlo ahí.

Guardamos los cambios y ya tenemos configurado SonarQube.

Nuestro archivo de sonar.properties quedaría así:

```
# User credentials.
# Permissions to create tables, indices and triggers must be granted to JDBC
user.
# The schema must be created first.
sonar.jdbc.username=sonarqube
sonar.jdbc.password=sonarqube

#----- Embedded Database (default)
# H2 embedded database server listening port, defaults to 9092
#sonar.embeddedDatabase.port=9092
#----- MySQL 5.6 or greater
# Only InnoDB storage engine is supported (not myISAM).
# Only the bundled driver is supported. It can not be changed.
sonar.jdbc.url=jdbc:mysql://localhost:3306/sonarqube?useUnicode=true&characterEn
coding=utf8&rewriteBatchedStatements=true&useConfigs=maxPerformance
```

El resto de líneas que no salen en este fragmento están comentadas, por lo que las obviamos.

Una vez tenemos todo configurado debemos volver a arrancar SonarQube. Si por alguna razón no hemos puesto la configuración como debe de ser, o no puede conectar con la base de datos, el arranque fallará. Por lo que sabemos que si arranca es que está usando correctamente la configuración que pusimos.

```
jvm 1    | 2017.04.01 23:51:46 INFO  app[][o.s.p.m.Monitor] Process[es] is stopping
jvm 1    | 2017.04.01 23:51:46 ERROR app[][o.s.p.m.Monitor] Process[web] failed to start
jvm 1    | 2017.04.01 23:51:47 INFO  app[][o.s.p.m.Monitor] Process[es] is stopped
wrapper  | <-- Wrapper Stopped

C:\Users\User>
```

Ilustración 4. Consola fallo de arranque SonarQube

Cuando está usando la base de datos interna, dentro de la aplicación, al pie de cada página, aparece un mensaje indicando que no se debe usar la base de datos de pruebas.

Si estamos usando correctamente la base de datos MySQL, este mensaje ya no aparecerá.

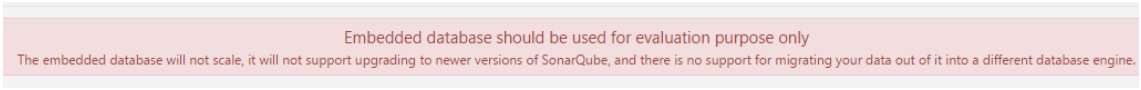


Ilustración 5. Aviso web de uso de base de datos embebida

Instalar SonarQube Scanner

Hasta ahora hemos instalado la aplicación web de SonarQube, con la cual podemos ver el resultado de los análisis del proyecto a través de un navegador, pero con esta aplicación no podemos realizar un análisis. Vamos a tener que instalar otra herramienta específica para esto. Esta herramienta se conoce como SonarQube Scanner (Anteriormente SonarQube Runner) y se descarga por separado de SonarQube desde este enlace:

https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner

Al igual que SonarQube, es un archivo ZIP, que contiene una carpeta que descomprimiremos en un lugar accesible como por ejemplo C:\

En este caso es necesario incluir la ruta de los binarios de SonarQube Scanner en el Path del sistema para no tener que mover de carpeta el proyecto que queremos analizar. Para ello realizamos los mismos pasos que hicimos para añadir SonarQube al Path.

Finalmente, para comprobar que se ha instalado bien, abrimos una nueva consola (si ya teníamos una consola abierta antes de añadir la ruta al Path, no funcionará aún), y ejecutamos el comando **sonar-scanner –h,** el cual nos debe dar una ayuda de cómo usarlo, si es que está bien instalado.

Instalar Ant

Debemos descargar Ant desde su web: http://ant.apache.org/bindownload.cgi

En mi caso he elegido la versión 1.10.1 en formato ZIP.

Este archivo contiene una carpeta la cual debemos descomprimir en el lugar donde queremos instalar Ant.

En las variables de entorno crearemos una nueva llamada ANT_HOME, al igual que hicimos cuando instalamos Java. Esta nueva variable tendrá por valor la ruta donde tenemos instalado Ant, en mi caso la siguiente: C:\apache-ant-1.10.1

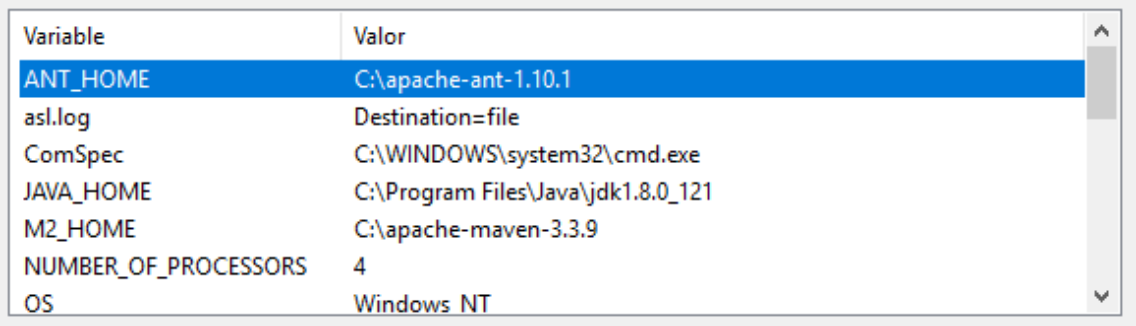


Ilustración 6. Variable de sistema en Windows ANT_HOME

Seguidamente en la variable Path, añadimos la ruta de los binarios de Ant en base a la variable que acabamos de crear. Será así: %ANT_HOME%\bin.

Para poder realizar análisis de proyectos con Ant, necesitaremos una biblioteca para poder ejecutar el target de Ant que lanzará el análisis. Los target son un conjunto de acciones que ejecutará Ant para cumplir un objetivo. Un target de ejemplo puede ser “compilar”. Estos son creados por el desarrollador, indicando los pasos que se deben hacer cuando se ejecuta.

Lo recomendable es añadir esta biblioteca en ${user.home}/.ant/lib, que es uno de los directorios donde Ant busca las bibliotecas de usuario necesarias para ejecutar target. En mi caso será la ruta: C:\Users\Hector\.ant\lib

Para generar esta carpeta en Windows, necesitamos usar la consola, ya que el nombre .ant no lo admite el explorador de Windows a la hora de crear una carpeta. Para ello, abrimos una nueva consola, que por defecto ya está situada en la carpeta del usuario. Simplemente ejecutamos el comando **mkdir .ant\lib** y ya se generan ambas carpetas.

Añadimos el .jar que se puede descargar aquí:

https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+Ant

Esta biblioteca no es más que un *wrapper* de sonar-scanner. Cuando se ejecuta el target de Ant y éste llama a la biblioteca, internamente lo que hace es usar sonar-scanner, pasándole los parámetros que le hemos configurado en el build.xml de Ant.

Solo nos falta comprobar que Ant funciona correctamente. En una nueva consola ejecutamos el comando **ant –h** y debemos obtener una lista de las opciones que se pueden usar con el comando. Si no nos aparece esta lista, algo está mal instalado.

Instalar Maven

Para instalar Maven tendremos que seguir los mismos pasos que en el punto anterior.

Descargamos Maven desde su web: https://maven.apache.org/download.cgi en mi caso la versión 3.3.9 en formato ZIP.

Dentro del ZIP hay una carpeta que tenemos que descomprimir en el lugar donde queremos instalarlo.

En la variables de entorno debemos crear una nueva llamada M2_HOME que indique la ruta donde hemos descomprimido el ZIP, y añadir al Path la ruta de los binarios de Maven en relación a la anterior variable: %M2_HOME%\bin

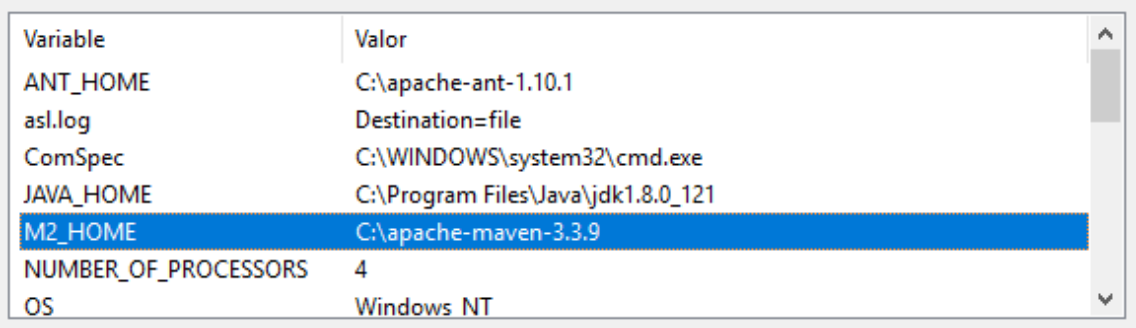


Ilustración 7. Variable de sistema en Windows M2_HOME

En el caso de Maven no necesitamos ninguna biblioteca externa, ya que Maven se encargará de descargar las que necesite cuando ejecutemos el análisis de SonarQube.

Comprobamos que Maven está correctamente instalado pidiendo que nos muestre su ayuda. Para ello ejecutamos el comando **mvn -h** que nos dará la lista de opciones.

5. PRIMEROS ANÁLISIS DE PROYECTOS LOCALES

Ahora que tenemos el entorno preparado y configurado, ya podemos empezar a analizar algún proyecto. SonarQube provee varias vías para poder hacer un análisis. Aunque las ideales son las de integrarlo con Maven o Ant y que haga un análisis automático cuando ejecutemos cierta tarea, como por ejemplo la de compilación, no es la única manera. También se puede hacer un análisis ejecutando nosotros mismos la herramienta de análisis, y así podremos analizar un proyecto concreto cuando queramos, o algún proyecto que no cuente con Maven o Ant.

En este apartado vamos a ver cómo se puede hacer un análisis ejecutando nosotros la herramienta, cómo analizar un proyecto construido con Maven, y cómo analizar un proyecto con Ant.

Existe un repositorio en GitHub con ejemplos de proyectos ya preparados para lanzar el análisis y observar los resultados en SonarQube.

El repositorio es el siguiente: https://github.com/SonarSource/sonar-scanning-examples

Nos descargamos el repositorio en local para poder realizar los primeros análisis, y explicar cómo funciona cada uno.

Análisis con Sonar-Scanner

Con Sonar-Scanner podemos analizar cualquier proyecto sin que tenga preparada ninguna configuración para la gestión del ciclo de vida sobre él (Ant, Maven…) [5].

Para ello, dentro de la raíz del proyecto tendremos que crear un archivo de propiedades que usará sonar-scanner. Este archivo se llama sonar-project.properties

El archivo contendrá las siguientes líneas como mínimo:

```
sonar.projectKey=key.del.proyecto
sonar.projectName=Proyecto_de_prueba
sonar.projectVersion=1.0
sonar.sources=\ruta\relativa\del\codigo
```

En las que definimos la clave de identificación del proyecto, el nombre del proyecto, la versión y la ruta relativa del código fuente.

Cada proyecto deberá tener su archivo sonar-project.properties para poder ser analizado.

Dentro del repositorio que nos hemos descargado hay una carpeta llamada sonarqube-scanner, la cual tiene un proyecto preparado para analizar con esta herramienta.

Vemos que, en la raíz, ya tiene el archivo sonar-project.properties con el siguiente contenido:

```
sonar.projectKey=org.sonarqube:sonarqube-scanner
sonar.projectName=Example of SonarQube Scanner Usage
sonar.projectVersion=1.0

sonar.sources=src,copybooks

sonar.sourceEncoding=UTF-8

## Cobol Specific Properties

# comma-separated paths to directories with copybooks
sonar.cobol.copy.directories=copybooks
# comma-separated list of suffixes
sonar.cobol.file.suffixes=cbl,cpy
sonar.cobol.copy.suffixes=cpy



## Flex Specific Properties

# retrieve code coverage data from the Cobertura report
sonar.flex.cobertura.reportPath=coverage-report/coverage-
cobertua-flex.xml



# PL/I Specific Properties
sonar.pli.marginLeft=2
sonar.pli.marginRight=0
```

Vemos que las propiedades que anteriormente mencionábamos como obligatorias las incluye. El key, el nombre del proyecto, la versión, y la ruta de los fuentes. En este último vemos que contiene dos rutas separadas por una coma: la carpeta src y la carpeta copybooks.

Además han incluido una propiedad para indicar que la codificación de los ficheros es UTF8, además de otras específicas para análisis de ciertos lenguajes como Cobol, Flex y PL/I

Esto se debe a que este proyecto de ejemplo tiene muchos archivos en diferentes lenguajes. Si exploramos la carpeta src, vemos que hay subdirectorios con ejemplos de: css, java, javascript, php, sql, swift, xml… entre otros muchos; por ello no es solo un proyecto Java.

Solo nos queda lanzar el análisis sobre estos ejemplos. Tendremos que abrir una consola y navegar hasta el directorio sonarqube-scanner donde está el archivo sonar-project.properties.

Ejecutando el comando **sonar-scanner** ya empieza el análisis.

Una vez que nos salga lo siguiente en la consola sabremos que ha acabado.

```
INFO: More about the report processing at http://localhost:9000/api/ce/task?id=AVsQ0N1r4-SDxdxtWC60
INFO: Task total time: 12.360 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 13.692s
INFO: Final Memory: 54M/387M
INFO: ------------------------------------------------------------------------

C:\sonar-scanning-examples-master\sonarqube-scanner>
```

Ilustración 8. Consola de análisis completado con Sonar-Scanner

Si entramos ahora en la web de SonarQube (http://localhost:9000/) vemos que en la parte derecha ya nos sale un proyecto analizado:

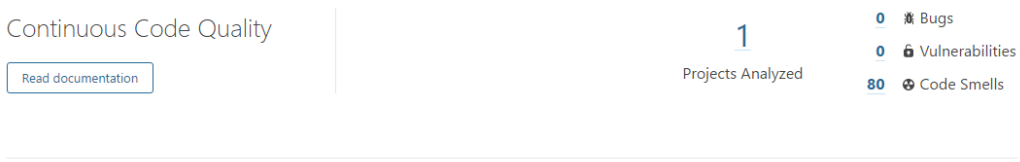


Ilustración 9. Web, pantalla de inicio de SonarQube

Si pinchamos en el 1, nos lleva al listado de proyectos analizados y vemos el siguiente:

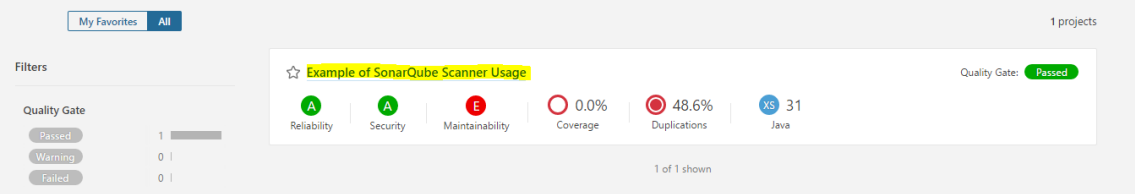


Ilustración 10. Web, listado de proyectos analizados

Vemos que el nombre del proyecto es el que indicamos en el archivo sonar-project.properties
Si pinchamos en el título del proyecto, podemos ver más detalles del mismo, junto con un resumen más amplio del resultado del análisis.

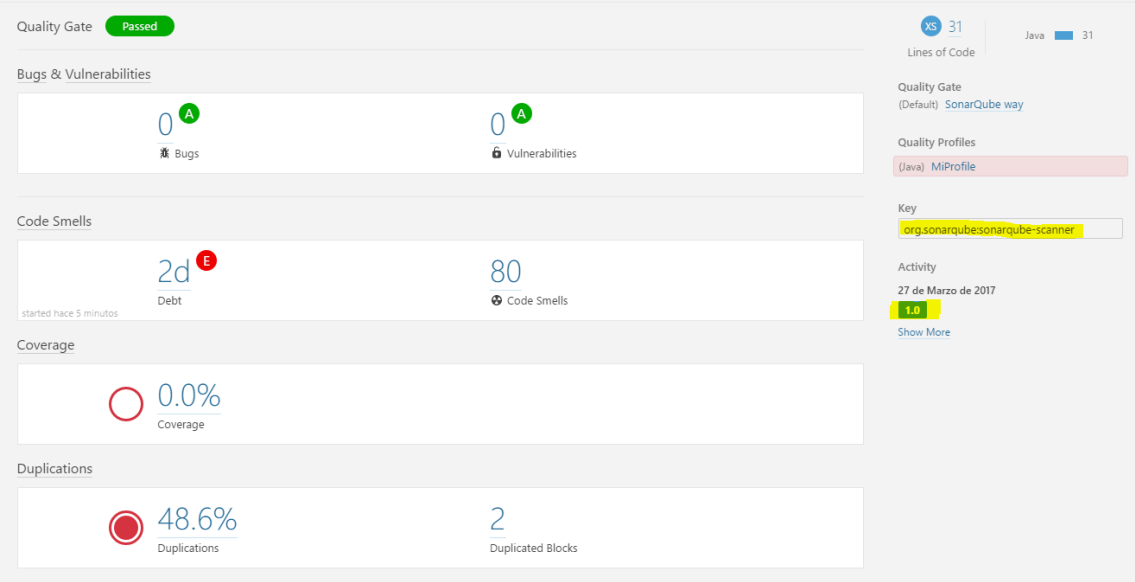


Ilustración 11. Web, resumen del análisis de un proyecto

En la parte derecha vemos que también viene el Key del proyecto y la versión de éste, que son los que indicamos en el archivo de propiedades.

Más adelante veremos cómo interpretar estos resultados obtenidos. De momento nos quedamos en cómo se lanza un análisis con Sonar-Scanner.

Análisis con Ant

Como ya dejamos Ant preparado para hacer análisis con SonarQube, solo nos queda ver cómo se modifica el build.xml del proyecto donde se definen las propiedades y target del mismo para que se ejecuten con Ant. El objetivo es crear un target que realice el análisis, dependiendo de otros targets que pudieran ser necesarios, como el de compilar [6].

Debemos añadir un nuevo espacio de nombres junto con los que pudiera haber en el archivo anteriormente en la cabecera del proyecto. El espacio de nombres nuevo es el siguiente:

```
xmlns:sonar="antlib:org.sonar.ant"
```

Y un ejemplo de cabecera de proyecto sería así:

```
<project name="practica4" default="compilar" basedir="."
xmlns:jacoco="antlib:org.jacoco.ant" xmlns:sonar="antlib:org.sonar.ant">
```

Ahora, en los properties de Ant, tenemos que definir los que usará SonarQube. Estas propiedades vienen a sustituir a los definidas en el archivo sonar-project.properties del apartado anterior.

Los properties obligatorios para que empiece a funcionar serían los siguientes:

```
<property name="sonar.projectKey" value="org.MiProyecto" />
<property name="sonar.projectName" value="Proyecto Java de ejemplo analizado con la tarea para
Ant de SonarQube" />
<property name="sonar.projectVersion" value="1.0" />
<property name="sonar.sources" value="${src.dir}" />
```

Vemos que son los mismos que usábamos en el apartado anterior.

Una vez que tenemos el espacio de nombres y los properties necesarios, tenemos que definir la tarea 'sonar', que sería parecida a ésta:

```
<target name="sonar" depends="compilar">
            <taskdef uri="antlib:org.sonar.ant" resource="org/sonar/ant/antlib.xml">

            </taskdef>

            <!-- Execute the SonarQube analysis -->
            <sonar:sonar />
</target>
```

Como ya añadimos la biblioteca que necesita Ant para poder conectar con SonarQube en la carpeta .ant\lib, no tendremos que especificar en el *taskdef* donde se encuentra. En caso contrario, deberíamos añadir dentro del *taskdef* el elemento *classpath* que defina dónde se encuentra la biblioteca, con una línea como la siguiente:

```
<classpath path="${lib}/sonarqube-ant-task-2.5.jar" />
```

En el repositorio de ejemplos, encontramos una carpeta llamada sonarqube-scanner-ant que ya tiene un archivo build.xml con los datos del proyecto, listo para usar. El build.xml es el siguiente:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="Simple Project analyzed with the SonarQube Scanner for Ant"
default="all" basedir="." xmlns:sonar="antlib:org.sonar.ant">

	<!-- ======== Define the main properties of this project ======== -->
	<property name="src.dir" value="src" />
	<property name="build.dir" value="target" />
	<property name="classes.dir" value="${build.dir}/classes" />

	<!-- Define the SonarQube global properties (the most usual way is to
pass these properties via the command line) -->
	<property name="sonar.host.url" value="http://localhost:9000" />

	<!-- Define the Sonar properties -->
	<property name="sonar.projectKey" value="org.sonarqube:sonarqube-
scanner-ant" />
	<property name="sonar.projectName" value="Example of SonarQube Scanner
for Ant Usage" />
	<property name="sonar.projectVersion" value="1.0" />
	<property name="sonar.language" value="java" />
	<property name="sonar.sources" value="src" />
	<property name="sonar.binaries" value="target" />
	<property name="sonar.sourceEncoding" value="UTF-8" />

	<!-- ======= Define "regular" targets: clean, compile, ... ======= -->
	<target name="clean">
		<delete dir="${build.dir}" />
	</target>

	<target name="init">
		<mkdir dir="${build.dir}" />
		<mkdir dir="${classes.dir}" />
	</target>

	<target name="compile" depends="init">
		<javac srcdir="${src.dir}" destdir="${classes.dir}" fork="true"
debug="true" includeAntRuntime="false" />
	</target>

	<!-- ========= Define SonarQube Scanner for Ant Target ========= -->
	<target name="sonar" depends="compile">
		<taskdef uri="antlib:org.sonar.ant"
resource="org/sonar/ant/antlib.xml">
			<!-- Update the following line, or put the "sonar-ant-
task-*.jar" file in your "$HOME/.ant/lib" folder -->
			<classpath path="path/to/sonar/ant/task/lib/sonarqube-ant-
task-*.jar" />
		</taskdef>

		<!-- Execute SonarQube Scanner for Ant Analysis -->
		<sonar:sonar />
	</target>

	<!-- ========= The main target "all" ========= -->
	<target name="all" depends="clean,compile,sonar" />

</project>
```

Vemos que, además de las propiedades necesarias, han añadido otras como:

- sonar.language para indicar que lenguaje hay que analizar.
- sonar.binaries para indicar dónde se guardan los .class de Java.

Dentro del *taskdef* del target sonar, han dejado el elemento *classpath* para que pongamos la ruta de nuestro Jar pero, como ya he mencionado, no nos hace falta, por lo que podemos borrar esa línea.

Para ejecutar el análisis solo tendremos que navegar con la consola hasta la ruta raíz de este proyecto, donde se encuentra el `build.xml`. En este caso el proyecto es únicamente en Java, y solo hay una clase java dentro del directorio `src`.

Si queremos lanzar el análisis, en este caso tenemos varias maneras.

- Ejecutar la tarea sonar con el comando: **ant sonar**
- Ejecutar la tarea all con el comando **ant all**.

La idea de esta última tarea es, una vez que hayamos hecho los cambios que sean sobre el código, y para saber qué cosas tenemos que corregir según las normas establecidas en SonarQube, es hacer un clean del proyecto, para eliminar los binarios que ya existan. Volver a compilar el proyecto para tener la última versión de los binarios, y después hacer el análisis con SonarQube con los datos más actuales.

Una vez hayamos ejecutado la tarea que consideremos oportuna, si vamos al dashboard de SonarQube vemos que ya tenemos dos proyectos analizados en nuestra lista de proyectos.

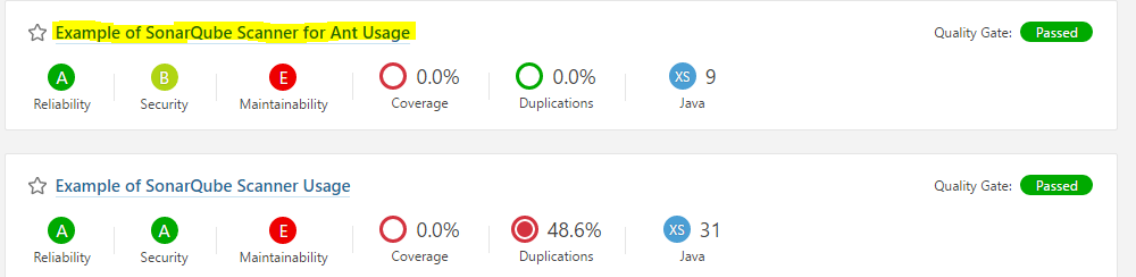


Ilustración 12. Web, listado de proyectos analizados 2

Análisis con Maven

Para analizar un proyecto con Maven es muy parecido al punto anterior. En este caso no tenemos nada que añadir en el pom.xml ya que, con los datos que tiene habitualmente este archivo, es suficiente para realizar el análisis [7].

El pom.xml es un archivo que modela un proyecto, el cual incluye el nombre del proyecto, el "Key", la versión, dependencias con bibliotecas externas entre otras muchas cosas. Por eso no es necesario añadir ningún parámetro de SonarQube, ya que se usan los existentes.

Como los proyectos Maven siguen una estructura conocida como Archetype, el propio Maven ya sabe dónde se encuentra el código fuente del proyecto, los test, los binarios… por lo que tampoco tenemos que indicar dónde está el código que se quiere analizar, pues Maven ya lo sabe.

Sí que puede ser necesario que indiquemos algún parámetro para añadir información que Maven no tiene, como el servidor donde tenemos SonarQube para que guarde ahí los resultados del análisis, cuando no sea en local.

En el repositorio de ejemplos encontramos una carpeta llamada sonarqube-scanner-maven con el pom.xml.

Debemos navegar con la consola hasta este directorio para realizar el análisis.

El objetivo que debemos ejecutar para el análisis es *sonar:sonar* por lo que el comando sería: **mvn sonar:sonar**

Al igual que en Ant, se puede combinar con diferentes objetivos según lo que busquemos.

En este caso el autor de los ejemplos nos recomienda usar el comando

mvn clean install sonar:sonar

Esto hará una limpieza del proyecto, eliminando los datos de compilaciones anteriores, recompilará los archivos, lo empaquetará como esté configurado, pasará los test si existieran y, finalmente, realizará el análisis de SonarQube, el cual dispondrá además del código fuente para analizar, los binarios para poder analizarlos también, el resultado de los test y la cobertura para añadirlo al informe de SonarQube.

Ahora, en el listado de proyectos analizados en SonarQube, nos aparece este último.

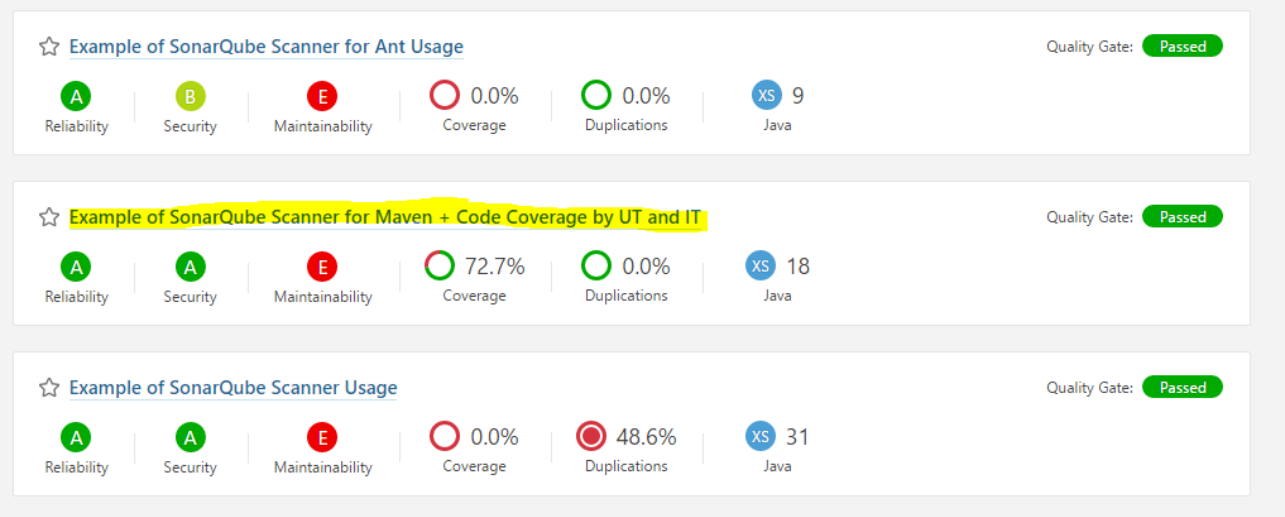


Ilustración 13. Web, listado de proyectos analizados 3

6. PLUGINS Y PERFILES

Plugins

Ahora que ya sabemos cómo lanzar análisis de proyectos a través de varias herramientas, vamos a ver cómo gestionar diferentes plugins que permiten analizar multitud de lenguajes, o añadir más reglas de un lenguaje específico. Concretamente veremos el caso de Java.

Lo primero que haremos es desinstalar todos los plugins que vienen por defecto para ver cómo analiza SonarQube. Para ello, cuando tengamos el programa arrancado, vamos al dashboard, y en la esquina superior derecha vemos la opción de *Log in*. Pulsamos y nos identificamos con el usuario **admin** y la contraseña **admin**. Son los que vienen configurados por defecto. Deberemos estar siempre logueados, ya que muchas de las cosas que vamos a hacer a continuación solo se permiten a usuarios con ciertos permisos. El admin por defecto puede hacer todo.

Una vez identificados, en la barra superior nos dirigimos a *Administration*. En este apartado se podrán configurar numerosos parámetros de SonarQube. El que nos interesa ahora es, en el submenú *System,* el apartado *Update center*

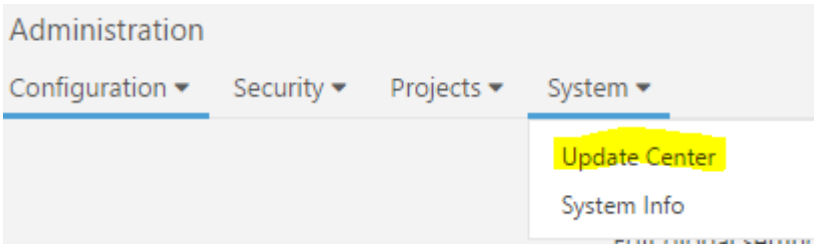


Ilustración 14. Web, submenú de administración

Lo primero que vemos por defecto es el listado de plugins instalados. Vamos a ir pulsando en *Unistall* en cada uno de los plugins en la parte derecha de la pantalla.

Finalmente, en la parte superior ha aparecido un recuadro amarillo con las opciones *Restart* y *Revert***.** Pulsamos en Restart para que se guarden los cambios.

Una vez que se vuelva a iniciar SonarQube, si volvemos a este apartado, el listado de plugins instalados estará vacío.

Antes de lanzar un nuevo análisis, para no liarnos con los resultados de anteriores análisis, ya que vamos a seguir usando los mismos ejemplos, eliminaremos todos los proyectos analizados.

En el submenú *Projects* entramos en la opción *Management*. Aquí nos salen todos los proyectos analizados. Marcamos el check en todos y pulsamos en *Delete* en la derecha de la pantalla. Con esto ya tenemos todo listo.

Ahora lo que nos queda es volver a analizar un proyecto. De los ejemplos que usamos anteriormente, vamos a continuar con el que se analiza con la herramienta Sonar-Scanner, ya que tiene ficheros en multitud de lenguajes. Mediante la consola nos dirigimos al directorio de este proyecto y lanzamos el análisis con Sonar-Scanner.

Observamos que sale el siguiente error:

```
INFO: ------------------------------------------------------------------------
INFO: EXECUTION FAILURE
INFO: ------------------------------------------------------------------------
INFO: Total time: 1.485s
INFO: Final Memory: 38M/159M
INFO: ------------------------------------------------------------------------
ERROR: Error during SonarQube Scanner execution
ERROR: No quality profiles have been found, you probably don't have any language plugin installed.
ERROR:
ERROR: Re-run SonarQube Scanner using the -X switch to enable full debug logging.
```

Ilustración 15. Consola error al analizar proyecto con Sonar-Scanner

Como bien indica el error, no tenemos ningún plugin de lenguaje instalado, por lo que no sabe analizar ningún lenguaje.

Para analizar cada lenguaje necesitamos un plugin específico, ya que SonarQube por sí solo no sabe analizarlos.

Volvemos al *update center* para añadir el plugin de Java.

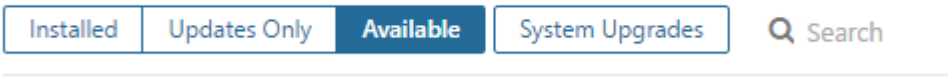


Ilustración 16. Web, cabecera Update Center

Nos colocamos en el apartado de *Available* donde aparecerán todos los plugins disponibles para instalar. Buscamos el general de Java que actualmente se llama SonarJava.

Ilustración 17. Web, información de plugin

Pulsamos *Install* en la parte derecha, y reiniciamos SonarQube de la misma manera que hicimos al desinstalar los plugins.

Una vez vuelva a estar activo volvemos a intentar analizar el proyecto. Ahora todo debe funcionar.

En la barra superior seleccionamos *Projects* y seleccionamos *All* para ver todos. Ahí vuelve a aparecer nuestro proyecto.

Si entramos en él, y en el submenú superior pulsamos en *Code*, podemos ver que solo ha analizado los ficheros de Java, ya que es el único lenguaje para el que tenemos instalado un analizador, aunque en el proyecto existan muchos más ficheros de otros lenguajes.

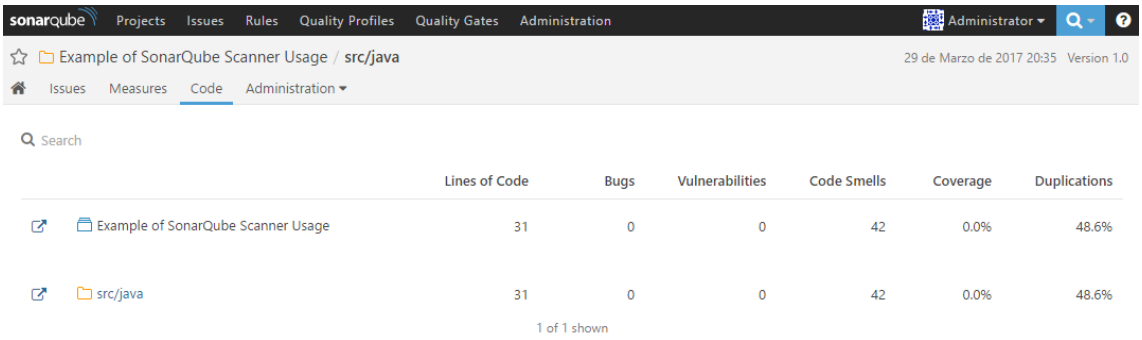


Ilustración 18. Web, código fuente analizado

Podemos probar a añadir plugins de otros lenguajes para ver que, efectivamente, los analiza. En mi caso voy a añadir el de CSS, que son otros ficheros que también están en el proyecto.

Volvemos a analizar el mismo proyecto y vemos que ahora salen también los ficheros CSS.

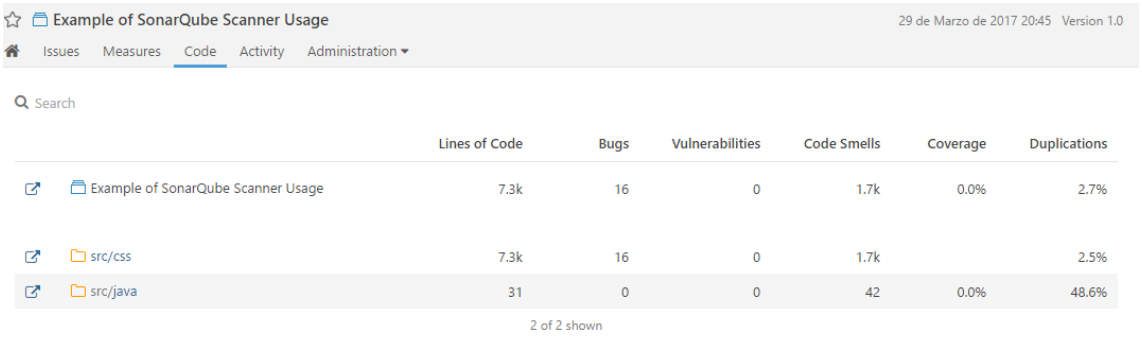


Ilustración 19. Web, código fuente analizado 2

Para Java, además del plugin por defecto, existen otros plugins que añaden más reglas que el de SonarQube no tiene. La mayoría de estos plugins dependen del de SonarQube para poder funcionar y son integraciones con otras herramientas que ya analizaban el código de Java.

Los más interesantes son los siguientes:

- **PMD**: busca potenciales errores que pudieran provocar en algún momento un fallo en tiempo de ejecución. También es capaz de encontrar *'dead code'* (código que nunca se ejecutará), sentencias innecesarias o vacías, código con mal rendimiento… [8], [9]

- **FindBugs**: busca errores pero no sobre el propio código fuente, si no sobre el bytecode resultado de compilar los ficheros java. Por eso es importante indicar a SonarQube dónde se guardan los bytecode de nuestro proyecto, y compilarlo antes de analizarlo, ya que sin estos bytecodes FindBugs no analizará nada [10], [11].

- **Checkstyle**: no encuentra errores en código, sino incumplimiento de los estándares de codificación en Java. A diferencia de otros lenguajes, en Java se podría escribir todo en una línea, lo que haría un código ilegible e inmantenible. Por eso existen unos estándares sobre cómo se debe formatear el código, cómo deben nombrarse las variables, métodos y clases, cómo documentar el código, etc [12], [13].

Todos estos plugins lo que hacen es añadir más y más reglas al programa. Puede que no nos interese aplicar todas al analizar nuestro proyecto, si no un subconjunto de cada plugin.

Por ello SonarQube dispone de lo que se conoce como *perfiles de calidad*, que no es más que una agrupación de ciertas reglas de entre todas las disponibles.

Reglas

Primero vamos a explicar qué es una regla.

Una regla es una la especificación de algo que debe cumplir nuestro código o de una situación que no debe darse en nuestro código.

Vamos a ver las reglas que existen en SonarQube por tener solo instalado el plugin de Java. Para ello, entramos en nuestro SonarQube a través de un navegador y, en la barra superior, pulsamos en *Rules***.** Nos aparece una lista con todas las reglas disponibles

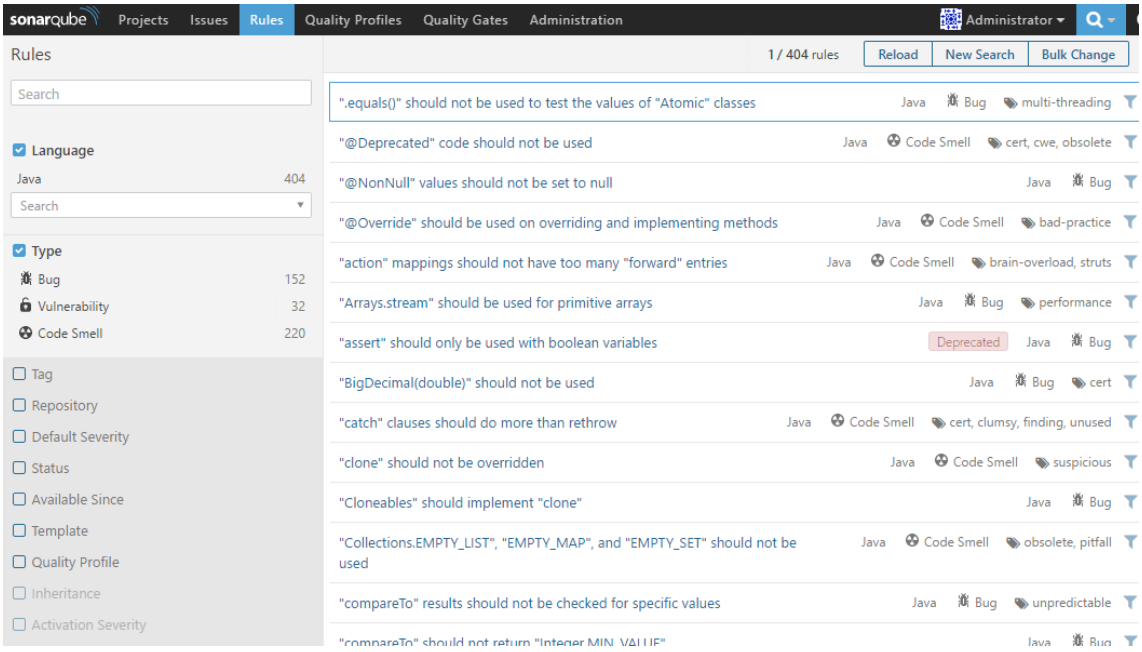


Ilustración 20. Web, buscador de reglas

En la parte izquierda tenemos un panel de filtros para poder buscar más concretamente.

- Language: en este caso solo tenemos Java, pero si tuviéramos algún lenguaje más instalado aparecería en esta lista para poder filtrar las reglas.
- Type: todas las reglas se clasifican en tres tipos:
  - Bug
  - Vulnerability
  - Code Smell
- Tag: cada regla puede tener una o varias etiquetas para clasificarla, como por ejemplo: rendimiento, multi-hilo…

- Repository: las reglas pertenecen a un repositorio. Lo normal sería que todas las reglas de un plugin pertenezcan a un repositorio, pero existen plugins que incluyen varios repositorios.

- Default severity: las reglas tienen una gravedad, según lo importante que sean. En SonarQube tenemos cinco tipos de gravedad que van desde *informativa* a *bloqueante*, pasando por *menor*, *mayor* y *crítica.* A la hora de usar las reglas podemos considerar que no tiene la gravedad que viene por defecto, por lo que podemos cambiarla. Este filtro usará la gravedad que tiene la regla por defecto.

- Status: las reglas también pueden tener un estado. La mayoría estarán en *Ready*, pero puede que existan algunas en estado *Deprecated*, o en estado *Beta*.

- Available since: aquí podemos filtrar desde que fecha está disponible la regla. Esta fecha indica cuándo se añadió en nuestro SonarQube particular.

- Template: permite filtrar unas reglas especiales que sirven como plantillas para generar nuevas reglas.

- Quality profile: como veremos a continuación, existen los llamados perfiles de calidad, que son agrupaciones de estas reglas. En este apartado podemos filtrar las reglas por perfil, para así ver qué reglas se están usando en uno determinado.

- Inheritance: cuando en el filtro “Quality profile” se ha seleccionado un perfil que hereda de otro, este filtro se activa y permite filtrar por las reglas que se heredan, las propias del perfil que no se heredan, o las que se sobrescriben del perfil padre.

- Activation severity: como ya he mencionado, las reglas tienen una gravedad que le ha asignado el desarrollador de la regla. Sin embargo, puede que una regla para su desarrollador no sea tan grave, pero nuestro proyecto sí que lo sea. Por eso, al incluir una regla en un perfil, permite establecerle una gravedad diferente solo en ese perfil.

  De esta manera, cuando estemos filtrando por un perfil de calidad, podemos usar este filtro para quedarnos solo con las de cierta gravedad dentro del perfil, aunque originariamente tuvieran otra gravedad.

Ahora que ya sabemos cómo usar el buscador de reglas, vamos a ver alguna en concreto para observar qué información nos da acerca de ella.

Para eso, en la lista de reglas pulsamos sobre alguna. En mi caso voy a ver una con el título **“BigDecimal(double)” should not be use**

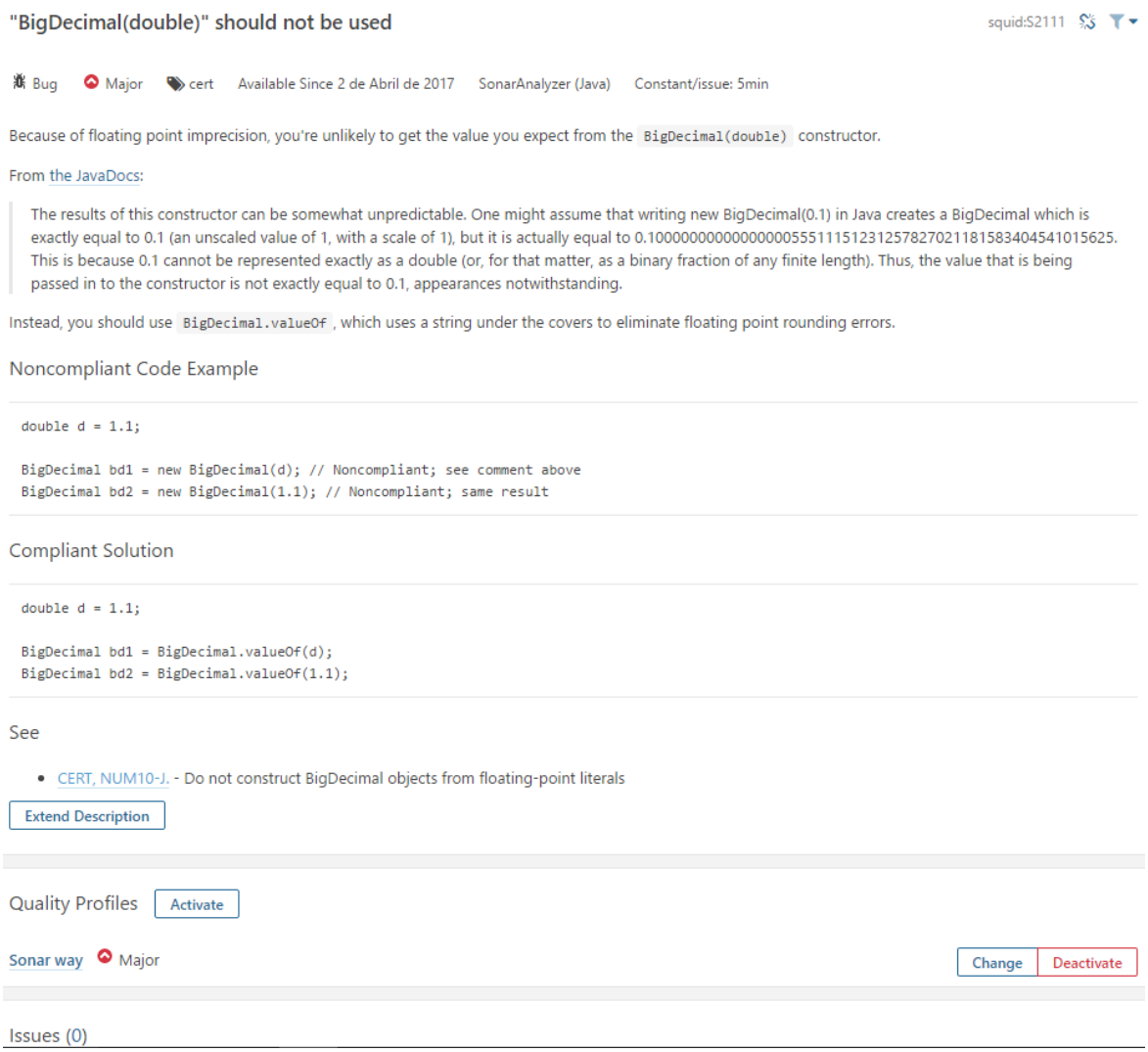


Ilustración 21. Web, ficha con información de una regla

Vemos una cabecera informativa de la regla donde podemos observar su Tipo (Bug), Gravedad (major), Tags (cert), fecha de disponibilidad, repositorio (SonarAnalyze(Java)), tiempo estimado de resolución (5min).

Después viene una explicación de la regla, donde explica por qué no se debe hacer así y qué alternativa tenemos.

A continuación pueden venir unos ejemplos de código, donde en uno se puede ver cómo no se está aplicando esta regla y, por lo tanto, es una muestra de lo que no debe hacer el código. Y otro donde se resuelve el problema del ejemplo anterior y es como se espera que lo hagamos.

Debajo de los ejemplos puede venir una serie de referencias a fuentes externas que hablan del problema de esta regla.

En el apartado de Quality Profiles, aparece un listado de los perfiles de calidad que incluyen esta regla, y la posibilidad para cada uno de estos perfiles de cambiar su gravedad, ya que podemos considerar que es más o menos grave de lo que viene establecido por defecto.

Finalmente hay un apartado de issues. Al realizar un análisis sobre un proyecto, cada vez que se incumple una de las reglas se crea un issue para solucionarlo. Entonces, en este apartado, aparece

el contador de cuántas veces se ha incumplido la regla entre todos los proyectos, y debajo una lista de los proyectos que más veces han incumplido la norma y un contador para cada proyecto

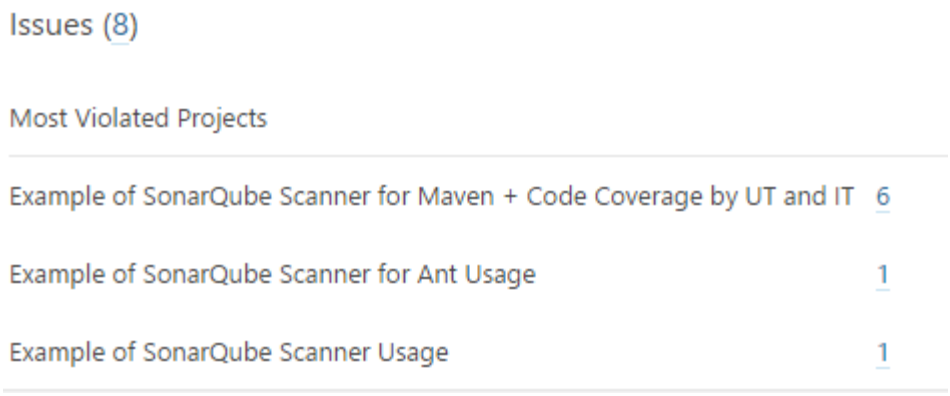


Ilustración 22. Web, contadores de aparición del issue en cada proyecto analizado

Perfiles

Ahora que ya sabemos lo que es una regla, podemos ver qué son los perfiles de calidad. En la barra superior pulsamos en *Quality Profiles*.

Veremos un listado de los perfiles existentes agrupados por lenguajes. Actualmente solo nos aparece un perfil de Java, que es el que se crea por defecto cuando instalamos el plugin correspondiente.

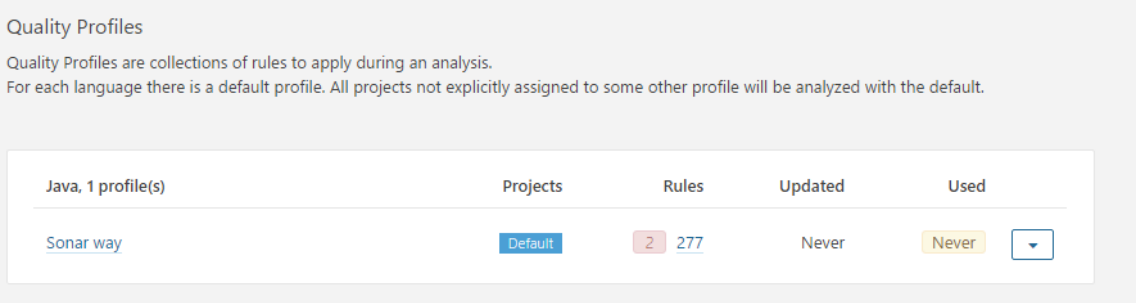


Ilustración 23. Web, listado de perfiles de calidad

Este perfil está marcado como *default* por lo que, al lanzar un análisis, si no se especifica ningún perfil específico, usará éste cuando tenga que analizar un fichero de Java, comprobando así que cumple todas las reglas añadidas al perfil.

En la columna de *Rules* nos indica el número de reglas incluidas en el perfil, y en el recuadro rojo nos indica cuántas de ellas están en estado *deprecated*. También nos informa de la última vez que se modificó el perfil, y la última vez que se usó en un análisis.

En el desplegable de la derecha tenemos varias acciones:

- Añadir más reglas
- Hacer una copia de seguridad del perfil
- Compararlo con otro perfil, lo que nos indica qué reglas se encuentran en uno y no en otro

- Hacer una copia del perfil
- Cambiar el nombre
- Marcar el perfil como *default*
- Borrarlo

No siempre aparecen todas las opciones, solo las que podemos realizar.

Si pinchamos en el nombre del perfil, podemos ver más detalles del mismo.

En la parte izquierda vemos un resumen de reglas activas (añadidas al perfil) e inactivas (están disponibles para añadir) clasificado por tipos. También hay un botón para añadir más reglas de la lista de disponibles.

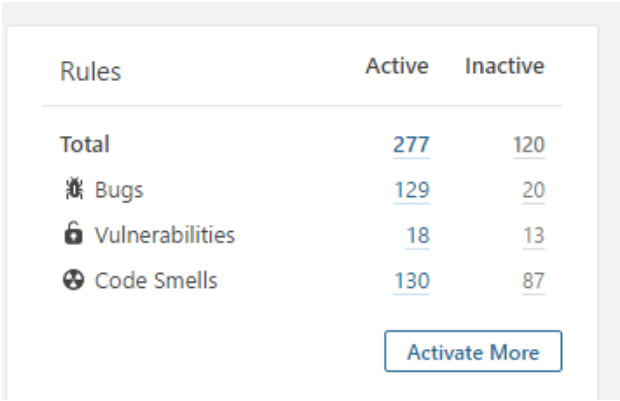


Ilustración 24. Web, información de reglas incluidas y no en el perfil

En la parte central hay un apartado el apartado de Herencia del cual hablaremos más adelante.

En el apartado Projects podemos ver un listado de los proyectos que usan este perfil por defecto. En este caso, como es el perfil *default*, ya se usa para todos los proyectos que no especifiquen el perfil con el que se tiene que analizar.

Para poder ver esto, vamos a crear un nuevo perfil de calidad y se lo asignaremos a alguno de los proyectos que ya hemos analizado. Si no tenemos ninguno, podremos volver a analizar los tres de ejemplos anteriores para trabajar con ellos.

Volvemos al apartado *Quality Profiles*, y en la esquina superior derecha pulsamos en *Create*.

En la ventana que nos aparece indicamos un nombre y un lenguaje para nuestro nuevo perfil.

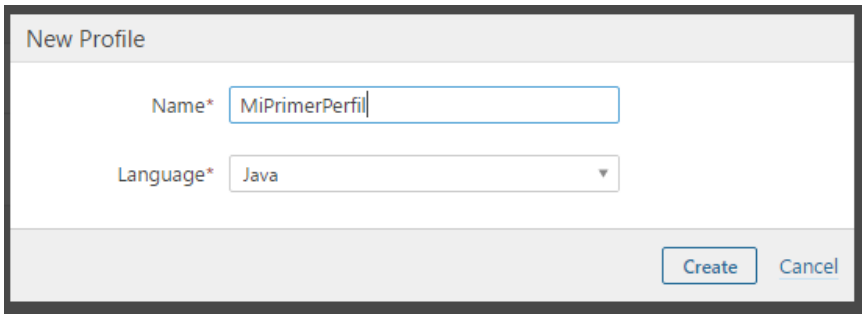


Ilustración 25. Web, formulario para la creación de un nuevo perfil de calidad

Una vez creado, nos muestra directamente el detalle del perfil. En la parte izquierda vemos que no tiene ninguna regla añadida, y el resto de reglas existentes de Java están disponibles para agregarlas. Pulsamos en *Activate More* y añadimos algunas de las reglas.

En la parte derecha del listado pulsamos en *Activate*, elegimos el perfil, la gravedad que consideremos nosotros y listo.

Si volvemos al apartado de *Quality Profiles* vemos que nuestro nuevo perfil ahora tiene 4 reglas, pero ningún proyecto.

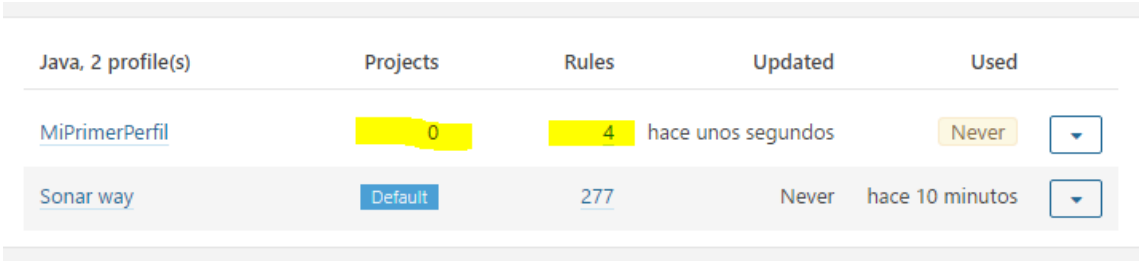


Ilustración 26. Web, listado de perfiles de calidad 2

Volvemos a entrar en el detalle de nuestra regla, y en el apartado de *Projects* vemos que ahora hay un botón llamado Change Projects. Si pulsamos en él nos sale una nueva ventana con 3 filtros:

- Proyectos incluidos
- Proyectos no incluidos
- Todos los proyetos.

Moviéndonos al filtro All, nos deberían salir los tres proyectos que ya analizamos en un apartado anterior. Voy a seleccionar dos de ellos para probar.

Ahora en el apartado Projects nos salen estos dos proyectos:

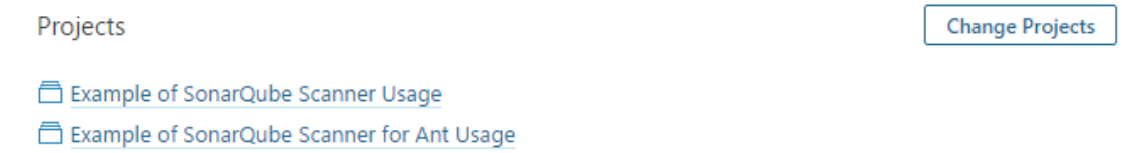


Ilustración 27. Web, lista de proyectos que usan el perfil

Esto significa que cuando volvamos a analizar esos proyectos usará este perfil de calidad en vez del marcado como *default*. Como solo hemos añadido cuatro reglas al perfil, solo comprobará que en esos proyectos se cumplen estas cuatro reglas, el resto las ignora.

Vamos a hacer una prueba para ver que, efectivamente, funciona. En el apartado de *Quality Profiles*, cada perfil tiene una columna que muestra la ultima vez que se usó. Voy a analizar uno de los proyectos incluido en nuestro nuevo perfil, y veremos que, efectivamente, se acaba de usar, mientras que el perfil *default* no se ha usado recientemente.

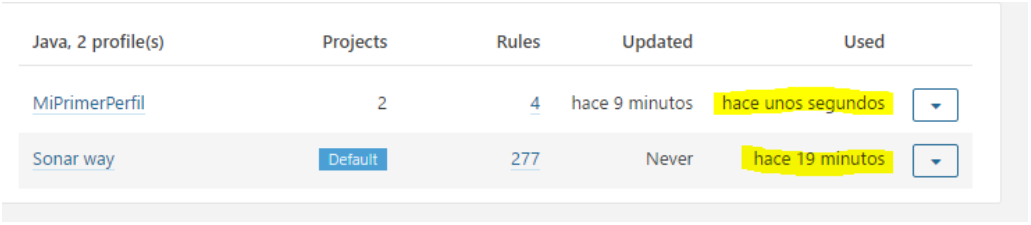


Ilustración 28. Web, listado de perfiles de calidad 3

Una de las opciones disponibles en el menú de acciones de cada perfil, es hacer un backup que más tarde podremos restaurar en este u otro servidor de SonarQube.

Abrimos el desplegable de acciones de nuestro nuevo perfil y pulsamos en backup.

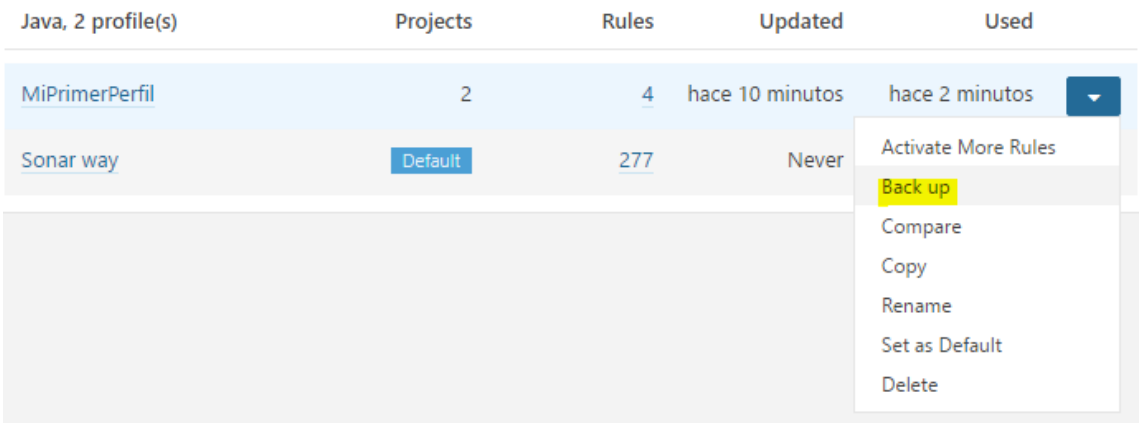


Ilustración 29. Web, desplegable de acciones sobre un perfil

Se descargará un fichero XML que es el backup que debemos guardar.

Ahora vamos a borrar nuestro perfil con la acción Delete del mismo menú.

Para recuperar el perfil a través del backup, vamos a ir al botón de Create, donde anteriormente creamos un nuevo perfil, pero esta vez vamos a abrir el desplegable que hay a su derecha. Nos aparece la opción *Restore Profile*. En la ventana que nos sale, seleccionaremos el XML que nos descargamos anteriormente y aceptamos.

Vemos que nuestro nuevo perfil ha vuelto a aparecer, pero lo único que recupera es el nombre del perfil y las reglas que tenía añadidas. Los proyectos que le asignamos no los recupera.

Ya solo nos queda ver el apartado de herencia que se verá fácilmente con el siguiente ejemplo.

A nuestro nuevo perfil le hemos incluido cuatro reglas (tres de ellas ya están en el perfil Sonar way), y el perfil de Sonar way tiene 277 reglas.

Podemos hacer que nuestro perfil herede de Sonar way, por lo que aparte de las cuatro reglas que ya teníamos, se le agregan por herencia las 277 de Sonar way. Como tres de nuestras cuatro reglas ya estaban en Sonar way, éstas no se agregan. Entonces en total tenemos 278 reglas en nuestro nuevo perfil, las 277 de Sonar way, más la que agregamos a nuestro perfil que no estaba en el otro.

Para crear esta herencia abrimos el detalle de nuestro nuevo perfil, y en el apartado de *Inheritance* pulsamos en el botón *Change Parent.*

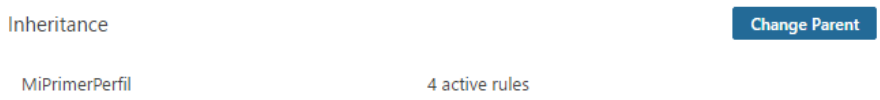


Ilustración 30. Web, cambio de herencia en perfiles

En la nueva ventana elegimos el perfil Sonar way y pulsamos en Change.

Vemos que, efectivamente, Sonar way tiene 277 reglas y nuestro nuevo perfil tiene 278.

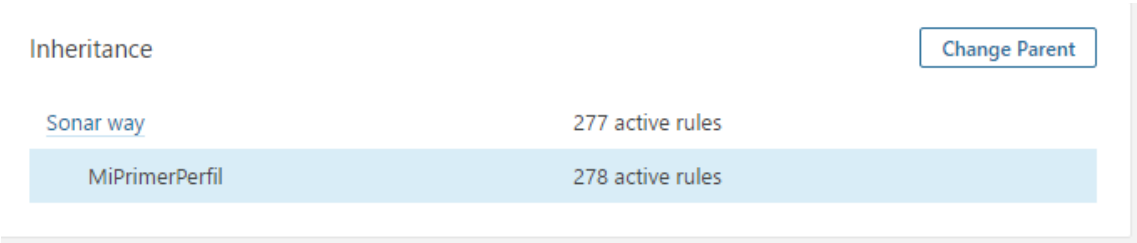


Ilustración 31. Web, visualización de herencia entre perfiles

La ventaja de esta herencia es que, si añadimos una nueva regla a Sonar way, automáticamente se agrega a nuestro nuevo perfil. Podemos probarlo añadiendo una regla a Sonar way que no tenga. Vemos que ahora Sonar way tiene 278 reglas y nuestro perfil 279.

Al igual que añadimos un nuevo padre a nuestro perfil, también podemos quitárselo de la misma manera, seleccionando como padre *None.*

Hay que tener cuidado con esto, ya que no vuelve a dejar nuestro perfil como antes. Lo que hace es quitar de nuestro nuevo perfil todas las reglas de Sonar way, por lo que nos quedaremos solo con 1 regla, en lugar de las 4 con las que contábamos al principio.

Ahora que ya dominamos los perfiles y las reglas, vamos a incluir nuevas reglas de Java. Para ello instalaremos los cuatro plugins para analizar Java: PMD, FindBugs, CheckStyle y CodeSmells.

Primero recordemos que, actualmente, tenemos 397 reglas disponibles para añadir a los perfiles. Esto podemos verlo en el buscador de Reglas:

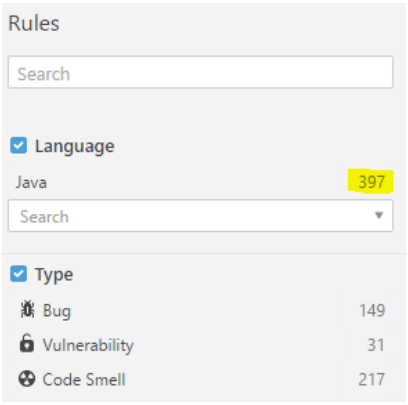


Ilustración 32. Web, detalle del buscador de reglas

Instalamos los plugins como se explicó anteriormente y vemos que ahora contamos con 1656 reglas de Java y 12 para JSP.

El funcionamiento es el mismo, estas reglas se pueden agregar a un perfil para que este las use en sus análisis.

En el buscador, el filtro Repository ahora tiene varios. Hay plugins como SonarJava que incluye todas sus reglas en un solo repositorio llamado SonarAnalyzer, pero otros como FindBugs separa sus reglas en tres repositorios: FindBugs, FindBugs Contrib y Find Security Bugs.

Ilustración 33. Web, detalle del buscador de reglas por repositorio

Si nos dirigimos a la parte de perfiles vemos que se han creado cinco nuevos, cuatro en Java y otro en JSP. Se puede intuir por el nombre que todos van relacionado con FindBugs.

Vamos a usar el nuevo perfil que creamos anteriormente, al cual le vamos a añadir todas las reglas disponibles para hacer análisis con todo.

Si no queremos ir añadiendo una a una las más de 1600 reglas, podemos hacerlo con todas de golpe. Dentro del detalle del perfil, pulsando en *Activate More* nos lleva al listado de reglas.

En la parte superior hay un botón llamado *Bulk Change* que sirve para hacer cambios masivos. De esta manera para agregar todas las reglas no tendremos que ir una por una pulsando en *activate.*

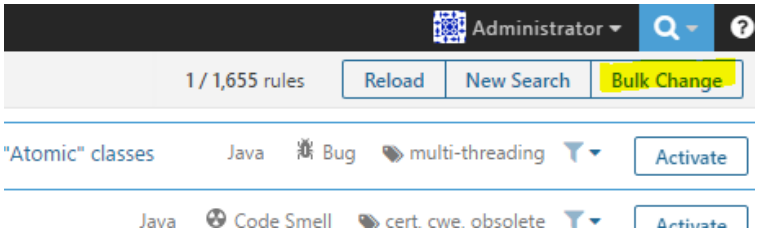


Ilustración 34. Web, botón Bulk Change

Lo pulsamos y elegimos la opción *Activate In MiPrimerPerfil*, aceptamos y ya tenemos todas añadidas. Si usamos los filtros de búsqueda, solo se añadiran las reglas que cumplan los requisitos de búsqueda.

Vamos a asignar a nuestro perfil el proyecto que se analiza mediante Ant, ya que es bastante simple y apenas debería tener ningún error. Es más, de un primer vistazo parece todo correcto en cuanto a calidad de código. La único problema a priori que se ve es que hay un método que se llama foo() en vez de getFoo().

La única clase que tiene para analizar es la siguiente:

```
public class One {
  String message = "foo";

  public String foo() {
    return message;
  }

  public void uncoveredMethod() {
    System.out.println(foo());
  }
}
```

Una vez tenemos el proyecto vinculado a nuestro perfil de calidad volvemos a analizar el proyecto.

Un resumen rápido del resultado obtenido, ya que leer los informes resultantes se explica en el siguiente apartado. Si vamos al proyecto analizado (Apartado Projects de la barra superior, y pulsar en el proyecto de Ant) vemos en el siguiente resumen que 9 simples líneas de código tienen 36 reglas incumplidas (35+1). ¡Sale una media de 4 errores por línea!

Si intentásemos ver los errores sobre el código es ilegible, ya que hay más texto de errores que de código y al final no sabemos en qué línea está cada error.

Con esto se pueden deducir algunas conclusiones. Aplicar todas las reglas disponibles en un análisis no es muy viable, ya que con tantos errores en tan poco código acabaremos ignorándolos.

Lo ideal es crear un perfil con las reglas que realmente son interesantes. Pero esto tiene un inconveniente: tendremos que leernos y entender todas las reglas para decidir cuáles aplicamos y cuáles no.

Si es un proyecto nuevo, estaría bien empezar con un perfil que ya tenga todas las reglas que de verdad queremos aplicar, y así tener un código lo más limpio posible desde el primer día.

Si por el contrario ya tenemos un proyecto en curso, y este es muy grande, aplicar todas las reglas que nos interesan de golpe no es la mejor opción, ya que posiblemente saldrán tantos incumplimientos que acabaremos abandonando SonarQube. Lo ideal es irlas aplicando progresivamente. Se introduce un pequeño conjunto de reglas y se arregla el código que no las cumple. Después añadimos otro grupo y volvemos a arreglar los errores. Así iterativamente, hasta que tengamos el perfil de calidad con todas las reglas que deseamos, y el proyecto cumple todas.

Probar plugin para GitHub y Bitbucket

He hablado en varias partes del TFG sobre los plugin para poder hacer análisis de proyectos almacenados en GitHub y Bitbucket. Todo lo que comentaba anteriormente es erróneo, y lo descubrí al llegar a esta parte, que es cuando me puse a investigar cómo funcionaban dichos plugins y como podía analizar mis proyectos almacenados en los servicios mencionados.

Estos plugins son para analizar pull-request hechos en GitHub o Bitbucket. Son necesarios para poder conectarse a estas plataformas, obtener los datos del pull-request que se le ha indicado, y dejar los resultados en un comentario del pull-request.

La idea es simple, pero es bastante complejo montar y configurar toda la infraestructura necesaria. Nunca conseguí que funcionara todo de manera automática. Aunque sí que lo conseguí ejecutando ciertas tareas de forma manual.

El escenario ideal sería tener un servidor con SonarQube montado, otro servidor con algún software de integración continua funcionando, como Jenkins o Travis, y un proyecto subido a GitHub o Bitbucket. La colaboración a este proyecto sería mediante pull-request. Un usuario se hace un fork del repositorio en su cuenta privada. Realiza los cambios necesarios para desarrollar una nueva funcionalidad o corregir errores. Una vez esté todo listo, para que el dueño del repositorio haga merge de esos cambios en el repositorio original del proyecto, el desarrollador tiene que crear un pull-request con los nuevos cambios que ha realizado. Esto se hace pulsando un botón [14]. Entonces el dueño del repositorio original recibe una notificación de que dispone de un nuevo pull-request en el proyecto, y tendrá que revisar los cambios añadidos valorando si los acepta y se mergea con el proyecto, o los rechaza descartando los cambios.

Es en este punto donde entra en funcionamiento todo el sistema. El repositorio original al recibir el pull-request, lanza una tarea en el servidor de integración continua. Obviamente tiene que estar previamente configurada esta conexión. Esta tarea puede realizar varias comprobaciones. Por ejemplo puede comprobar que si se añadieran los cambios propuestos en el pull-request, el proyecto sigue compilando sin errores y los test pasan sin fallos. Si esta comprobación es correcta se pasa a la siguiente que consiste en analizar los nuevos cambios añadidos mediante SonarQube, comprobando así, que cumple con la calidad que exigimos en nuestro proyecto.

Los resultados del análisis no se añaden al dashboard de SonarQube si no que se añaden en un comentario del pull-request, de tal manera que cuando el administrador del repositorio vaya a revisar si acepta o no el pull-request dispondrá de más información para valorarlo.

La dificultar es conectarlo todo para que funcione de manera automática. Estos servicios tienen una alta seguridad, por lo que no basta con poner un usuario y contraseña. Hay que crear accesos específicos mediante webhooks, tokens de seguridad y demás. Además hay que saber usar un sistema de integración continua, lo que puede llevar bastantes horas que no tenía planificadas en este TFG.

7. CASO PRÁCTICO: JHOTDRAW

Una vez que hemos aprendido cómo instalar y configurar SonarQube, cómo realizar análisis de proyectos, y cómo configurar los diferentes perfiles y reglas que queremos aplicar, vamos a ver un caso práctico más real, en el que veremos la información que obtenemos de los análisis realizados, y aprenderemos a interpretar dicha información, para ver qué partes deberemos mejorar en nuestro proyecto.

Para ello usaremos unos proyectos Open Source, en los que realizaremos los análisis y leeremos los resultados. Para estos análisis vamos a usar el perfil por defecto de SonarQube para Java, sin usar ningún plugin extra de los que hemos visto. Esto es porque si usamos las más de 1600 reglas que hay para Java, obtendremos tantos errores que no nos ayudarán demasiado.

Este primer proyecto será JHotDraw. JHotDraw es un framework Java para representar gráficos técnicos y estructurados en un interfaz gráfico. Sus desarrolladores aseguran que en él se utilizan algunos patrones de diseño bien conocidos [15].

Lo primero que haremos es descargar su código fuente y ver cómo está estructurado, para saber cómo podremos realizar el primer análisis. La versión que me he descargado es la 7.6 que por versionado debería ser la última, aunque en la web de descarga afirman que la última es la 7.0.6.

He elegido esta versión por que viene configurada con Maven, lo que nos puede facilitar las cosas. Se puede descargar desde aquí:
https://sourceforge.net/projects/jhotdraw/files/JHotDraw/JHotDraw%207.6/

Vemos que dentro del ZIP descargado contiene otro ZIP. Este último contendrá la carpeta del proyecto que consiste en cuatro subcarpetas:

- **Documentation** contiene documentación sobre lo que ofrece, temas de licencias, changelogs, etc.
- **JavaDoc** contiene la documentación técnica de cómo podemos usarlo dentro de nuestro proyecto.
- **Samples** contiene ejemplos de lo que se puede llegar a hacer.
- **Source** es donde está el código y es la que nos interesa.

Si navegamos dentro de la carpeta *Source* llegamos a la raíz del proyecto.

En esta raíz se encuentra el archivo `pom.xml` que nos interesa, junto con numerosos archivos build de Ant.

Si intentamos compilar y empaquetar el proyecto con el comando **`mvn clean install`** fallará, ya que no encuentra una de las dependencias.

He visto que en el repositorio de Maven ya no está disponible y existen versiones más nuevas. Si intentamos usar una de estas nuevas versiones también fallará la compilación porque intenta usar clases que esta nueva versión de la biblioteca no tiene.

JHotDraw parece un proyecto ya abandonado, porque los últimos cambios son de 2011.

Lo que vamos a hacer es eliminar la dependencia y la parte del SCM. Deberemos eliminar estas dos partes del pom.xml:

```
<scm>
      <connection>scm:https://svn.sourceforge.net/svnroot/jhotdraw/trunk</co
      nnection>
      <developerConnection>scm:https://svn.sourceforge.net/svnroot/jhotdraw/
      trunk</developerConnection>
</scm>
```

```
            <dependency>
                  <groupId>javax.jdo</groupId>
                  <artifactId>jdo</artifactId>
                  <version>1.0.1</version>
                  <scope>provided</scope>
            </dependency>
```

No podremos compilar el proyecto, pero sí que podemos analizar los ficheros Java. El proyecto tampoco tiene test automáticos, por lo que tampoco habrá problema por no compilar.

Ahora podemos lanzar el análisis contra nuestro servidor local de SonarQube con el comando **mvn sonar:sonar** En unos 2 minutos tendremos el resultado.

Antes de nada debemos recordar que hay tres tipos de issues:

- **Bug**: son fallos que podrían provocar errores inesperados en el sistema. Un ejemplo es el siguiente:

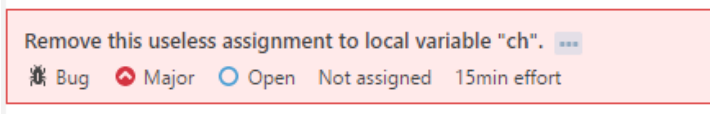


Ilustración 35. Web, issue tipo bug

Asignar un valor a una variable que luego nunca se utilizará puede provocar un error, ya que es posible que no fuera esa la variable a la que quisiéramos poner el valor.

- **Vulnerability**: son errores que se podrían aprovechar por gente malintencionada para hacer que el software funcione de una manera inesperada. Por ejemplo:

Ilustración 36. Web, issue tipo vulnerability

Poniendo ciertos atributos públicos y modificables, podría provocar que alguien externo modifique su valor o cambie su comportamiento. Que sea público no implica que el software llegue a un error inesperado sin intervención externa. Por eso no se considera de tipo bug.

- **Code Smell**: Son fallos que no implican inestabilidad en el sistema, si no que el código sea más fácil de entender y mantener. Por ejemplo:

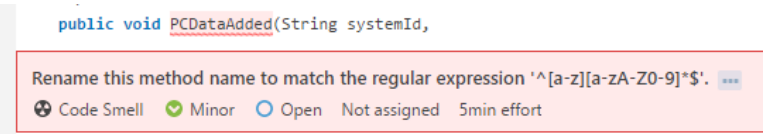


Ilustración 37. Web, issue tipo code smell

Que un método se llame de una manera u otra no provoca problemas de inestabilidad o inseguridad. Lo único que puede provocar es que, un programador que venga después a modificar el código y vea ese nombre de método que no sigue la convección establecida de Java, lo confunda con una clase. Pasado un tiempo se dará cuenta que es un método, pero ya ha hecho que para entender y modificar el código gaste más tiempo del que debiera.

Ya tenemos el informe del análisis. Vamos a entrar en SonarQube y vamos a la parte de proyectos. Lo primero que vemos es el resumen de JHotDraw:

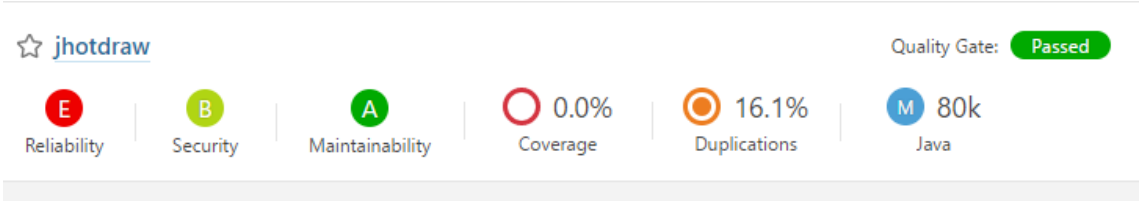


Ilustración 38. Web, resumen reducido de las medidas obtenidas de un proyecto

De aquí podemos leer, en la esquina superior derecha, que ha pasado nuestras exigencias de calidad (según lo que tenemos configurado). Esto significa que SonarQube le da el visto bueno para ponerlo en producción, a pesar de que pueda tener ciertos fallos. Si pusiera *Failed* significaría que antes de poner en producción esta versión, habría que mejorar ciertos aspectos para que pase la evaluación.

Después de izquierda a derecha vemos los siguientes apartados:

- Reliability: es por decirlo de alguna manera, la confianza de que no de errores por bugs. De que el software va a funcionar correctamente. Se basa en los issues de tipo Bug.
- Security: mide la seguridad frente a hackers, observando ciertas vulnerabilidades al leer nuestro código del proyecto. Es obvio que un indicador de 0 vulnerabilidades no es sinónimo de que el software es inhackeable. Solo indica que no tendremos ninguna de las vulnerabilidades más conocidas (y que tenemos configuradas en nuestro perfil de análisis). Se basa en los issues de tipo Vulnerability.
- Maintainability: mide lo fácil o difícil que puede ser mantener el proyecto. Se basa en los issues de tipo Code Smell.
- Coverage: medida del porcentaje de nuestro código que está cubierto por los test automáticos.
- Duplications: porcentaje de código duplicado dentro del proyecto.
- Java: tamaño del proyecto, medido en líneas de código.

La medida de los tres primeros apartados se mide en una escala que va desde la **E** (muy malo) hasta la **A** (muy bueno).

En este proyecto podemos ver que la posibilidad de fallo es muy alta (E). La seguridad es buena, pero aún muy mejorable (B) y que la mantenibilidad es bastante buena (A).

También observamos que la cobertura de los test es del 0.0%, algo obvio cuando no tenemos test, y que el código duplicado implica un 16.1% del código total.

Finalmente se ha catalogado el proyecto como tamaño M, con unas 80.000 líneas de código.

Estos son los tamaños disponibles:

- **XS** menos de 1000 líneas de código.
- **S** entre 1000 y 10.000 líneas de código.
- **M** entre 10.000 y 100.000 líneas de código.
- **L** entre 100.000 y 500.000 líneas de código.
- **XL** más de 500.000 líneas de código

Estos valores vienen configurados por defecto y no se pueden modificar.

Si pulsamos en el nombre del proyecto, podemos ver una ampliación de estos apartados.

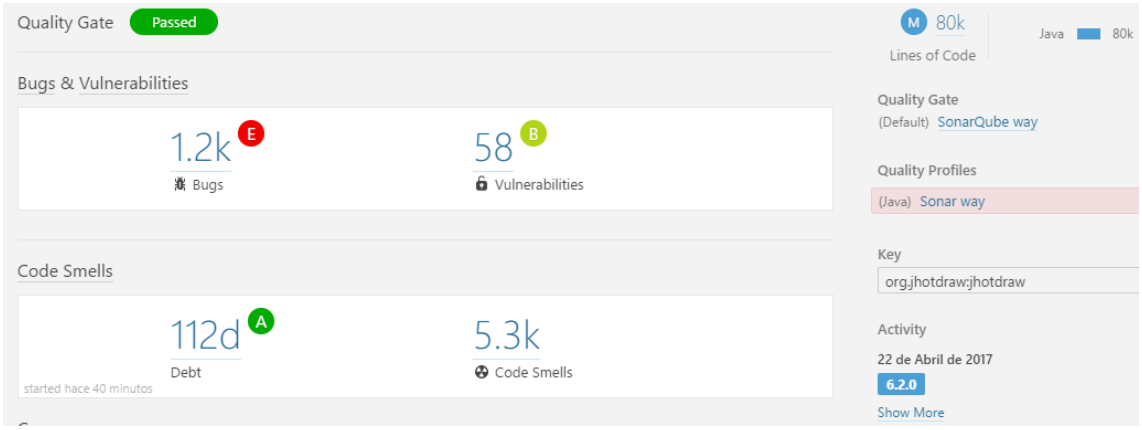


Ilustración 39. Web, parte del resumen de las medidas de un proyecto

Volvemos a ver en la parte superior que ha superado el análisis.

Justo debajo vemos el apartado *Bugs & Vulnerabilities*. Tiene dos medidas: unos 1200 bugs, que le otorgan una calificación de **E**. Esto es lo que antes hemos visto como Reliability. 58 Vulnerabilities, que le da una calificación de **B.** Esto es el apartado Security. La calificación no se obtiene de la cantidad de bugs si no de la gravedad de los mismos. Para ambas medidas se sigue el mismo patrón [16]:

- **A**: 0 vulnerabilities o bugs.
- **B**: al menos 1 vulnerability o bug de gravedad minor.
- **C**: al menos 1 vulnerability o bug de gravedad major.
- **D**: al menos 1 vulnerability o bug de gravedad critical.
- **E**: al menos 1 vulnerability o bug de gravedad blocker.

Estas medidas no son configurables desde la configuración de SonarQube.

Después tenemos el apartado *Code Smells*, que tiene dos medidas: 5300 code smells, y 112 días como deuda técnica. Estos 112 días es lo que se estima que se tardaría en arreglar los cerca de 5300 code smells. Se le otorga una calificación de **A** en base al ratio de deuda técnica que se explicara más adelante. Estos rangos de calificación sí que se pueden configurar en el apartado *Administration>Configuration>Technical Debt*. Aquí hay un apartado llamado *Maintainability rating grid* donde vemos que tiene el siguiente valor: $0.05,0.1,0.2,0.5$. Estos son los porcentajes expresados en un rango de 0 a 1 en los cuales cambia de una calificación mayor, a la siguiente. Es decir, con un ratio de deuda técnica entre el 0 y el 5% se otorga una calificación de A. Con una deuda técnica entre el 5% y el 10% se otorga una calificación de B, y así sucesivamente.

Cada regla tiene un tiempo estimado de resolución. Para obtener la deuda técnica se suma el tiempo de todos los issues de tipo Code Smell.

En la parte derecha vemos un resumen del proyecto y el análisis. Vemos el tamaño del proyecto, y los perfiles que se han aplicado a la hora de hacer el análisis, evaluando así si se cumplen o no las expectativas.

Finalmente quedan los dos últimos apartados de esta página:

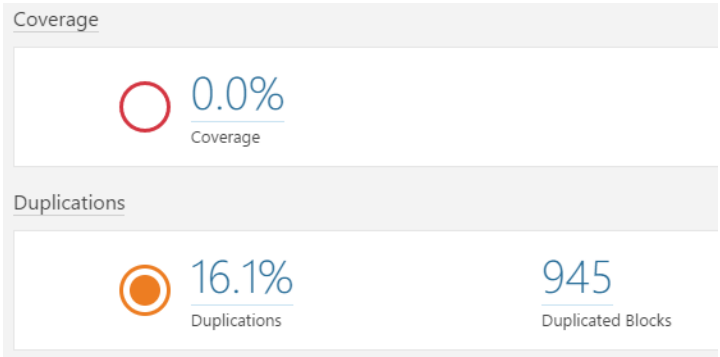


Ilustración 40. Web, parte del resumen de las medidas de un proyecto

El apartado de Cobertura, que nos sigue indicando lo mismo que en la pantalla anterior, un 0% de cobertura.

El apartado de duplicidades, que nos vuelve a indicar el porcentaje de código duplicado, y ahora también nos muestra el número de bloques de código duplicado.

Todo esto sigue siendo un resumen de los resultados. En la parte superior hay un menú para desplazarnos entre diferentes secciones, donde veremos los resultados más detallados.

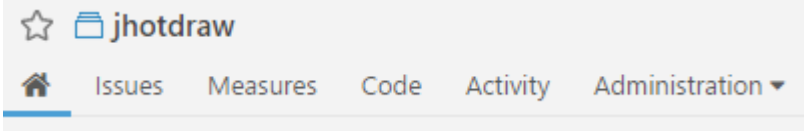


Ilustración 41. Web, menú de un proyecto

En la pestaña issues tendremos el buscador de issues, con numerosos filtros para poder refinar la búsqueda.

Recordemos que en nuestro código, cada vez que se incumple una de las reglas establecidas en el perfil, se crea un issue para avisarnos que está pendiente de solucionar.

Un ejemplo de issue puede ser el siguiente:

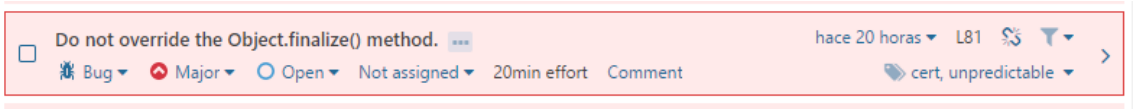


Ilustración 42. Web, ejemplo de la información de un issue

Tiene un título descriptivo de porqué está mal. Debajo del título dispone de cierta información como el tipo de fallo (Bug, Vulnerability o Code Smell), la gravedad, el estado del issue, persona asignada para resolverlo, tiempo de resolución estimado, y posibles comentarios que se pueden añadir.

Si pulsamos en los tres puntos a la derecha del título, nos abrirá una ventana con la regla incumplida que ha provocado este issue.

Si hacemos doble click en la línea, o clicamos en la flecha > situada a la derecha del todo, nos llevará al fichero y a la línea o líneas que incumplen la regla.

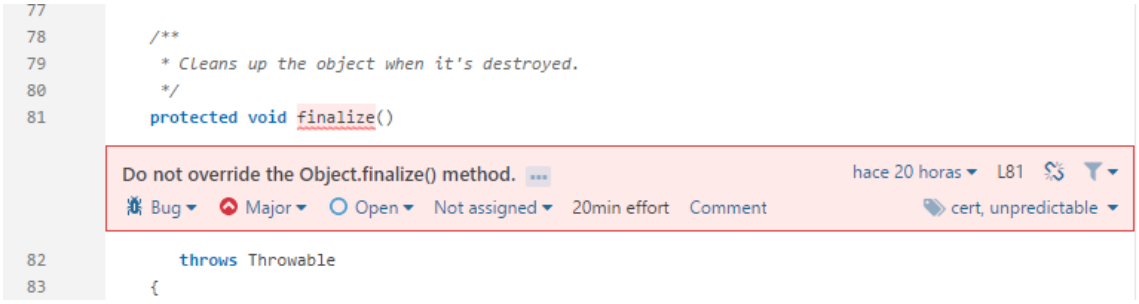


Ilustración 43. Web, ejemplo de código con un issue incrustado

Podemos ver cómo incrusta, entre líneas de código, la línea con la información del issue. Por eso, si tenemos muchos issues, al ver el código donde están los fallos, podríamos llegar a ver más resúmenes de issues que líneas de código, por lo que perderíamos la visibilidad del código.

En la pestaña *Measures* podremos ver los datos de las medidas obtenidas. Aparecerá un submenú para acceder a cada medida.

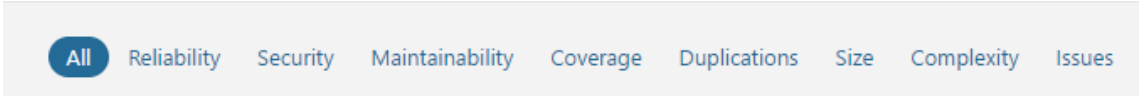


Ilustración 44. Web, submenú del apartado measures

Todo lo que hay en el apartado **All** se puede ver en cada uno del resto de apartado de medidas, por lo que los veremos uno por uno

- **Reliability** Medidas relacionadas con las reglas de tipo Bug incumplidas. Podemos ver el número exacto de bugs (1224), su calificación (**E**) y el tiempo estimado para solventar todos los bugs.

Debajo podremos ver un gráfico de burbujas que representa nuestros ficheros Java, en este caso, en dos dimensiones. En el eje X se establece el tamaño en líneas de código del fichero, y en el eje Y el tiempo estimado de resolución de todos los bugs que existen en ese fichero. Por lo tanto, cuanto más a la derecha esté el círculo que representa el fichero, mas líneas de código tendrá, y cuanto más arriba esté, más horas se necesitarán para dejarlo sin bugs.

También se puede observar que unos círculos son más grandes que otros. Esto es debido a que el tamaño de los círculos representa el número de bugs que tiene.

No debemos confundir el número de bugs con el tiempo de resolución de estos. Por eso se representan las dos medidas.

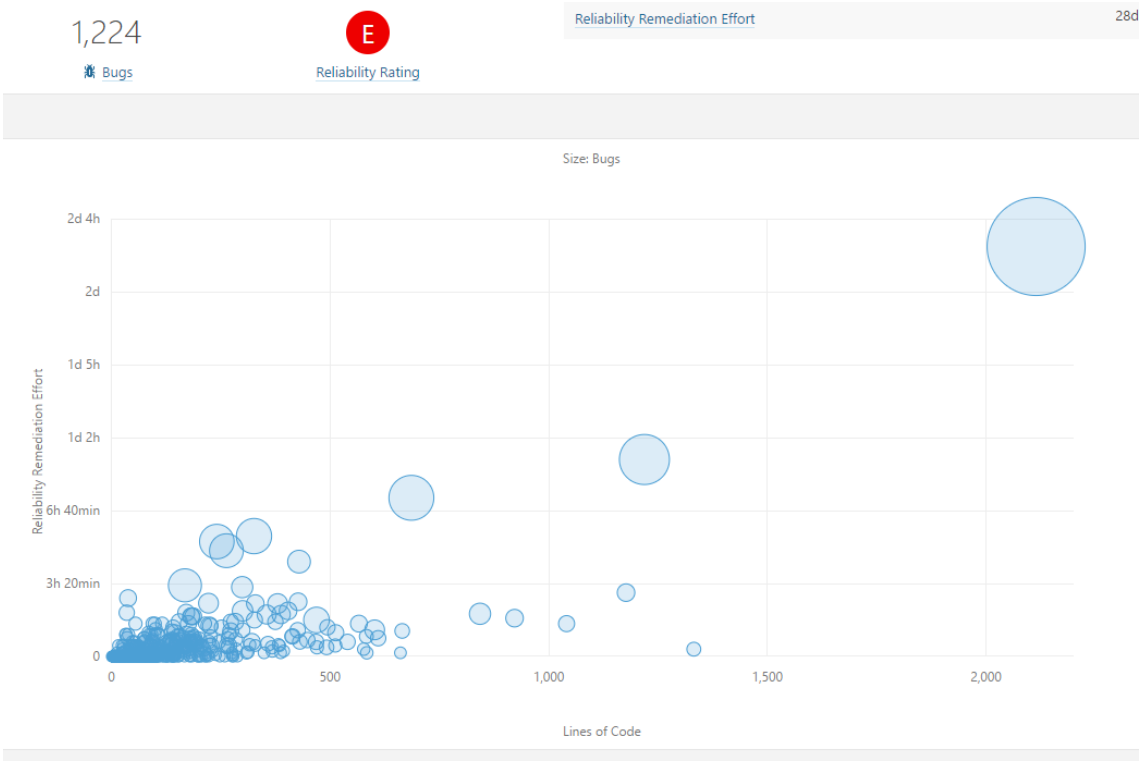


Ilustración 45. Web, detalle de Reliability

En la parte superior donde nos indica las medidas podemos pinchar en las palabras *Bugs*, *Reliability Rating* y *Reliability Remediaton Effort*. Esto nos llevará a un listado de los ficheros de nuestra aplicación, en el que en la parte derecha aparecerá el número de bugs, la calificación y el tiempo de resolución estimado respectivamente por cada fichero. Este listado aparece ordenado de mayor a menor.

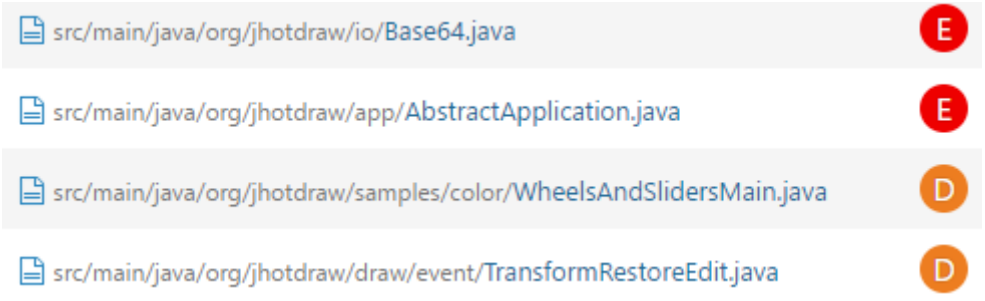


Ilustración 46. Web, parte del listado de ficheros con su calificación de Reliability

- **Security** Este apartado es exactamente igual al anterior, solo que las medidas son relacionadas con las issues de tipo Vulnerability encontrados. También nos indica el número de issues de este tipo, la calificación obtenida en este apartado (**B**) y el tiempo de resolución estimado. Igualmente posee un gráfico de burbujas para representar los ficheros en función de su tamaño en líneas de código, el tiempo de resolución y la cantidad de issues.

Podremos ver los datos por cada fichero pulsando en *Vulnerabilities*, *Security rating* y *Security Remediation Effort*.

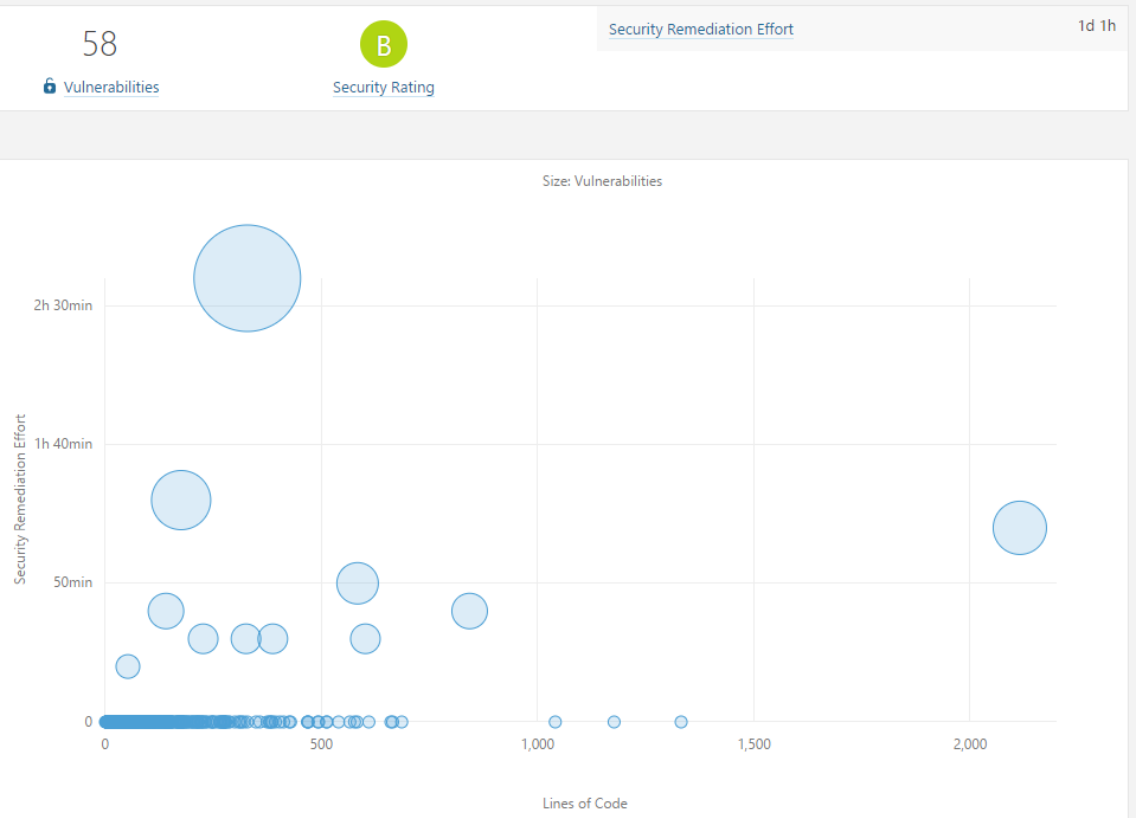


Ilustración 47. Web, detalle de Security

- **Maintainability** De forma muy similar a los dos anteriores, este apartado nos muestra las medidas sobre los issues del tipo Code Smell.

Este apartado tiene alguna medida más en la cabecera.

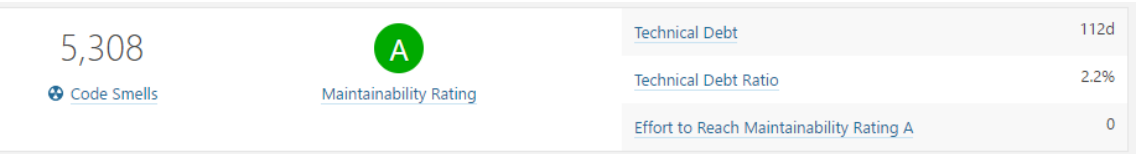


Ilustración 48. Web, detalle de Maintainability

Technical Debt es lo que antes decíamos que era el tiempo estimado de resolución. En este caso se conoce como deuda técnica.

Technical Deb Ratio es la proporción de tiempo que tenemos que dedicar a arreglar estos fallos, con respecto al tiempo ya dedicado a desarrollar todo el proyecto. En este caso, con un ratio de deuda técnica del 2.2% nos indica que, arreglar los 5308 issues de Code Smells, nos llevará el 2.2% del tiempo que nos ha llevado desarrollar todo el proyecto. Estos cálculos son

aproximados, ya que el dato del tiempo que hemos tardado en desarrollar todo el proyecto no es real, sino calculado según lo que hayamos establecido en las propiedades de SonarQube. Como no hemos tocado nada de las propiedades, aún está puesto el que trae SonarQube por defecto.

Este cálculo del ratio es muy sencillo de obtener. En las propiedades viene establecido por defecto que desarrollar una línea de código nos lleva 30 minutos, por lo que este proyecto de 80.000 líneas de código nos habrá llevado → 2.400.000 minutos → 40.000 horas.

Por otro lado, la deuda técnica según el tiempo de cada issue es de 112 días. Estos días son laborables, por lo que cada día son 8 horas. Entonces tenemos que la deuda técnica es de 896 horas. Si calculamos el ratio esto es: 896/40.000 = 0.0224 → **2.2%**

Estos datos hay que tomarlos con mucho cuidado, ya que la suposición de que desarrollar una línea de código nos lleva 30 minutos puede ser mucho suponer. Todo depende del lenguaje que usemos, de lo complicado que sea el proyecto, de la experiencia del desarrollador.

Effort to reach maintainability Rating A es el tiempo necesario para hacer que el proyecto tenga una mantenibilidad de nivel A.

- **Coverage** Medidas sobre la cobertura de los test. En este proyecto, como no hay test, no podemos ver mucho, pero es muy similar a los anteriores. Lo veremos en el siguiente proyecto que sí que dispone de test.
- **Duplications** Medidas sobre el código duplicado [16].

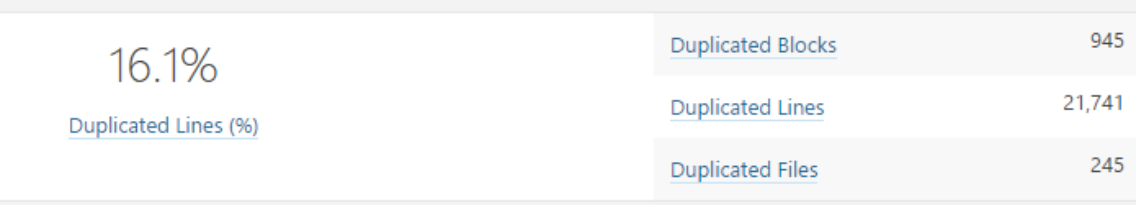


Ilustración 49. Web, medidas de código duplicado

En la parte derecha podemos ver el número de bloques de código duplicado, el número de líneas duplicadas, y el número de archivos que tienen algún bloque duplicado.

Para obtener estas medidas, SonarQube busca bloques de código idénticos. En java se considera dos bloques idénticos cuando tiene 10 *statements* de código iguales seguidos. Sin importar los *tokens* o el número de líneas.

En el resto de lenguajes también se considera que dos bloques son iguales si tienen 100 *tokens* seguidos iguales.

Por ejemplo, tenemos dos ficheros java en los que en uno hemos implementado una función que después hemos hecho copia y pega en otro fichero del proyecto. En este segundo fichero hemos pegado la función dos veces variando solo el nombre para que no de errores. Cuando SonarQube analice las duplicidades encontrará que el código de la primera función está repetido dos veces más. Considerará el código como un bloque, y nos informará que hay tres bloques de código duplicado. Por lo que en la medida de *Duplicated Blocks* nos indicará un 3. Un bloque por cada función repetida.

Cada bloque de código tendrá un número determinado de líneas, y no tiene por qué coincidir en los tres bloques, ya que en uno de ellos hemos podido formatear el código de otra

manera, o haber incluido más líneas en blanco o comentarios. La suma de líneas de cada bloque nos dará la siguiente medida, *Duplicated Lines*.

La medida de *Duplicated Files* cuenta el número de ficheros que tiene algún bloque de código duplicado. En nuestro ejemplo serían 2.

Finalmente el porcentaje de código duplicado se obtiene de la relación entre las líneas de código duplicadas entre las líneas totales del proyecto. Las líneas totales del proyecto incluyen tanto las líneas de código como las de comentarios y líneas en blanco.

En la imagen del proyecto que estamos analizando tenemos que hay 21.741 líneas de código duplicado, y el proyecto tiene 134.687 líneas. 21.741/134.687 = 0,1614 → **16,1%**

- **Size** En este apartado veremos información sobre el tamaño del proyecto.

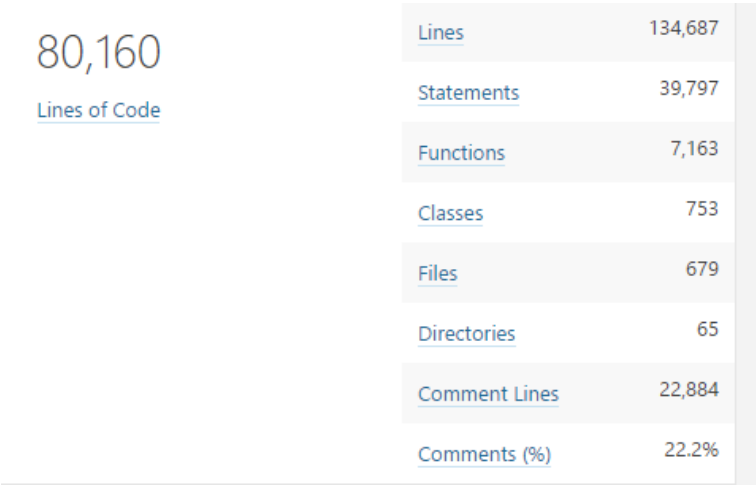


Ilustración 50. Web, medidas del tamaño del proyecto

  - Lines of Code Es el número de líneas de código sin contar líneas en blanco ni comentarios.
  - Lines Podemos ver las líneas totales, resultado de sumar las líneas de cada archivo. Tiene en cuenta todo, ya sean líneas en blanco, líneas de comentarios, líneas de código…
  - Statements Cuenta cuántas sentencias de Java existen en el código
  - Functions nos da el número de funciones de nuestro código.
  - Classes es el número de clases java que existen en el proyecto.
  - Files el número de ficheros.
  - Directories es el número de carpetas.
  - Comment Lines es el número de líneas de comentarios.
  - Comments (%) es el porcentaje de comentarios respecto del número de líneas que tienen contenido (suma de líneas de código mas líneas de comentarios).

    En el proyecto tenemos 22884 líneas de comentarios y 80160 líneas de código:

    22884/(22884+80160) → 22884/103044 = 0,22207 → **22,2%**

Al igual que en apartados anteriores, si vamos pulsando en cada una de estas medidas, nos lleva al listado de ficheros donde podemos ver esa medida pero desglosada en ficheros.

- **Complexity** Medidas sobre la complejidad del código.

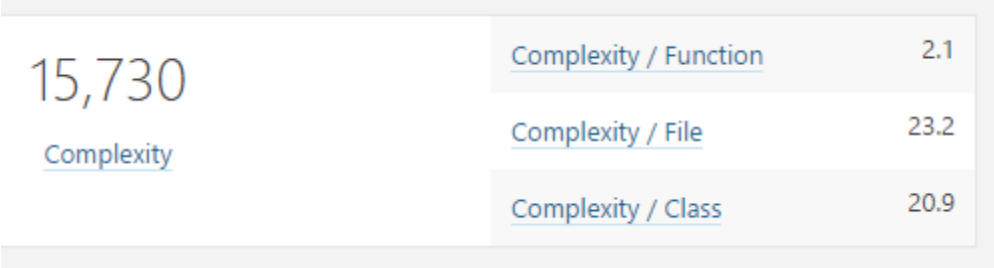


Ilustración 51. Web, medidas de complejidad del proyecto

Podemos ver tanto la complejidad total (15.730) como la complejidad media por función, por fichero y por clase.

Pulsando en cada medida podemos ver cada una de ellas desglosada por fichero.

Esta complejidad se refiere a la complejidad ciclomática que mide la complejidad lógica de nuestro software. Es decir, el número de caminos que el flujo del software puede tomar en una ejecución. Aunque SonarQube nos de la complejidad total del proyecto y las medias de fichero y clases, las más interesantes son las complejidades de las funciones. La complejidad total es la suma de las complejidades de todas las funciones.

Con la complejidad de una función se pretende medir los caminos independientes que puede tomar la ejecución cuando entra en esa función. Se ve más claro con un ejemplo. Supongamos el siguiente trozo de código.

```
public boolean metodo(int valor) {
	if (valor < 5) {
		return true;
	} else {
		return false;
	}
}
```

Cuando el flujo de ejecución entre en el anterior método, pueden suceder dos cosas, que el valor sea menor de cinco y se ejecute el código del *If*, o que el valor sea cinco o mayor y se ejecute el código del *Else*. Esto nos da que la complejidad de la función es 2, ya que solo hay dos caminos de ejecución.

A la hora de hacer tests esto es importante, puesto que podríamos pensar que con llamar a la función una vez con un valor y obtener el resultado esperado, hemos comprobado que funciona correctamente, y no es así. Con eso hemos comprobado que uno de los caminos es correcto. ¿Y si el código de la parte que no se ha ejecutado no funciona? Por eso debemos probar los dos caminos que puede tomar el código.

La complejidad ciclomática de una función nos da el mínimo número de test que necesitaremos para comprobar todos los caminos posibles. Un test por cada camino.

La complejidad de una función debe ser lo más pequeña posible. Si una función tiene una complejidad muy alta será muy difícil de entender y mantener, por lo que deberíamos pensar en reducir esa complejidad separándola en funciones más pequeñas y simples.

No existe un límite de cuál es la máxima complejidad que se debería permitir, pero se habla de que no debería superar el límite de 20 [17]. Algunos son más estrictos y dicen 10 ya es un buen límite máximo [18], y otros establecen que a partir de 6 hay que pensar en refactorizar [19]. También comentan que no hay que ser del todo estricto. Puede que existan funciones con una complejidad de 15 que no se deban simplificar [18]. Es posible que tengamos un algoritmo medianamente complejo y queramos tener todo el algoritmo encapsulado en una sola función.

- **Issues** Medidas sobre los issues

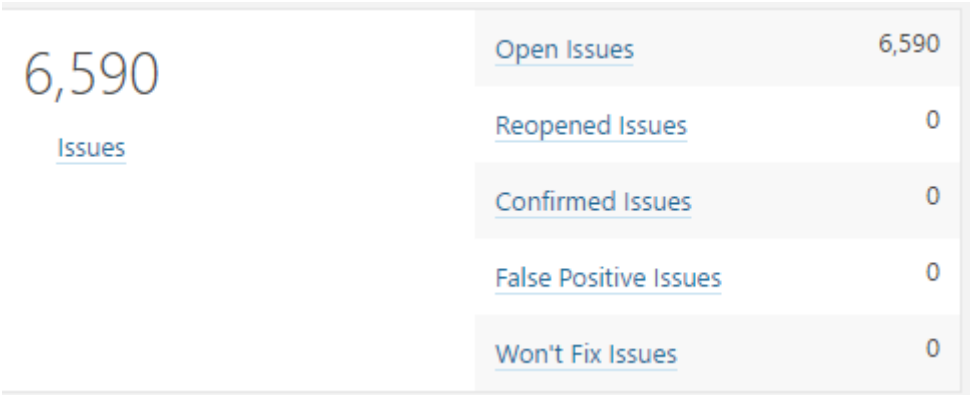


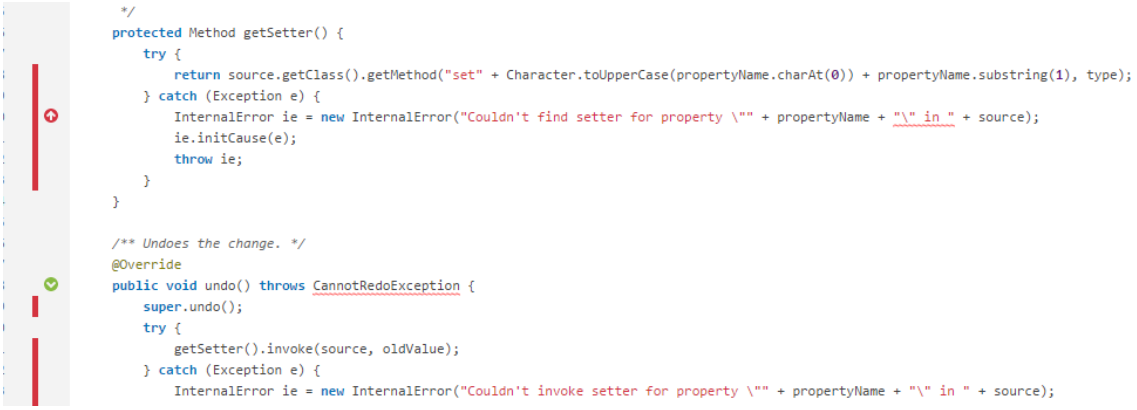
Ilustración 52. Web, medidas sobre el número de issues

Vemos cuántos issues existen, cuántos están abiertos, cuántos están reabiertos, cuántos confirmados, cuántos son falsos positivos, y cuántos no se van a resolver.

De la misma manera que en apartados anteriores, si entramos en cada uno de los apartados podemos ver la medida desglosada por ficheros.

Si volvemos al submenú superior nos queda el apartado Code, donde obtendremos un listado de los directorios con algún fichero, y si clicamos en el directorio podemos ver todos los archivos que contiene pudiendo ver el código de cada fichero.

En esta vista del fichero podemos apreciar el código cubierto o no cubierto por test con una barra de color a la izquierda. Ya veremos más adelante los significados. También podemos ver los issues que ha generado el fichero. En la parte izquierda vemos el icono de la gravedad, y en el propio código una parte subrayada en rojo que indica dónde está el error.



```
 */
protected Method getSetter() {
    try {
        return source.getClass().getMethod("set" + Character.toUpperCase(propertyName.charAt(0)) + propertyName.substring(1), type);
    } catch (Exception e) {
        InternalError ie = new InternalError("Couldn't find setter for property \"" + propertyName + "\" in " + source);
        ie.initCause(e);
        throw ie;
    }
}

/** Undoes the change. */
@Override
public void undo() throws CannotRedoException {
    super.undo();
    try {
        getSetter().invoke(source, oldValue);
    } catch (Exception e) {
        InternalError ie = new InternalError("Couldn't invoke setter for property \"" + propertyName + "\" in " + source);
```

Ilustración 53. Web, ejemplo de código con errores

Si pinchamos en el icono de la gravedad aparecerá la información del issue.

8. CASO PRÁCTICO: JFREECHART

JFreeChart es una biblioteca Java para poder representar gráficos en nuestros programas Java [20].

Usaré la última versión, la 1.0.19. Se puede descargar desde aquí:

https://sourceforge.net/projects/jfreechart/files/1.%20JFreeChart/1.0.19/jfreechart-1.0.19.zip/download

Una vez descomprimida la carpeta del proyecto, podemos ver que está configurado tanto con Maven como con Ant. Usaremos Maven por comodidad, ya que no deberíamos modificar nada, simplemente compilamos y lanzamos el análisis.

La primera vez que compilemos nos falla. Dice que han fallado los test y nos indica cuál ha fallado concretamente. Falla un test para comprobar los parsers de un objeto Month. Se intenta parsear el texto “March 1993” y, al comprobar si el objeto devuelve el número de mes correcto, el 3, falla por que devuelve un 1. Viendo el código del test veo que está dentro de un try-catch, y en el catch construye un mes con el valor 1, lo que me hace pensar que el parser está fallando y crea el objeto en el catch.

El mes está escrito en inglés, y mi sistema operativo lo tengo en español, por lo que puede que sea ese el fallo. Tengo dos opciones, o cambiar el Locale de mi sistema operativo, o modificar el test y poner el mes en español.

Compruebo que poniendo el mes en español, pero en mayúsculas, sigue sin funcionar, pero si pongo marzo con todo en minúsculas el test pasa correctamente.

Quedan otros dos fallos para poder compilar y analizar. En SonarQube debemos instalar el plugin de SVN debido a que el POM de este proyecto está conectado a su repositorio mediante la configuración scm de Maven, y al intentar analizarlo, si no disponemos del plugin falla. También en el POM, en el plugin de maven-gpg-plugin debemos añadir las siguientes líneas para que ignore el goal [21].

```
<configuration>
    <skip>true</skip>
</configuration>
```

Este plugin es para firmar los artefactos generados por Maven, pero como no tenemos instalado el software necesario, falla, por lo que le indicamos que ignore esta firma [22].

Una vez hechas estas modificaciones ya podemos ejecutar el comando **mvn clean install sonar:sonar** y funcionaría correctamente. Pero nos falta un detalle, obtener el informe de cobertura para que se añada a la información que nos muestra SonarQube. Para ello necesitamos que se genere el informe al pasar los test, e indicarle a SonarQube dónde está dicho informe.

Para obtener el informe voy a usar Jacoco. Solo hay que añadir el siguiente plugin al POM [23]:

```
<plugin>
	<groupId>org.jacoco</groupId>
	<artifactId>jacoco-maven-plugin</artifactId>
	<version>0.7.9</version>
	<configuration>
		<destFile>${basedir}/target/coverage-reports/jacoco-unit.exec</destFile>
		<dataFile>${basedir}/target/coverage-reports/jacoco-unit.exec</dataFile>
	</configuration>
	<executions>
		<execution>
			<id>jacoco-initialize</id>
			<goals>
				<goal>prepare-agent</goal>
			</goals>
		</execution>
		<execution>
			<id>jacoco-site</id>
			<phase>package</phase>
			<goals>
				<goal>report</goal>
			</goals>
		</execution>
	</executions>
</plugin>
```

En la parte de propiedades debemos añadir una más para indicar a SonarQube dónde está el reporte de Jacoco. Las propiedades quedarán así habiendo añadido nosotros la última [24]:

```
<properties>
	<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	<project.source.level>1.6</project.source.level>
	<project.target.level>1.6</project.target.level>
	<sonar.jacoco.reportPaths>${basedir}/target/coverage-reports/jacoco-
unit.exec</sonar.jacoco.reportPaths>
</properties>
```

Ahora cuando realicemos el análisis incluirá la información de cobertura en SonarQube.

Este es nuestro nuevo informe que vamos a analizar:

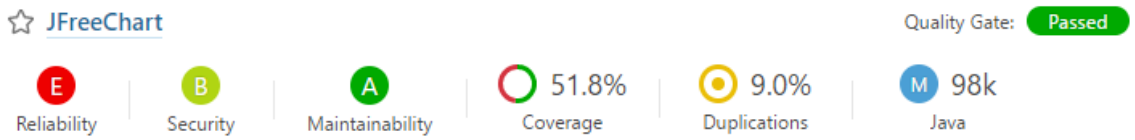


Ilustración 54. Web, resumen reducido de medidas de un proyecto

La diferencia con el anterior es que en este nos indica qué código cubierto por los test es del 51.8%.

En este nuevo proyecto SonarQube ha dado por bueno el proyecto. Tiene un alto riesgo de fallo, una seguridad ligeramente mejorable y una buena mantenibilidad. La mitad del código está cubierto por test y tenemos un 9% de código duplicado.

Ya vimos que significaba el resto de medidas del proyecto, solo nos falta la parte de cobertura, por lo que es en la que me centraré en este apartado.

Si nos vamos a la parte de medidas del proyecto encontraremos la siguiente:

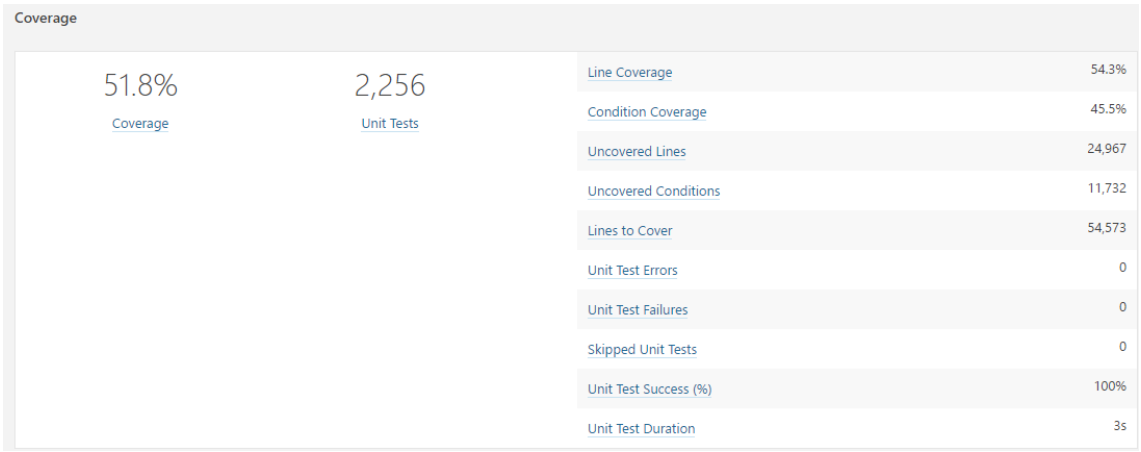


Ilustración 55. Web, medidas sobre la cobertura de los test automáticos

Podemos ver a la izquierda del todo la cobertura total del código. Más a la derecha nos indica el número de test unitarios, 2256 en este proyecto. No tiene que existir un fichero por cada test, es posible que dentro de un mismo fichero existan varios test relacionados.

En la parte derecha nos indica en el siguiente orden:

- Porcentaje de líneas cubiertas por algún test
- Porcentaje de condiciones comprobadas. Esto se da en estructuras condicionales como los *If*. Puede que tengamos un *If* en el que el test haga un caso de prueba en el cual la evaluación sea true, pero nos quedaría probar el caso en el que fuera false.

  También es posible casos más complejos como un *If* con dos condiciones unidas mediante el operador and. Puede que hagamos un test en el que las dos condiciones sean true, y otro en el que una de ellas sea false, por lo que la condición total se evalúa como false. En este caso no solo nos basta con comprobar esos dos casos. Nos faltaría la condición en el que los dos fueran false, y la condición en el que uno fuera true y el otro false, diferente de la anterior.
- Número de Líneas aún no cubiertas por test
- Número de condiciones no cubiertas
- Líneas totales que hay que cubrir mediante test unitarios.
- Test que han tenido error
- Test que han tenido fallo
- Test no ejecutados
- Porcentaje de test satisfactorios
- Duración de la ejecución de los test unitarios

Al igual que en el resto de apartados, si pulsamos sobre el nombre nos lleva a un listado desglosado por fichero.

9. NUEVAS REGLAS USANDO XPATH

Introducción a XPath [25]–[28]

XPath es un lenguaje de consulta que permite seleccionar nodos de un documento XML a través de una ruta o expresión llamada Path.

Este Path puede incluir ciertas restricciones para evaluar si un nodo debe o no ser seleccionado.

El Path tiene el siguiente aspecto:

eje::testDeNodo[predicado]

XPath considera el documento XML como un árbol de nodos. Existen siete tipos de nodos diferentes:

- Raíz
- Elemento
- Atributo
- Texto
- Comentario
- Instrucciones de procesamiento
- Espacio de nombres

El nodo raíz es un nodo virtual y único en todo el documento que tiene como único hijo el elemento raíz del documento. Se representa mediante el símbolo /

Dado el siguiente XML vamos a ver varios ejemplos de nodos:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<biblioteca>
	<libro idioma="es">
		<titulo>Construcción de software orientado a objetos</titulo>
		<autor>Bertrand Meyer</autor>
		<anyo>1999</anyo>
	</libro>
	<libro idioma="en">
		<titulo>Clean code</titulo>
		<autor>Rober C. Martin</autor>
		<anyo>2008</anyo>
	</libro>
</biblioteca>
```

<biblioteca> Es el elemento raíz (diferente del nodo raíz) y es un nodo de tipo elemento

<titulo>Clean code</titulo> otro nodo elemento

idioma="es" nodo atributo

El anterior XML se puede representar como un árbol de nodos, el cual se presenta a continuación. Cada uno de los recuadros representa un nodo diferente, y los colores simbolizan los diferentes tipos de nodos. En este ejemplo solo tenemos un nodo raíz, varios nodos elementos, dos nodos atributo y seis nodos texto.

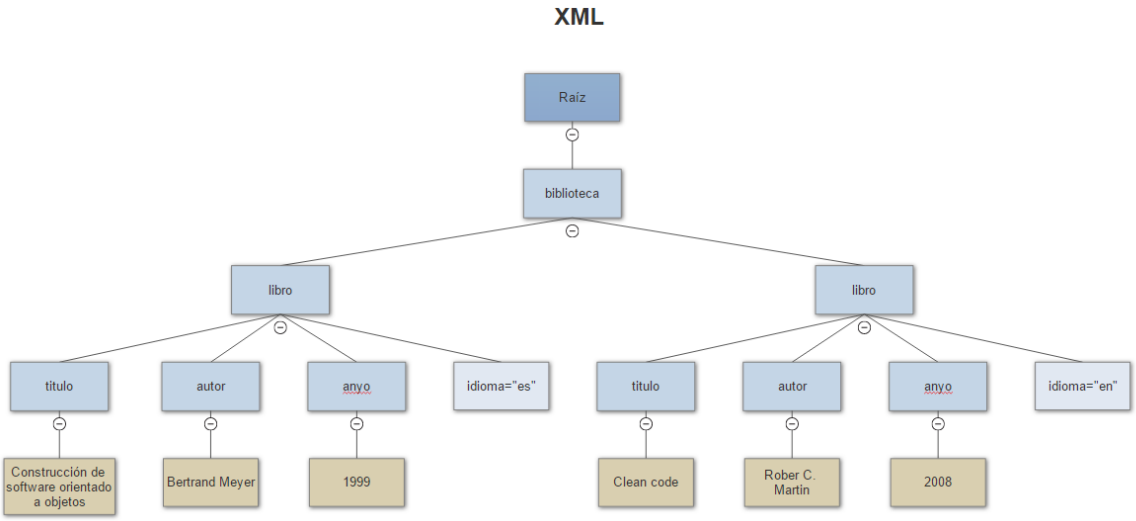


Ilustración 56. Esquema de la arquitectura de un XML

Vemos como efectivamente un XML se puede representar en una estructura de árbol, y mediante los Path de XPath podremos navegar y seleccionar nodos de esta estructura.

Dada la estructura de una expresión XPath que vimos al principio, en la parte de *testDeNodo* es donde indicamos el nombre de los nodos que vamos a seleccionar (biblioteca, libro, titulo, autor, anyo, idioma).

Podría quedarnos una expresión así:

$$\text{eje::libro[predicado]}$$

Los nodos se relacionan unos con otros con lo que se conoce como ejes de navegación. Estos ejes de navegación indican, partiendo del nodo/s contexto, cuál debemos seleccionar. Existen ejes de navegación y son los siguientes:

- Self
- Child
- Descendant
- Descendant-or-self
- Parent
- Ancestor
- Ancestor-or-self

- Preceding
- Following
- Preceding-sibling
- Following-sibling
- Attribute
- Namespace

Estos ejes son los que sustituiremos en la expresión XPath donde aparecía la palabra *eje.*

Hay que mencionar que, cuando empezamos una expresión XPath, el nodo contexto del que partimos es el nodo raíz (que no el elemento raíz), por lo que el primer eje se aplica sobre este nodo.

Siguiendo el ejemplo de seleccionar los nodos elemento *libro*, partimos del nodo raíz, y para poder llegar a los elementos *libro* con un solo eje deberíamos seleccionar un eje que nos de todos los nodos que hay por debajo del nodo raíz, o lo que es lo mismo, todos los descendientes. Yo voy a seleccionar el eje *descendant-or-self*, por lo que la expresión ahora tendría la siguiente forma:

descendant-or-self::libro[predicado]

Una expresión XPath también puede incluir un predicado, que no es más que una expresión booleana que decide si un nodo se incluye o no en el conjunto.

Por ejemplo, usando el XML del principio, podríamos querer seleccionar los nodos *libro*, pero solo los que estén en español. Para ello se indicaría que se seleccionen todos los nodos libro, pero aplicándole un predicado le decimos que, de todos esos nodos, solo nos quedamos con los que tengan el atributo *lenguaje* con el valor *es.*

Dentro del predicado podemos seleccionar otros nodos usando los ejes de navegación, tomando como nodo contexto los que se acaban de seleccionar justo antes de empezar el predicado.

descendant-or-self::libro[attribute::idioma='es']

El predicado deberá generar un valor booleano, y para ello podemos usar operadores o funciones.

Los operadores disponibles son los siguientes:

- and
- or
- =
- !=
- <
- <=
- >
- >=

- +
- -
- *
- div

En cuanto a las funciones, XPath provee numerosas funciónes que podemos usar. Aquí podemos ver algunos ejemplos del total de funciones:

- count()
- starts-with()
- string-length()
- last()

Con todo lo visto hasta ahora ya podemos construir nuestras expresiones, pero existe una forma de reducir su tamaño. Para ello, algunos de los ejes de navegación tienen una abreviatura que hace a la expresión menos verbosa.

- child:: (no indicar nada)
- self:: .
- parent:: ..
- attribute:: @
- descendant-or-self:: //
- Todos los hijos *
- Todos los atributos @*
- [position()=n] [n]

Por lo que el ejemplo que veníamos haciendo hasta ahora quedaría así:

```
//libro[@idioma='es']
```

Estos paths se pueden ir concatenando para refinar más la búsqueda. Partiendo del ejemplo anterior, imaginemos que queremos los nombres de los autores de libros en español. Ya tenemos los libros en español, solo nos queda seleccionar el autor, y quedarnos con el texto:

```
//libro[@idioma='es']/autor/text()
```

Finalmente, una expresión en XPhat puede devolver cuatro valores diferentes:

- Un conjunto de nodos
- Un número
- Un valor booleano true/false
- Una cadena de caracteres

ADVERTENCIA: durante el transcurso del proyecto he descubierto que SonarQube ya no cuenta con esta forma de crear nuevas reglas para Java [29], [30].

Esta funcionalidad no la proporciona SonarQube, si no el plugin de Java para SonarQube, que actualmente se llama SonarJava. Los desarrolladores decidieron eliminarla en la versión 2.5 del plugin, argumentando que esa forma de crear nuevas reglas no es fiable. Se crean *paths* basados en el AST del fichero que se está analizando, confiando que la forma de este AST no cambie nunca. Pero la forma del AST puede cambiar de una versión a otra, lo que hacía que las reglas dejaran de funcionar sin que los usuarios se dieran cuenta. También argumentan que existe la posibilidad de crear nuevas reglas mediante Java, que es mucho más potente y fiable.

Por lo que la última versión del plugin que dispone de dicha funcionalidad es la 2.4 (julio de 2014). Actualmente la última versión disponible es la 4.9, por lo que es una diferencia notable.

Los plugins en SonarQube no solo se pueden instalar desde el centro de actualizaciones, también podemos descargar el .jar del plugin que queremos e instalarlo de forma manual. Para ello, teniendo el jar de la versión 2.4 que se puede descargar desde aquí:

http://central.maven.org/maven2/org/codehaus/sonar-plugins/java/sonar-java-plugin/2.4/sonar-java-plugin-2.4.jar

Vamos a la siguiente ruta: {SONARQUBE_HOME}/extensions/plugins

Si teníamos plugins ya instalados, estarán ahí sus .jar. Lo que queremos ahora es eliminar la versión 4.9 de SonarJava para poner la 2.4, por lo que localizamos el jar del plugin que queremos eliminar, y colocamos el que acabamos de descargar. Cuando arranquemos de nuevo SonarQube, si vamos al centro de actualizaciones, vemos que ya tenemos la versión 2.4.

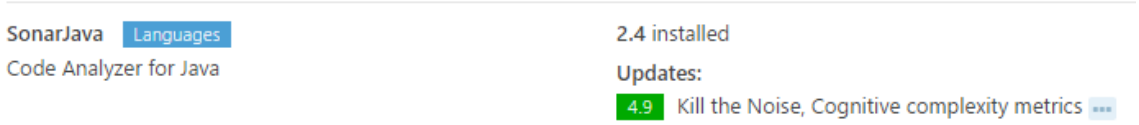


Ilustración 57. Web, detalle de plugin instalado donde se ve la versión

He comprobado que ya se pueden crear nuevas reglas de Java mediante XPath. He creado una nueva regla muy simple de ejemplo para comprobar que funciona, pero a la hora de analizar un proyecto con Maven salta el siguiente error:

```
[ERROR] Failed to execute goal org.sonarsource.scanner.maven:sonar-maven-plugin:3.2:sona
r (default-cli) on project KataFizzBuzz: Unable to register extension org.sonar.plugins.
jacoco.JaCoCoSensor from plugin 'java': Lorg/sonar/api/scan/filesystem/ModuleFileSystem;
: org.sonar.api.scan.filesystem.ModuleFileSystem -> [Help 1]
```

Ilustración 58. Error al analizar proyecto

Investigando el problema he descubierto que la clase *ModuleFileSystem* perteneciente al API de SonarQube ha sido eliminada en la versión 6.1 de SonarQube [31]. Estaba en deprecated desde la versión 4.2 [32].

Por lo que aún teniendo la versión 2.4 del plugin de Java, no nos es suficiente para que funcione. Este plugin tiene varias dependencias con el API de SonarQube que actualmente ya no existen. Usando la versión 6.0 tampoco es suficiente, ya que sigue fallando otras dependencias.

La versión más reciente que he encontrado en la que funcione el plugin de Java es la 4.5.7 que se puede descargar desde este enlace:

https://sonarsource.bintray.com/Distribution/sonarqube/sonarqube-4.5.7.zip

En su momento fue una versión LTS, por lo que la última actualización fue en Abril de 2016 [33].

Conclusión: usando la versión 4.5.7 de SonarQube y la versión 2.4 del plugin SonarJava, podremos comprobar cómo funcionaba el mecanismo de creación de nuevas reglas mediante XPath.

Aunque esta forma de crear nuevas reglas se eliminó para Java, aún hay ciertos lenguajes que siguen disponiendo de ella. Por ello, aunque no nos valga para crear nuevas reglas en Java, lo aquí aprendido lo podemos usar en otros lenguajes. Concretamente: Flex, PL/SQL, PL/I, Python y XML.

Ahora que ya tenemos un nuevo entorno donde poder probar las reglas en XPath para Java, veamos cómo se desarrollan estas.

AST y SSLR Toolkit

Una de las fases de la compilación de un fichero Java a bytecode es la traducción del código java a un árbol de sintaxis abstracta o abstract syntax tree (AST). Esta etapa se la conoce como análisis sintáctico [34].

Al tener una jerarquía de árbol, este AST se puede interpretar como un XML del cual podremos obtener ciertos nodos usando expresiones de XPath.

Para poder investigar cómo es el AST de cierto código fuente, existe un programa desarrollado por SonarQube, el cual nos muestra cómo es el AST de un código que le indiquemos. Este programa se conoce como SSLR Toolkit. Existen versiones diferentes para cada lenguaje. La última versión para Java la podemos encontrar en el siguiente enlace.

https://mvnrepository.com/artifact/org.codehaus.sonar-plugins.java/sslr-java-toolkit/2.4

Es un JAR autoejecutable. Una vez lo abramos nos mostrará la siguiente interfaz:

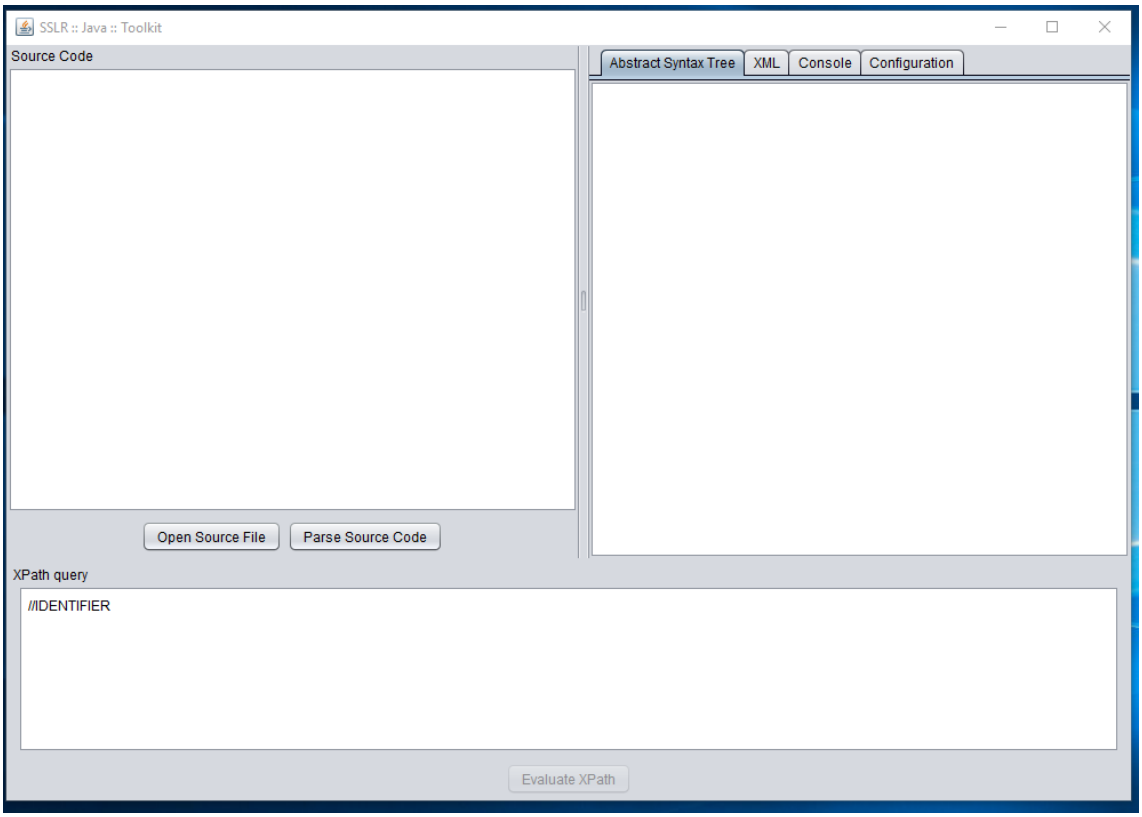


Ilustración 59. Ventana del programa SSLRToolkit

Es muy simple. En el recuadro de la izquierda pondremos el código que queremos investigar. También podremos abrirlo desde un fichero con el botón "*Open source file*".

En el recuadro de la derecha podremos ver el AST de diferentes formas: como una jerarquía desplegable al estilo explorador de directorios o como un XML. Para verlo así, una vez que tenemos el código en el recuadro de la izquierda, pulsamos en "*Parse Source Code*" y veremos el resultado.

Finalmente, en el recuadro inferior podremos escribir una expresión XPath y evaluarla sobre el AST que acabamos de generar. Esto hará que los nodos resultantes se remarquen en ambos recuadros superiores.

Veamos un ejemplo en la siguiente imagen donde buscaremos todos los nodos de tipo IDENTIFIER:

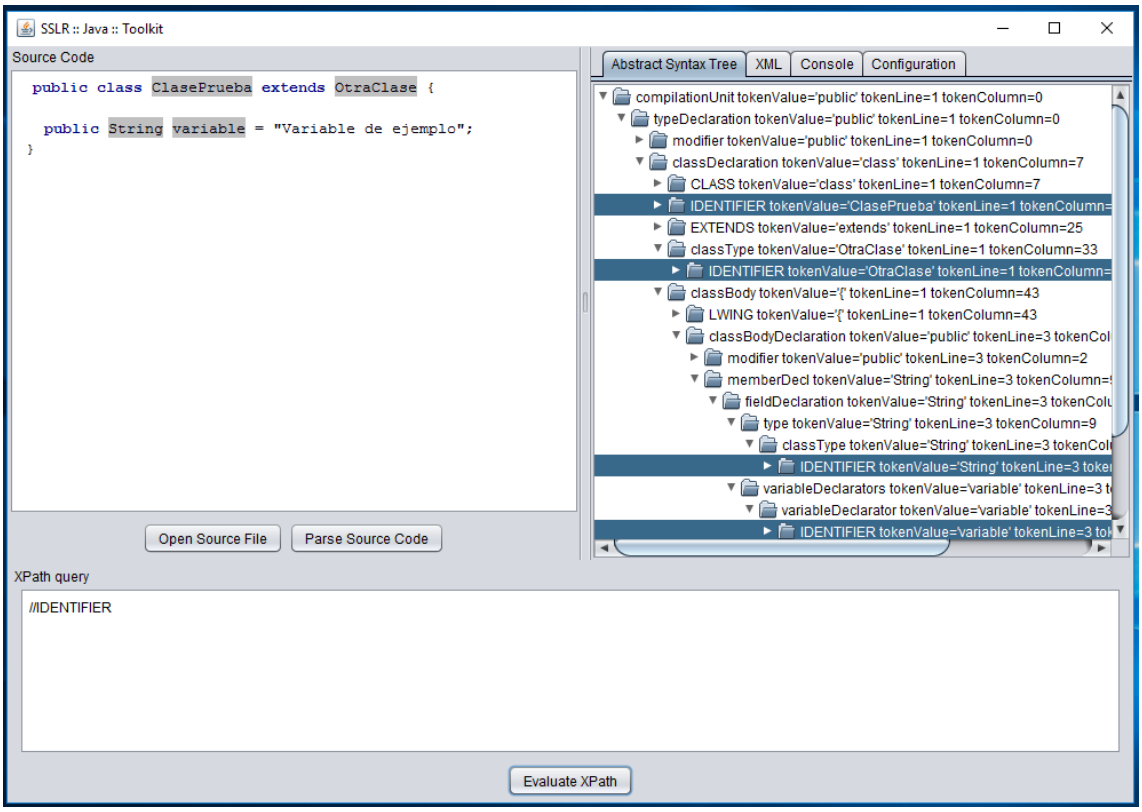


Ilustración 60. SSLRToolkit con datos de ejemplo

Esta herramienta es la que usaremos para investigar cómo es la estructura del código que estamos buscando, para poder generar así nuestra expresión de XPath.

Definición de una nueva regla

La regla que voy a implementar consistirá en proponer al usuario que use Enums en vez de constantes cuando una clase tenga varias.

Usar constantes no es incorrecto ni una mala práctica, pero en ocasiones es mejor modelar esas constantes en un enum, teniendo así un conjunto cerrado y predefinido de posibles valores [35].

Imaginemos por ejemplo que estamos modelando un personaje de videojuego con una clase Personaje. Este personaje puede ser un arquero, un caballero o un soldado. Por lo tanto, la clase Personaje tendrá un atributo llamado tipoPersonaje donde almacenará su valor. Este personaje también tendrá un género, masculino o femenino, por lo que tendremos que tener otro atributo generoPersonaje. Los posibles valores los podemos almacenar como constantes dentro de la misma clase Personaje. La implementación podría ser algo parecido a esto:

```java
public class Personaje {
	//Constantes para el tipo
	public static final String ARQUERO = "ARQUERO";
	public static final String CABALLERO = "CABALLERO";
	public static final String SOLDADO = "SOLDADO";

	//Constantes para el genero
	public static final String MASCULINO = "MASCULINO";
	public static final String FEMENINO = "FEMENINO";


	private String tipoPersonaje;
	private String generoPersonaje;

	public String getTipoPersonaje() {
		return tipoPersonaje;
	}
	public void setTipoPersonaje(String tipoPersonaje) {
		this.tipoPersonaje = tipoPersonaje;
	}
	public String getGeneroPersonaje() {
		return generoPersonaje;
	}
	public void setGeneroPersonaje(String generoPersonaje) {
		this.generoPersonaje = generoPersonaje;
	}
}
```

Alguien que use esta clase se podría equivocar de la misma manera que en el siguiente código:

```java
	public Personaje creaArqueroFemenino(){
		Personaje p = new Personaje();
		p.setTipoPersonaje(Personaje.FEMENINO);
		p.setGeneroPersonaje(Personaje.ARQUERO);

		return p;
	}
```

Si nos fijamos le está poniendo la constante de género en el tipo, y viceversa. Sin embargo, esto no provoca ningún error de compilación, por lo que no nos daremos cuenta y provocará un error cuando se ejecute e intente usar las constantes.

Si en este caso hubiéramos usado enums no tendríamos ese problema, ya que de esa manera tipificamos el género y el tipo del personaje, no aceptando valores que no sean género o tipo para cada atributo.

Por eso es interesante avisar al usuario de que, como está usando varias constantes, podría interesarle usar enums, evitando problemas como el que he explicado, y que el propio usuario valore si quiere hacerlo con constantes o con enums.

Lo primero es dejar claro qué es lo que consideramos una constante en Java.

Lo peculiar de la constante es que tiene un valor preestablecido que nunca cambiará, al menos durante la ejecución del programa. De esto podemos deducir que una constante será una declaración de una variable, que contendrá el modificador *"final"* para que nadie lo pueda modificar. Para que no se replique en cada objeto que creemos a partir de una clase, y para que no necesitemos una instancia de la clase para poder usar la constante, también llevará el modificador *"static"*.

Al llevar el modificador *final* es necesario inicializar la constante al momento de declararla; si no tendremos un error de compilación, por lo que no será necesario que comprobemos este requisito en nuestra expresión XPath.

Por lo que una constante en java tendrá este aspecto:

```
public static final int NOMBRE_CONSTANTE = 0;
```

Desarrollo de la nueva regla

Ahora que ya tenemos claro cómo se declara una constante en Java, a través del SSLR toolkit debemos investigar cómo es el árbol que representa esta declaración.

Para ello nos creamos una clase java donde definiremos la constante y la pegaremos en el programa.

```
public class ClasePrueba {
	public static final int NOMBRE_CONSTANTE = 0;
}
```

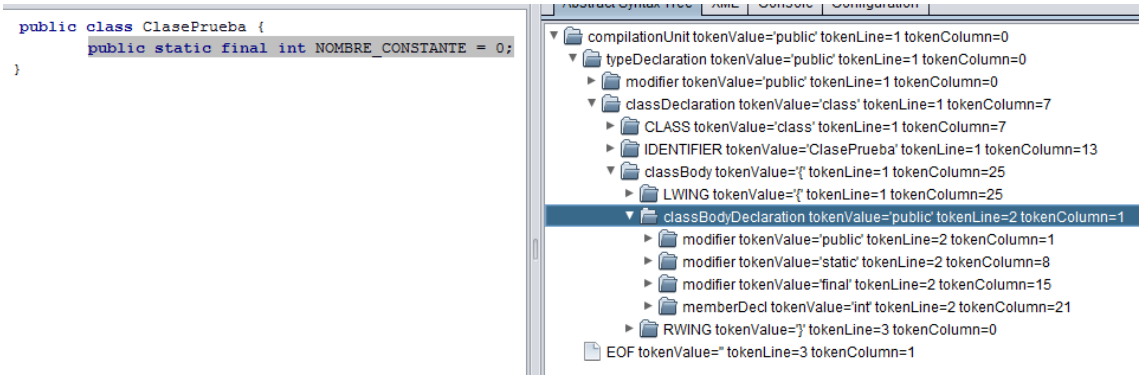


Ilustración 61. SSLRToolkit con un nodo seleccionado

Podemos ver en la imagen que el nodo que tengo seleccionado en la derecha remarca toda la línea que nos interesa. Este nodo tiene el nombre de "*classBodyDeclaration*", lo cual tiene sentido, ya que es una declaración de una constante dentro del cuerpo una clase.

También podemos ver que sus ancestros son: "*classBody*", "*classDeclaration*", "*typeDeclaration*" y "*compilationUnit*".

Como hijos de "*classBodyDeclaration*" tenemos tres nodos del tipo "*modifier*" y un nodo del tipo "*memberDecl*".

Si nos fijamos en el token de los tres nodos "*modifier*", vemos que corresponden con los modificadores *public*, *static* y *final*. Esto es importante, ya que serán algunos de los nodos que buscaremos.

Si seguimos investigando dentro del nodo "*memberDecl*" vemos que hay un nodo "*fieldDeclaration*" con tres nodos hijos: "*type*" que representa el tipo de la constante o constantes, "*variableDeclarators*" que contendrá las declaraciones de las variables, y el nodo "*SEMI*" que represente el punto y coma del final de la sentencia.

Explorando el nodo "*variableDeclarators*" vemos que contiene otro nodo "*variableDeclarator*", el cual ya contiene el nodo "*IDENTIFIER*" que es el nombre de la variable/constante el nodo "*EQU*" que representa el símbolo = y un nodo "*variableInitializer*".

Que exista un nodo "*variableDeclarators*" con un hijo "*variableDeclarator*" me hizo pensar por qué se hizo así. Esto me recordó que los dos bloques de código siguientes son válidos y equivalentes.

```
public static final int CONSTANTE_A = 0;
public static final int CONSTANTE_B = 1;
```

```
public static final int CONSTANTE_A = 0, CONSTANTE_B = 1;
```

Podemos definir una constante por sentencia, o podemos usar una sola sentencia para declarar varias constantes del mismo tipo.

En este segundo caso es cuando el nodo "*variableDeclarators*" contiene varios hijos, uno por cada declaración.

De esta observación podemos sacar otro requisito a tener en cuenta en nuestra regla, no solo debemos detectar la declaración de varias constantes en varias sentencias, sino que una sola sentencia puede tener varias constantes y tenemos que encontrarlas.

Podemos resumir lo que tiene que hacer nuestra expresión XPath de la siguiente manera:

Encontrar la existencia de más de una declaración de variable con los modificadores *static* y *final* dentro de una misma clase.

Con el siguiente código de prueba vamos a ir construyendo nuestra expresión XPath paso a paso, y la iremos comprobando en SSLR toolkit para ver que cada vez nos acercamos más a los nodos que buscamos.

```
public class Clase {

	public static final int CONSTANTE_A = 0;
	public static final int CONSTANTE_B = 1;
	private int resultado;
	private static int resultadoGlobal;
	public static final int CONSTANTE_C = 2, CONSTANTE_D = 3;


	public static final void nuevoMetodo() {
		int var1 = 0;
		int var2;
	}

	public static final void segundoMetodo(int parametro) {
		int numero = parametro;
	}


	private static final class ClasePrivada{
		public static final int CONSTANTE_M = 0;
		public static final int CONSTANTE_N = 1;
	}

	private static final class ClasePrivadaConDosConstantesYUnaDeclaracion{
		public static final int CONSTANTE_G = 50, CONSTANTE_H = 20;
	}

	private static final class ClasePrivadaQueNoIncumpleRegla{
		public static final int CONSTANTE_S = 25;

	}
}

class OtraClase{
	public static final int CONSTANTE_Z = 0;
	public static final int CONSTANTE_Y = 1;
}

class OtraClaseQueTampocoIncumpleRegla{
	public static final int CONSTANTE_Z = 100;
}
```

Encontrar la declaración de variables:

//variableDeclarator

Que tengan el modificador *static* y *final.* Estos modificadores no son ancestros, por lo que debemos navegar hasta el nodo "*classBodyDeclaration*" y comprobar que este tiene como hijos dichos nodos.

//variableDeclarator[../../../../modifier/STATIC and ../../../../modifier/FINAL]

Comprobar que existe más de un resultado con estos parámetros. Esto no se podrá ver en SSLR toolkit.

count(//variableDeclarator[../../../../modifier/STATIC and ../../../../modifier/FINAL])>1

Con esto ya tenemos una primera solución al problema. Podemos añadirla a nuestro SonarQube y probar que, efectivamente, funciona.

Para añadirla abrimos nuestro nuevo entorno de SonarQube que tiene disponible la extensión mediante XPath y nos logueamos como admin. Vamos al apartado Rules y buscamos una llamada XPath. Nos deberá aparecer una regla con la etiqueta Java como en la siguiente imagen:

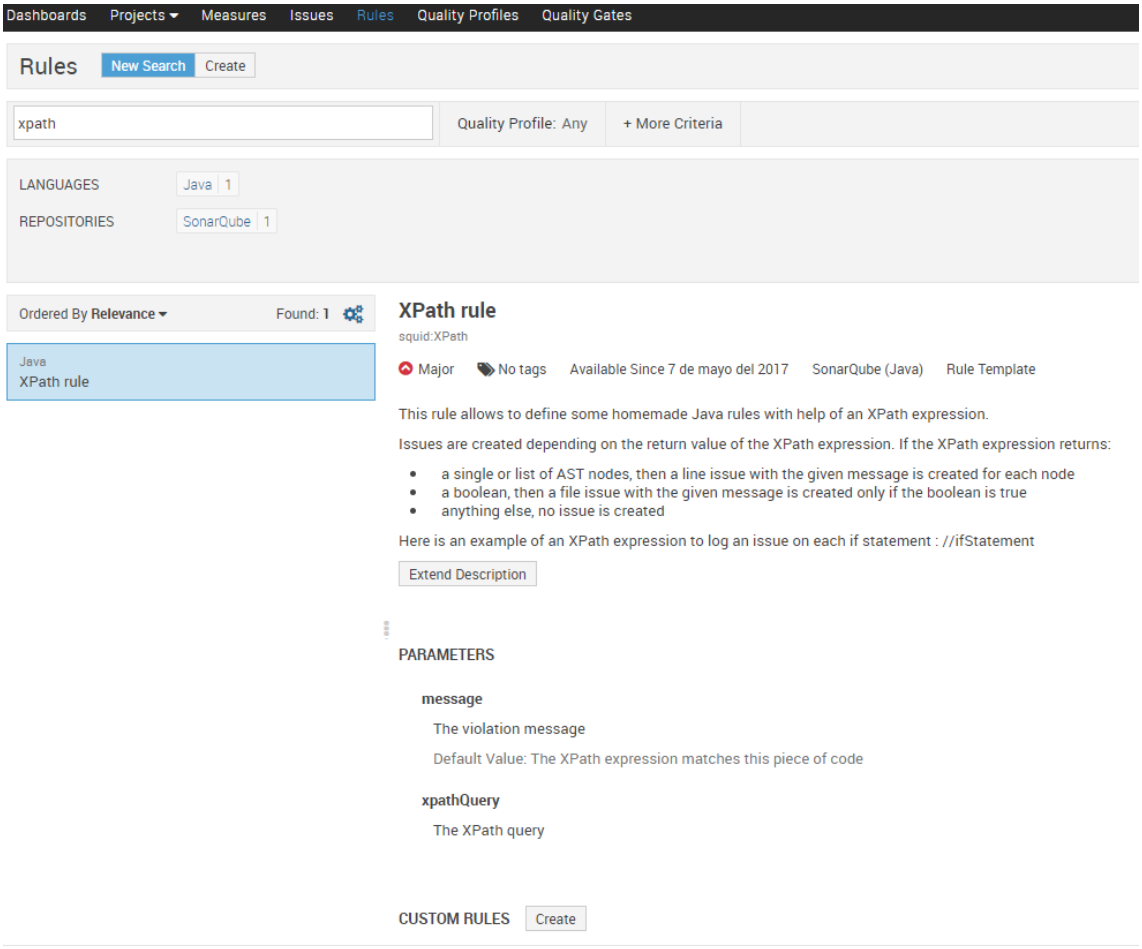


Ilustración 62. Web, detalle de regla XPath

En la parte inferior hay un apartado CUSTOM RULES y un botón Create. Pulsamos y nos sale la siguiente pantalla

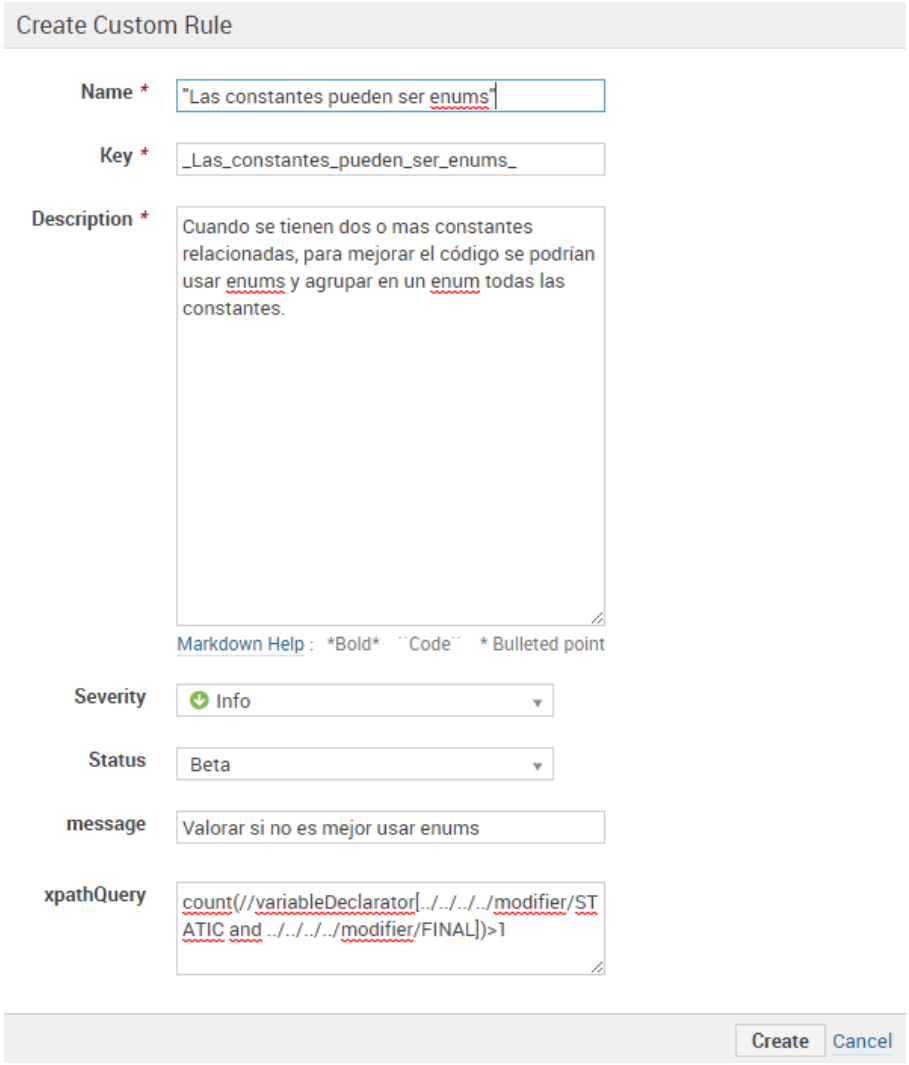


Ilustración 63. Web, formulario para incluir regla en XPath

Podemos rellenar los datos como en la imagen, incluyendo nuestra expresión XPath en el recuadro final, y pulsando en create.

Ahora en el listado de Rules aparece nuestra regla. (He puesto el nombre entre comillas para que salga de las primeras y así poder localizarla más fácilmente).

Debemos incluir esta regla en el perfil que usaremos para analizar nuestro código, y lanzar el análisis sobre el código de ejemplo que vimos antes.

Yo he creado un perfil nuevo que solo incluye mi regla y lo he puesto para que se use por defecto.

Una vez finalice el análisis, en el apartado de issues aparecerá un issue relativo a que hemos incumplido la regla.

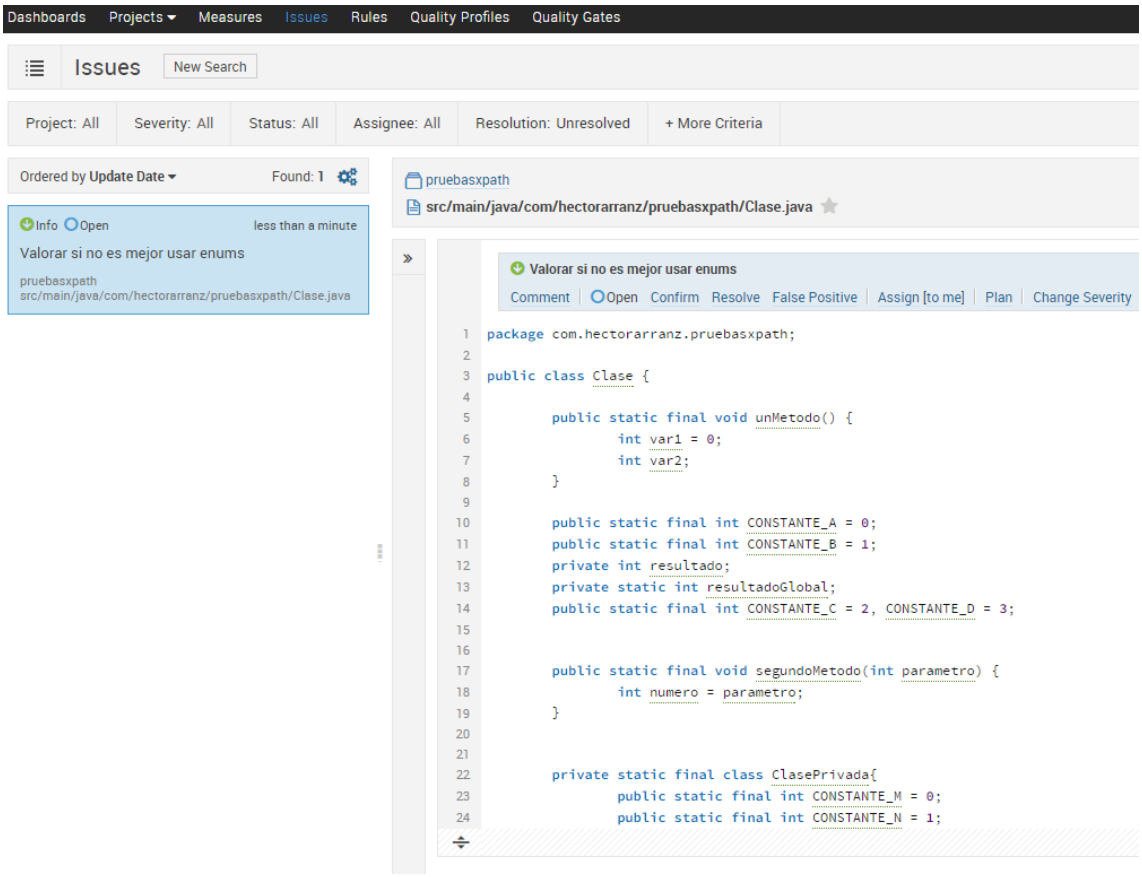


Ilustración 64. Web, issue de la nueva regla generada y el código donde se incumple

Vemos que sale nuestra regla incumplida en el fichero de ejemplo, pero no sale sobre una parte del código en concreto, si no directamente en el archivo. Esto es porque la expresión XPath, al usar el count() y la comprobación >1, devuelve un booleano en vez de un conjunto de nodos, por lo que SonarQube no sabe dónde se está incumpliendo concretamente.

Lo ideal sería que saliera un issue por cada clase que incumpla la regla, y no por cada fichero. Esto es porque sería una mala decisión de desarrollo repartir varias constantes relacionadas en diferentes clases. Si están relacionadas lo lógico es que estén en la misma clase.

En nuestro ejemplo tenemos cuatro clases que incumplen la regla en el mismo fichero y solo sale un issue.

Podemos mejorar esto haciendo que si se incumple la regla, el aviso salga sobre la clase que lo incumple. Para ello usaremos la expresión actual como el predicado de otra nueva.

```
//classDeclaration[count(.//variableDeclarator[../../../../modifier/STATIC and
../../../../modifier/FINAL])>1]
```

De esta manera seleccionamos los nodos “*classDeclaration*” que cumplen que tienen más de un nodo “*variableDeclarator*” descendiente, los cuales cumplen el requisito de ser static y final.

Hay que fijarse como en el predicado de “*classDeclaration*”, para seleccionar los nodos “*variableDeclarator*” , he cambiado // por .//

Esto es porque solo con las dos barras seleccionaban todos los nodos del archivo, por lo que todas las clases incumplían la regla al haber más de una constante en todo el documento.

Poniendo .// indicamos que los nodos “*variableDeclarator*” tienen que ser descendientes del nodo contexto, que en este caso es “*classDeclaration*”. Por lo que solo selecciona los “*variableDeclarator*” que existan en la clase.

Si volvemos a probar esto en SonarQube veremos lo siguiente:

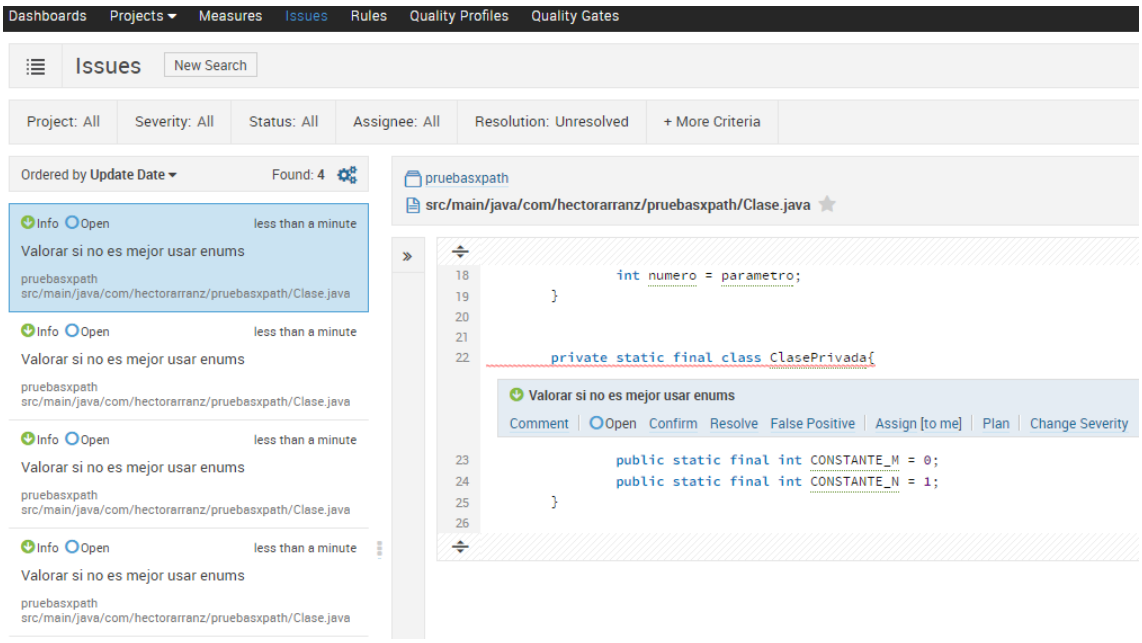


Ilustración 65. Web, issues de la nueva regla generada y el código donde se incumple 2

Ahora ya tenemos cuatro issues, uno por cada clase que incumple la regla, y vemos como en la parte derecha donde aparece el código, remarca la clase que la incumple.

Aún podemos refinar y complicar más la expresión XPath. Si en vez de generar el issue sobre la clase que contiene más de dos constantes quisiéramos generar el issue sobre la primera constante de cada clase que tenga más de dos de estas, deberemos cambiar un poco la estrategia de búsqueda.

El problema es quedarnos solo con la primera constante de una clase, teniendo en cuenta que puede haber varias clases por fichero, por lo que podría darse el caso de tener más de una primera constante en cada fichero. Si no fuera así sería muy fácil, pues obtendríamos los nodos de cada constante y nos quedaríamos con el que esté en la primera posición.

Vamos a ver paso a paso cómo se construye esta expresión para poderla ir probando en SSLR toolkit.

Primero nos quedamos con todas las declaraciones que tengan los modificadores *static* y *final.*

```
//classBodyDeclaration[modifier/STATIC and modifier/FINAL]
```

El siguiente paso es quedarnos solo con la primera de cada clase. Como todas las constantes están al mismo nivel (hijas de "*classBody*"), deducimos que son hermanas, por lo que para seleccionar la primera nos quedaremos con el nodo en el que ninguno de sus anteriores hermanos del tipo "*classBodyDeclaration*" tiene los modificadores *static* y *final.* Para ello usaremos el eje *preceding-sibling::* y la función *not()*

```
//classBodyDeclaration[modifier/STATIC and modifier/FINAL and not(preceding-
sibling::classBodyDeclaration[modifier/STATIC and modifier/FINAL]) ]
```

Vemos que en este caso no nos basta solo con indicar que tiene los modificadores static y final, ya que también nos selecciona los métodos y clases que tienen dichos modificadores y se encuentran en el cuerpo de la clase.

Debemos añadir que el "*classBodyDeclaration*" tenga descendientes del tipo "*variableDeclarator*". Pero no nos basta con que sean solo descendientes, ya que el primer método también tiene este tipo de nodo como descendientes. Incluiremos toda la ruta de una constante hasta el nodo "*variableDeclarator*"

```
//classBodyDeclaration[modifier/STATIC and modifier/FINAL and
memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator and not(preceding-
sibling::classBodyDeclaration[modifier/STATIC and modifier/FINAL and
memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator]) ]
```

Ahora que ya tenemos seleccionadas todas las primeras constantes de cada clase, nos quedaremos con las que pertenecen a una clase que contiene más de una constante. Usaremos de la primera regla que creamos la parte donde contamos cuantas constantes hay. Tenemos que contar cuantas "*variableDeclarator*" con Static y Final hay dentro de la clase.

Teniendo en cuenta que el contexto es "*classBodyDeclaration*", debemos subir un nivel hasta "*classDeclaration*" y de ahí seleccionar todos los nodos descendientes del tipo "*variableDeclarator*, comprobando si contienen los modificadores que buscamos.

```
//classBodyDeclaration[modifier/STATIC and modifier/FINAL and
memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator and not(preceding-
sibling::classBodyDeclaration[modifier/STATIC and modifier/FINAL and
memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator]) and
count(..//variableDeclarator[../../../../modifier/STATIC and ../../../../modifier/FINAL])>1 ]
```

Podemos comprobar que si cambiamos la forma de contar las constantes buscando los "*classBodyDeclaration*" con static y final, en vez de los "*variableDeclarator*" como hacemos ahora, las clases que tengan declaradas todas las constantes en una única sentencia no incumplen la regla. Podemos usar la siguiente expresión que es como la anterior, solo que cambiando la parte de dentro del count()

//classBodyDeclaration[modifier/STATIC and modifier/FINAL and memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator and not(preceding-sibling::classBodyDeclaration[modifier/STATIC and modifier/FINAL and memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator]) and count(..//classBodyDeclaration[modifier/STATIC and modifier/FINAL and memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator])>1]

Vemos que en la clase de ejemplo hay una línea que no se marca cuando sí que incumple la norma.

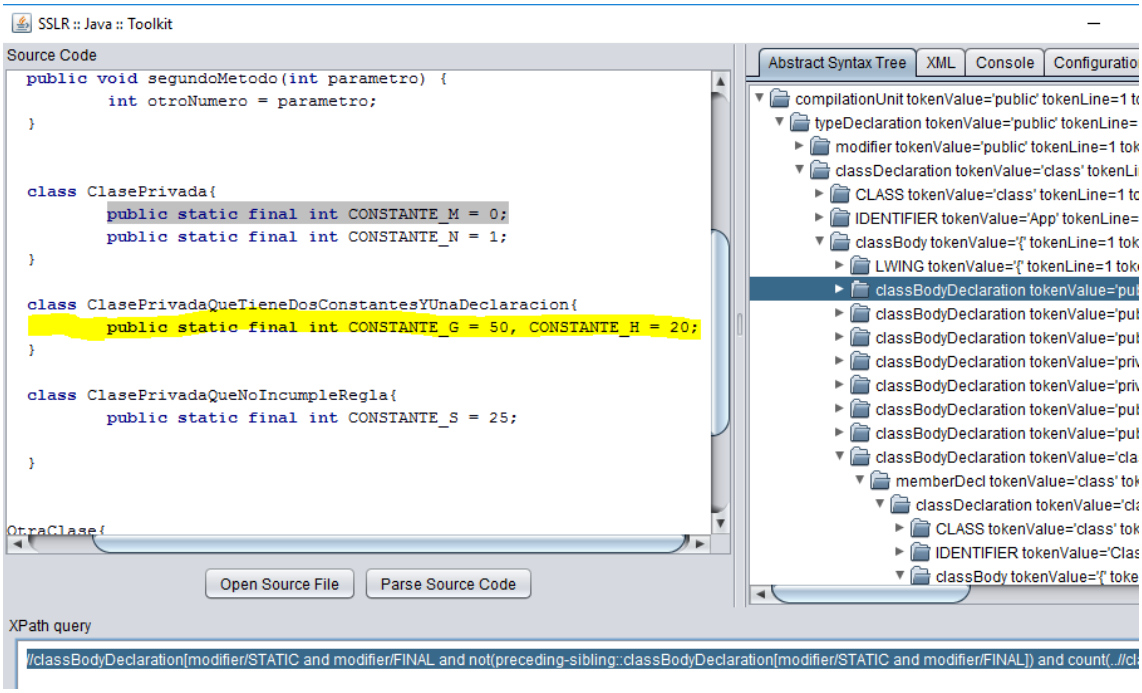


Ilustración 66. SSLRToolkit con código que debería estar remarcado

(La línea subrayada en amarillo debería estar seleccionada).

Por esta razón la expresión que cumple con lo que queremos es la siguiente:

//classBodyDeclaration[modifier/STATIC and modifier/FINAL and memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator and not(preceding-sibling::classBodyDeclaration[modifier/STATIC and modifier/FINAL and memberDecl/fieldDeclaration/variableDeclarators/variableDeclarator]) and count(..//variableDeclarator[../../../../modifier/STATIC and ../../../../modifier/FINAL])>1]

Si la probamos en SonarQube obtendremos lo siguiente

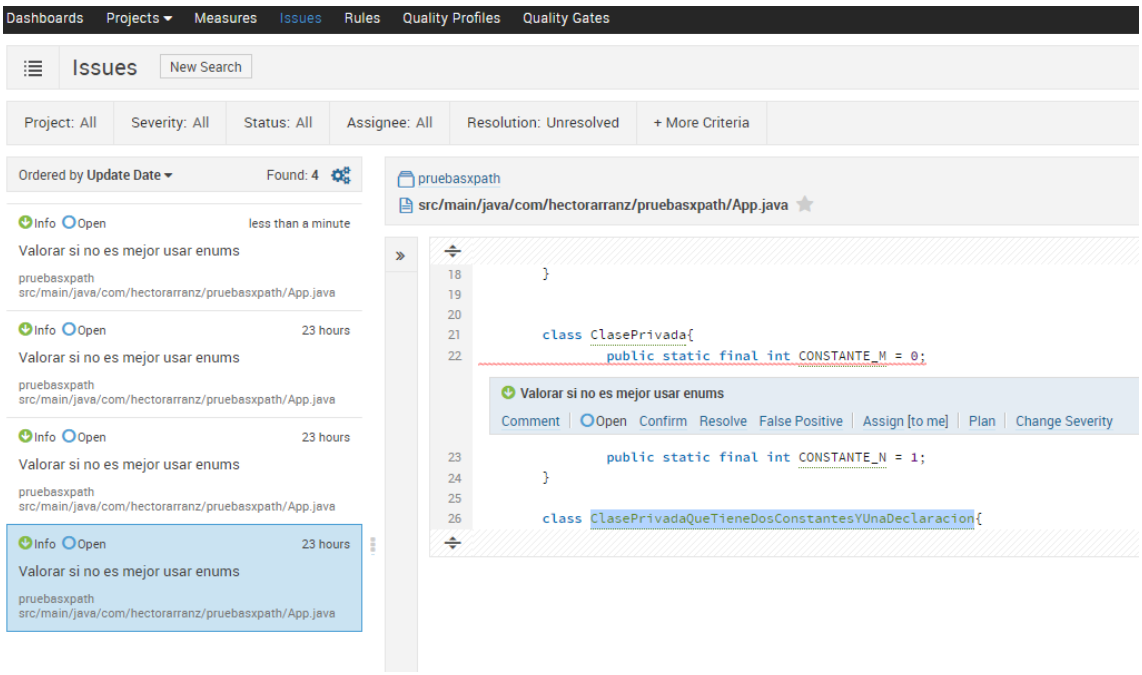


Ilustración 67. Web, issue de la nueva regla generada y el código donde se incumple 3

Ahora ya nos marca el error solo en la primera constante de cada clase.

Solo nos queda un detalle bastante importante que después de varias horas no he conseguido solucionar.

Si una clase tiene dos constantes, una del tipo int y otra de tipo String, lo más lógico es que no estén relacionadas. Para ello deberían ser del mismo tipo. Por lo que en un caso como este no debería crearse issue, aunque existan dos constantes en la misma clase. Solo debería incumplirse la regla si hay dos o más constantes del mismo tipo en la clase.

No he sido capaz de hacer esta comprobación en XPath, no sé si porque es demasiado compleja o porque no hay manera de realizar esto con XPath.

10. NUEVAS REGLAS USANDO JAVA

Estudio de la extensibilidad mediante Java

Para crear nuevas reglas de SonarQube para Java la mejor forma, y actualmente la única soportada [36], es mediante Java. Para ello tenemos que crear un proyecto que se empaquete en un JAR, y lo instalaremos como un plugin más de SonarQube. Automáticamente se añadirán las nuevas reglas que hemos programado en la lista de reglas de Java.

Para ello, los desarrolladores de SonarQube han dejado en GitHub un repositorio de un proyecto con la estructura preparada para que podamos añadir nuevas reglas, y contiene varias de ejemplo. El repositorio donde podemos descargarlo es el siguiente:

https://github.com/SonarSource/sonar-custom-rules-examples/tree/master/java-custom-rules

Lo descargamos y lo importamos en nuestro IDE. En mi caso usaré Eclipse ya que es el que mejor conozco.

Veamos varias cosas que debemos saber de esta plantilla [37]. Abrimos el POM y nos fijamos en las siguientes líneas:

```
<groupId>org.sonarsource.samples</groupId>
<artifactId>java-custom-rules</artifactId>
<version>1.0-SNAPSHOT</version>

<name>SonarQube Java Custom Rules Example</name>
<description>Java Custom Rules Example for SonarQube</description>
<inceptionYear>2016</inceptionYear>

<properties>
      <sonar.version>6.3</sonar.version>
      <java.plugin.version>4.7.1.9272</java.plugin.version>
</properties>
```

En propiedades tenemos `<sonar.version>` donde indicaremos la versión mínima de SonarQube que nuestro plugin soportará. Yo estoy usando la **6.3**, por lo que no tendré problemas.

En `<java.plugin.version>` indicamos la versión mínima requerida del plugin SonarJava para que funcione nuestro plugin. Yo tengo la **4.10.0.10260**, por lo que tampoco será ningún problema.

En el resto de campos deberemos modificarlos para que se adapten a nuestro proyecto. El groupID, artifactID, la versión, el nombre, la descripción…

Si seguimos las recomendaciones de los desarrolladores, una regla consta de tres partes:

- Un fichero Java en /src/main/java, que contendrá la implementación de la regla que estamos desarrollando.
- Un test de JUnit en /src/test/java, con el que comprobaremos que nuestra regla funciona.

- Un fichero java en /src/test/files, que contendrá uno o varios ejemplos de código que incumplen la regla y otras partes de código que si que la cumplen. Este fichero es el que usara JUnit para comprobar que todo funciona correctamente.

Para ver cómo se crea una regla haré un ejemplo muy sencillo. Una regla que nos diga que el nombre de un método no puede tener más de 10 caracteres.

También, por recomendación de SonarQube, lo ideal es seguir un desarrollo TDD, por lo que empezaremos con los test.

Lo primero será escribir un código de ejemplo con algún método que cumpla la regla y otro que no. Este código Java no es necesario que sea compilable, el único requisito es que sea sintácticamente correcto para que se pueda generar un árbol sintáctico muy similar al AST que vimos en XPath.

```
public class EjemploRule {

	public void nombreMeto(String parametro1, Integer parametro2){

	}

	public String nombreMeto2(){ // Noncompliant
		return null;
	}
}
```

Vemos que tenemos dos métodos, uno con nombre "*nombreMeto*" que tiene justo 10 caracteres y por lo tanto cumple la regla, y otro con nombre "*nombreMeto2*" con 11 caracteres y que incumple la regla. He usado los valores límite para que el test sea más fiable. Podría haber usado un método con cinco caracteres y otro con quince, pero si a la hora de desarrollar la regla me equivoco e indico que tiene que ser estrictamente menor que 10 en vez de menor o igual, con ese test no veríamos el fallo y nos diría que está todo correcto.

Podemos apreciar que en la línea de declaración del segundo método he añadido el comentario `//Noncompliant` al final de la línea. Esto es muy importante, ya que es la forma de indicarle a JUnit en qué línea tiene que generar un error esta regla porque se está incumpliendo. Si se genera un issue en otra línea que no tenga dicho comentario, o en la línea que sí tenga el comentario no se crea ningún issue el test fallará.

Este código lo almacenaremos en un archivo en /src/test/files y no es necesario que el fichero se llame igual que la clase, ya que no hay que compilarlo.

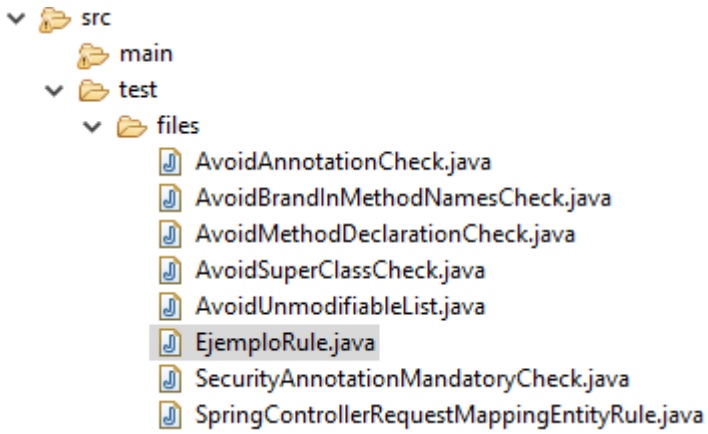


Ilustración 68. Nuevo fichero de pruebas dentro del proyecto

Ahora que ya tenemos el código donde probar nuestra regla, tenemos que crear el test que hará que pruebe nuestra implementación de la regla con ese código.

Este test hará referencia a la clase que implementa la regla, por lo que debemos crear un stub donde, posteriormente, desarrollaremos nuestra regla. De esta manera haremos que el test compile y, siguiendo la metodología TDD, comprobaremos que el test primeramente falla.

Un stub no es más que una clase sin implementación. La crearemos en /src/main/java y en mi caso la llamaré *EjemploRule.java*. Por convención, las clases donde se implementan reglas de SonarQube suelen acabar en Rule o Check.

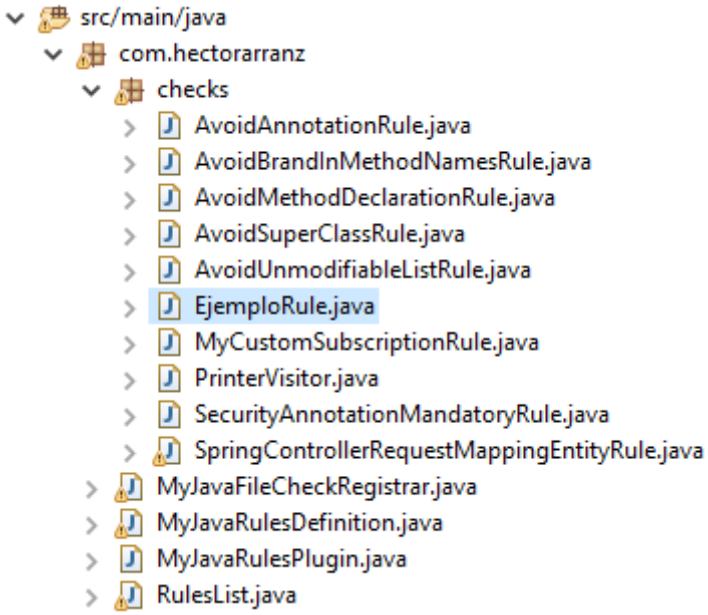


Ilustración 69. Nueva clase creada dentro del proyecto

Esta clase deberá extender de *IssuableSubscriptionVisitor* y sobreescribir el método *nodesToVisit()*. Ya veremos más adelante para qué sirve esa clase y ese método. También debemos poner una anotación a la clase: `@Rule(key = "EjemploRule")`

Nuestro stub tendrá el siguiente código:

```
package com.hectorarranz.checks;

import java.util.List;

import org.sonar.plugins.java.api.IssuableSubscriptionVisitor;
import org.sonar.plugins.java.api.tree.Tree.Kind;

import com.google.common.collect.ImmutableList;

@Rule(key = "EjemploRule")
public class EjemploRule extends IssuableSubscriptionVisitor {

	@Override
	public List<Kind> nodesToVisit() {
		return ImmutableList.of();
	}

}
```

Ya podemos crear nuestro test JUnit. En la carpeta /src/test/java creamos la clase *EjemploRuleTest.java* dentro del paquete que corresponda. Por convección las clases de test suelen nombrarse con el nombre de la clase que se prueba y acabado en Test.

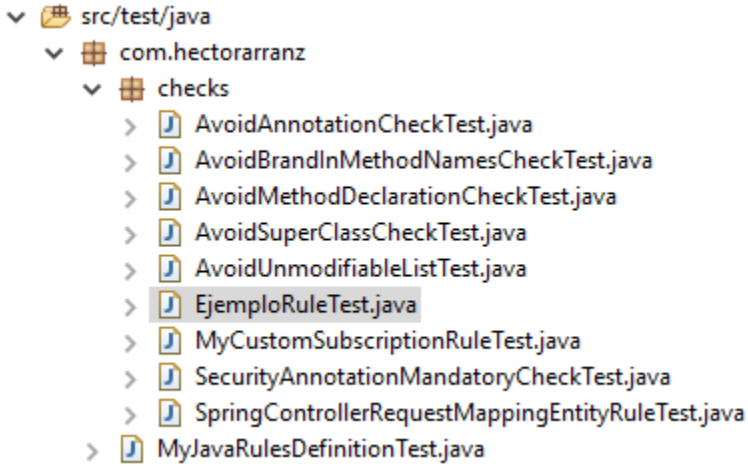


Ilustración 70. Nuevo test creado en el proyecto

Usamos JUnit 4, por lo que el test no hace falta que extienda de ninguna clase ni que los métodos tengan un nombre especial. Solo nos hace falta un método al que añadiremos la anotación `@Test`

Para reglas más complejas, es posible que nos haga falta algún método o test más o ciertas inicializaciones para que todo funcione, pero no es lo habitual.

La implementación del método de test es muy simple. Usando el método *verify()* de la clase *JavaCheckVerifier* que pertenece al API de prueba de reglas que proporciona el plugin de Java para SonarQube, le pasamos un String con la ruta donde se encuentra el fichero con el código de prueba, y una instancia de la clase que implementa nuestra regla.

La clase *JavaCheckVerifier* proporciona métodos útiles para validar implementaciones de reglas, lo que nos permite abstraer totalmente todos los mecanismos relacionados con la inicialización del analizador.

Nuestra clase de test queda así:

```
package com.hectorarranz.checks;

import org.junit.Test;
import org.sonar.java.checks.verifier.JavaCheckVerifier;

public class EjemploRuleTest {

	@Test
	public void test() {
		JavaCheckVerifier.verify("src/test/files/EjemploRule.java", new
EjemploRule());
	}

}
```

Para cumplir el primer paso de un ciclo TDD debemos comprobar que los test fallan.

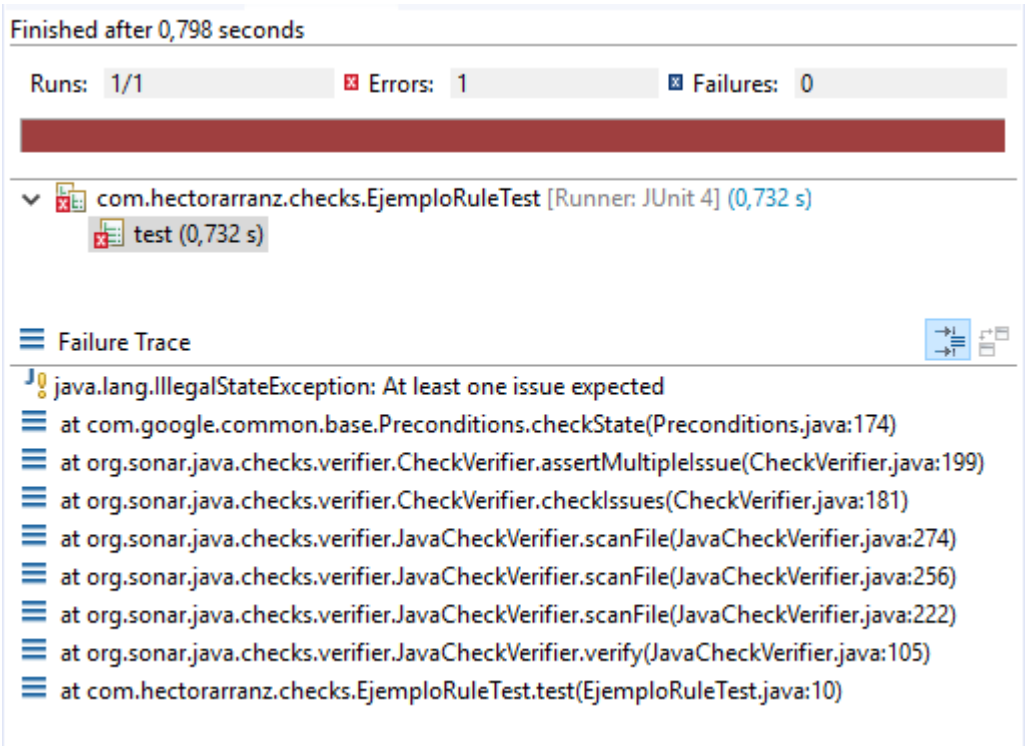


Ilustración 71. Test en rojo

Ahora que ya tenemos las test listos y fallando, debemos pasar a desarrollar el stub que generamos anteriormente.

Como comentaba anteriormente, el plugin de Java convierte el código en una estructura equivalente llamada árbol sintáctico (Syntax Tree). Cada construcción del lenguaje puede ser representada por un tipo específico de árbol sintáctico. Cada una de estas construcciones está asociada con un Kind específico, así como una interfaz que describe explícitamente todas sus particularidades. Por ejemplo, la declaración de un método (también existen árboles sintácticos para las invocaciones de métodos y para la referencia de métodos) se usará el tipo `Kind.METHOD` y la interfaz será *MethodTree*.

También vimos que la clase donde implementaremos nuestra regla extendía de *IssuableSubscriptionVisitor*. Esta clase define la estrategia que se usará cuando se analice un fichero. Como su nombre indica usa un mecanismo de subscripción. Nos subscribimos a los Kind que necesitemos investigar, y cuando el analizador encuentre alguno nos invocara pasándonos el árbol del Kind que ha encontrado y al cual estamos subscritos.

Para subscribirnos a los Kind que queramos tenemos el método que hemos sobreescrito: *nodesToVisit()*. Este método devuelve una lista de Kind que serán a los que nos interesan.

En nuestro caso como solo vamos a comprobar los nombres de los métodos cuando se declaran, nos tendremos que subscribir a `Kind.METHOD`.

El método quedará de la siguiente manera:

```
@Override
public List<Kind> nodesToVisit() {
	return ImmutableList.of(Kind.METHOD);
}
```

Si nos interesaran más tipos solo tendríamos que añadir su Kind separado por comas.

Ahora que ya nos hemos subscrito a lo que nos interesa, el plugin nos invocará cada vez que encuentre una de las construcciones que hemos indicado para que hagamos las comprobaciones necesarias. Para ello necesitamos sobreescribir otro método: *visitNode(Tree tree)*

Cuando el plugin invoque este método, nos pasara un árbol sintáctico de alguno de los tipos a los que nos subscribimos anteriormente. Todos las interfaces de árboles sintácticos extienden de Tree, por lo que si nos interesan varias construcciones deberemos comprobar qué tipo hemos recibido.

En nuestro caso, como solo nos hemos subscrito al `Kind.METHOD`, sabemos que el tree que nos pasan será una instancia de alguna clase que implementa la interfaz *MethodTree*, por lo que podemos hacer downcast sin problema.

A través de este objeto que representa un método, podemos obtener el nombre de dicho método con la función *simpleName()* que nos devuelve un *IdentifierTree*, el cual es otro árbol de sintaxis que representa un identificador. Este *IdentifierTree* tiene un método *name()* que nos devuelve el String con el nombre del método en este caso. Sobre este String ya podemos comprobar su longitud y en caso de ser mayor de 10 reportar el issue.

Para reportar un issue debemos llamar al método *reportIssue()* al cual debemos pasarle un *Tree* que será donde hemos encontrado el problema, y un String indicando el problema.

Nuestra regla de ejemplo quedará así:

```
package org.sonar.samples.java.checks;

import java.util.List;

import org.sonar.check.Rule;
import org.sonar.plugins.java.api.IssuableSubscriptionVisitor;
import org.sonar.plugins.java.api.tree.MethodTree;
import org.sonar.plugins.java.api.tree.Tree;
import org.sonar.plugins.java.api.tree.Tree.Kind;

import com.google.common.collect.ImmutableList;

@Rule(key = "EjemploRule")
public class EjemploRule extends IssuableSubscriptionVisitor {

	@Override
	public List<Kind> nodesToVisit() {
		return ImmutableList.of(Kind.METHOD);
	}

	@Override
	public void visitNode(Tree tree) {
		MethodTree method = (MethodTree) tree;
		if (method.simpleName().name().length()>10) {
			reportIssue(method.simpleName(), "Nombre muy largo");
		}
	}

}
```

Si volvemos a ejecutar el test ya tenemos el OK, por lo que hemos completado la segunda etapa del ciclo del TDD. Ya tenemos nuestra regla desarrollada, y solo faltaría la etapa de refactor, aunque en este caso no es necesaria.

Más adelante veremos cómo se instala esta regla en SonarQube.

Hemos visto cómo crear reglas usando *IssuableSubscriptionVisitor*, pero no es la única clase de la que podemos extender. También existe *BaseTreeVisitor* en la cual no nos subscribimos a los tipos que nos interesan, si no que hay un método para cada tipo, los cuales deberemos sobreescribir según lo que nos interesen.

Para crear nuevas reglas tenemos disponibles dos APIs. Una API básica con la información sintáctica, que es la que he usado en el ejemplo. Otra API semántica, la cual provee cierta información sobre los nodos que no tiene que ver con la estructura del documento.

En el ejemplo hemos buscado las declaraciones de métodos y hemos accedido a su nombre. Con la API básica nos ha sido suficiente. Pero si quisiéramos saber si el método es privado, por ejemplo, deberemos de usar la API semántica. Con ella podremos consultar los modificadores, los tipos de los parámetros, el tipo de dato que retorna, etc.

Definición de nueva regla

Ya sabemos lo básico para desarrollar nuevas reglas, ahora desarrollaremos la misma regla que en la parte de XPath pero añadiendo algo que no pudimos hacer con XPath:

- Que el usuario pueda indicar a partir de qué número de constantes debe aparecer el issue en vez de tenerlo a 2 por defecto.
- Que se tenga en cuenta el tipo de constante.

Creación de los tests

Lo primero es crear un archivo de prueba con código que cumpla y que no cumpla la regla. Me basaré en el que teníamos en la parte de XPath, modificándole un poco para que tenga en cuenta la diferenciación de tipos. Como ahora el usuario podrá configurar el parámetro del número mínimo de constantes de un mismo tipo en una misma clase para que salte el issue, deberé hacer dos ficheros de prueba ya que tendremos dos test. Esto es así porque, por defecto, este parámetro tendrá un valor que dejaré en dos como en XPath, pero tenemos que probar que si el usuario indica otro valor deberá funcionar correctamente.

Fichero para la prueba por defecto:

```
public class Clase {

        public static final int CONSTANTE_A = 0; // Noncompliant
        public static final int CONSTANTE_B = 1;
        private int resultado;
        private static int resultadoGlobal;
        public static final int CONSTANTE_C = 2, CONSTANTE_D = 3;
        public static final String CONSTANTE_1 = "A", CONSTANTE_2 = "B"; // Noncompliant

        public static final void nuevoMetodo() {
                int var1 = 0;
                int var2;
        }

        public static final void segundoMetodo(int parametro) {
                int numero = parametro;
        }

        private static final class ClasePrivada {
                public static final int CONSTANTE_M = 0; // Noncompliant
                public static final int CONSTANTE_N = 1;
        }

        private static final class ClasePrivadaConDIferentesTipos {
                public static final int CONSTANTE_A1 = 0;
                public static final String CONSTANTE_B1 = "B1";
        }

        private static final class ClasePrivadaConDosConstantesYUnaDeclaracion {
                public static final int CONSTANTE_G = 50, CONSTANTE_H = 20; // Noncompliant
        }

        private static final class ClasePrivadaQueNoIncumpleRegla {
                public static final int CONSTANTE_S = 25;

        }
}

class OtraClase {
        public static final int CONSTANTE_Z = 0; // Noncompliant
        public static final int CONSTANTE_Y = 1;
}

class OtraClaseCorrecta {
        public static final int CONSTANTE_Z = 0;
        public static final double CONSTANTE_Z2 = 1.0;
}

class OtraClaseQueTampocoIncumpleRegla {
        public static final int CONSTANTE_Z = 100;
}
```

Fichero de prueba para un valor mínimo establecido de 5:

```
public class Clase {

        public static final int CONSTANTE_A = 0;
        public static final int CONSTANTE_B = 1;
        private int resultado;
        private static int resultadoGlobal;
        public static final int CONSTANTE_C = 2, CONSTANTE_D = 3;
        public static final String C_1 = "A", C_2 = "B", C_3 = "C", C_4 = "D"; // Noncompliant
        public static final String C_5 = "E";

        public static final void nuevoMetodo() {
                int var1 = 0;
                int var2, var3, var4, var5;
        }

        private static final class ClasePrivada {
                public static final int CONSTANTE_M = 0; // Noncompliant
                public static final int CONSTANTE_N = 1;
                public static final int CONSTANTE_O = 2, CONSTANTE_P = 3;
                public static final int CONSTANTE_Q = 4;
        }

        private static final class ClasePrivadaConDIferentesTipos {
                public static final int CONSTANTE_A1 = 0, CONSTANTE_A2 = 5;
                public static final int CONSTANTE_A3 = 15, CONSTANTE_A4 = 25;
                public static final String CONSTANTE_B1 = "B1";
        }
}

class OtraClase {
        public static final int CONSTANTE_Z = 0; // Noncompliant
        public static final int CONSTANTE_Y = 1;
        public static final int CONSTANTE_X = 2;
        public static final int CONSTANTE_W = 3;
        public static final double CONSTANTE_A1 = 1.0; // Noncompliant
        public static final double CONSTANTE_A2 = 1.1;
        public static final double CONSTANTE_A3 = 1.2;
        public static final double CONSTANTE_A4 = 1.3;
        public static final double CONSTANTE_A5 = 1.4;
        public static final int CONSTANTE_V = 4;
}

class OtraClaseCorrecta {
        public static final int CONSTANTE_Z = 1;
        public static final int CONSTANTE_Y = 5;
        public static final int CONSTANTE_X = 25;
        public static final double CONSTANTE_Z2 = 1.0;
        public static final double CONSTANTE_Z3 = 1.1;
}
```

Para poder general los test de JUnit crearemos primeramente el stub. Aparte de extender de *IssuableSubscriptionVisitor* debemos añadir una constante que tendrá el valor por defecto, una variable global donde se almacenará el valor que tendrá, y una anotación sobre esta variable que hará que SonarQube ponga el valor que indique el usuario en esa variable.

```java
package com.hectorarranz.checks;

import java.util.List;

import org.sonar.check.Rule;
import org.sonar.check.RuleProperty;
import org.sonar.plugins.java.api.IssuableSubscriptionVisitor;
import org.sonar.plugins.java.api.tree.Tree.Kind;

import com.google.common.collect.ImmutableList;

@Rule(key = "ConstantsAsEnumsRule")
public class ConstantsAsEnumsRule extends IssuableSubscriptionVisitor {

	private static final int DEFAULT_VALUE = 2;

	@RuleProperty(
			defaultValue = "" + DEFAULT_VALUE,
			description = "Mínimo número de constantes en una clase
para que recomiende usar enum")
	protected int minimo = DEFAULT_VALUE;

	...

}
```

Ya solo queda crear la clase donde incluiremos nuestros test. En este caso debemos tener dos métodos test, ya que tenemos que probar que con el valor por defecto funciona, y otro en el que le establezcamos otro valor al parámetro.

```java
package com.hectorarranz.checks;

import org.junit.Test;
import org.sonar.java.checks.verifier.JavaCheckVerifier;

public class ConstantsAsEnumRuleTest {

	@Test
	public void testDefaultValue() {

	JavaCheckVerifier.verify("src/test/files/ConstantsAsEnumRuleDefault.java", new
ConstantsAsEnumRule());
	}

	@Test
	public void testSetingValue() {
		ConstantsAsEnumRule rule = new ConstantsAsEnumRule();
		rule.minimoNoAceptado = 5;
		JavaCheckVerifier.verify("src/test/files/ConstantsAsEnumRule.java",
rule);
	}
}
```

Una vez comprobado que los test fallan procedemos a implementar la regla.

Implementacion de la regla

El algoritmo es sencillo. Nos subscribimos a `Kind.CLASS` y cada vez que recibamos un *`Tree`* de class obtendremos sus miembros con el método members(). Esto nos da una lista de *`Tree`* que pueden ser de diferentes tipos (`Kind.VARIABLE`, `Kind.METHOD`, `Kind.CLASS`, etc). Nos quedaremos solo con los de tipo Variable, ya que una constante será una variable con ciertos modificadores. Esta lista solo tendrá construcciones a nivel de clase, por lo que si nuestra clase tiene otra clase interna y esta tiene constantes, en nuestra lista solo tendremos el *`Tree`* de la clase interna, pero no los de las variables pertenecientes a la clase interna.

De esta manera podemos estar seguros que solo tenemos en cuenta las constantes que pertenecen a una clase y no interfieren las constantes declaradas en clases internas.

Una vez tenemos una variable, comprobamos si tiene los modificadores *Static* y *Final*. Para ello se usa el API semántico.

Si es una constante comprobamos su tipo e incrementamos en uno el contador de este tipo. También almacenamos la primera ocurrencia de cada tipo, para en caso de que se incumpla la norma poder lanzar el issue sobre esa ocurrencia.

Finalmente recorremos todos los contadores de cada tipo, y si igualan o superan el valor establecido generamos el issue.

Todo esto se entiende mejor leyendo el código:

```java
@Rule(key = "ConstantsAsEnumRule")
public class ConstantsAsEnumRule extends IssuableSubscriptionVisitor {

	private static final int DEFAULT_VALUE = 2;

	@RuleProperty(defaultValue = ""
			+ DEFAULT_VALUE, description = "Mínimo número de constantes en
una clase para que se recomiende usar Enum")
	protected int minimo = DEFAULT_VALUE;

	@Override
	public List<Kind> nodesToVisit() {
		return ImmutableList.of(Kind.CLASS);
	}

	@Override
	public void visitNode(Tree tree) {
		Map<Type, Integer> contadoresPorTipo = new Hashtable<>();
		Map<Type, Tree> primeraOcurrenciaDelTipo = new Hashtable<>();

		ClassTree clase = (ClassTree) tree;
		List<Tree> miembrosDeLaClase = clase.members();

		for (Tree miembro : miembrosDeLaClase) {
			if (miembro.is(Kind.VARIABLE)) {
				VariableTree variable = (VariableTree) miembro;
				if (esUnaConstante(variable)) {
					Type tipo = variable.symbol().type();
					if (contadoresPorTipo.containsKey(tipo)) {
						Integer ocurrenciasActuales =
contadoresPorTipo.get(tipo);
						contadoresPorTipo.put(tipo,
ocurrenciasActuales.intValue() + 1);
					} else {
						contadoresPorTipo.put(tipo, 1);
						primeraOcurrenciaDelTipo.put(tipo,
variable);
					}
				}
			}
		}

		Set<Type> tipos = contadoresPorTipo.keySet();
		for (Type tipo : tipos) {
			Integer ocurrencias = contadoresPorTipo.get(tipo);
			if (ocurrencias.intValue() >= minimo) {
				Tree primeraOcurrencia =
primeraOcurrenciaDelTipo.get(tipo);

				reportIssue(primeraOcurrencia, "Se estan usando
demasiadoas constantes del tipo " + tipo.name()
						+ ". Puede que sea mejor agruparlas en un
Enum");
			}
		}
	}

	private boolean esUnaConstante(VariableTree variable) {
		return variable.symbol().isStatic() && variable.symbol().isFinal();
	}

}
```

Hay otra forma de resolver el problema. Nos tendríamos que subscribir a los `Kind.VARIABLE`, y cada vez que recibamos una comprobar si tiene los modificadores necesarios para ser constante. Si es así ya tenemos una constante y es más sencillo que investigar los miembros de una clase.

Pero por otro lado se complica la parte de los contadores, ya que no solo hay que llevar un contador por cada tipo de constante porque las constantes pueden pertenecer a cualquier clase.

Con el *Tree* de la constante deberíamos acceder al padre, que sería un *Tree* de clase, y según cuál sea actualizar unos contadores u otros.

Esta forma me ha parecido más compleja y más difícil de entender, pero la he probado y funciona correctamente.

Ya tenemos la implementación; si ejecutamos los test deberían pasar sin problemas, por lo que tenemos nuestra regla finalizada.

Solo nos queda añadirla a SonarQube. Pero antes podemos añadirle más metadatos para que se puedan ver en SonarQube.

En la anotación `@Rule` de la clase podemos añadir más información:

```
@Rule(
      key = "ConstantsAsEnumRule",
      name = "Constants As Enums",
      description = "Si se tienen varias constantes equivalentes puede que
sea mejor agruparlas en un Enum",
      priority = Priority.INFO,
      tags = {"code-smell"})
public class ConstantsAsEnumRule extends IssuableSubscriptionVisitor {
```

Solo nos queda registrar nuestra regla en el plugin. En /src/main/java hay una clase llamada `RuleList.java`. Si la abrimos la clase hay un método llamado *getJavaChecks()* donde se añaden todas las clases que implementan las reglas en una lista. Solamente debemos añadir la nuestra:

```
public static List<Class<? extends JavaCheck>> getJavaChecks() {
  return ImmutableList.<Class<? extends JavaCheck>>builder()
    .add(SpringControllerRequestMappingEntityRule.class)
    .add(AvoidAnnotationRule.class)
    .add(AvoidBrandInMethodNamesRule.class)
    .add(AvoidMethodDeclarationRule.class)
    .add(AvoidSuperClassRule.class)
    .add(AvoidUnmodifiableListRule.class)
    .add(MyCustomSubscriptionRule.class)
    .add(SecurityAnnotationMandatoryRule.class)
    .add(ConstantsAsEnumRule.class)
    .build();
}
```

Ilustración 72. Método donde subscribir la nueva regla

Podríamos quitar el resto de la lista para que el plugin que vamos a crear solo incluya nuestra regla y no las que vienen de ejemplo.

Una vez añadida solo nos queda ejecutar el comando **mvn clean install** para que compile todo y genere un JAR con nuestro plugin. Si todo ha ido bien, el JAR tiene que estar en la carpeta Target del proyecto.

Con SonarQube parado tenemos que copiar el JAR que hemos generado en {SONAR_HOME}/extensions/plugins y volver a arrancar SonarQube.

Vamos al listado de reglas y filtramos por nombre para poder encontrar nuestra regla

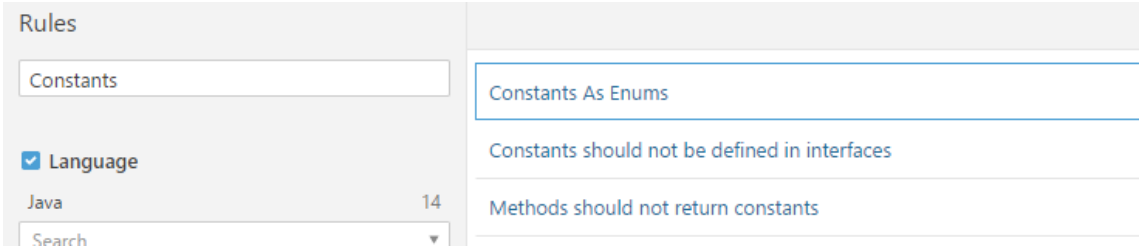


Ilustración 73. Web, nueva regla generada

Y si pulsamos en la regla podremos ver los detalles

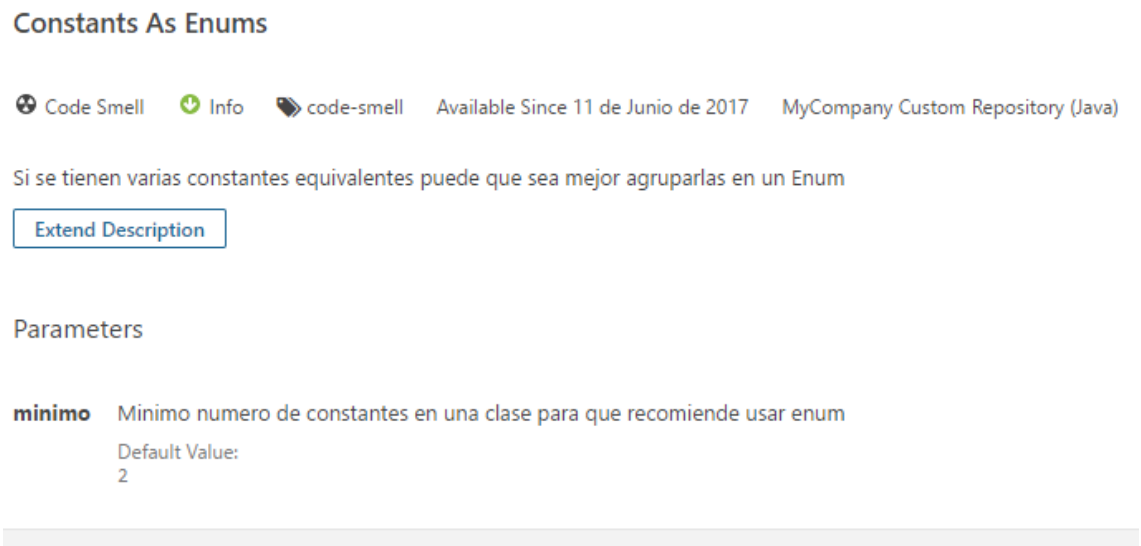


Ilustración 74. Web, detalle de nueva regla generada

Vemos como los metadatos que hemos añadido en la anotación de la clase aparecen en esta pantalla. También aparecen los datos relativos al parámetro que se puede configurar. Hay que tener cuidado porque el nombre que le demos a la variable en java es el que aparece en esta pantalla.

Al incluir la regla en un perfil nos aparece la siguiente pantalla:

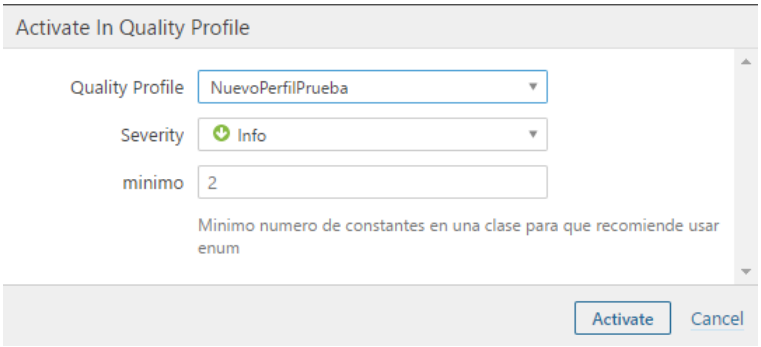


Ilustración 75. Web, formulario para incluir nueva regla en un perfil de calidad

Aquí es donde configuramos el valor del parámetro. Para cada perfil que incluya la regla puede tener un valor diferente.

Si ahora analizamos los ficheros de prueba con el perfil que incluye la regla (en mi caso he creado un perfil con solo esta regla), vemos como funciona perfectamente:

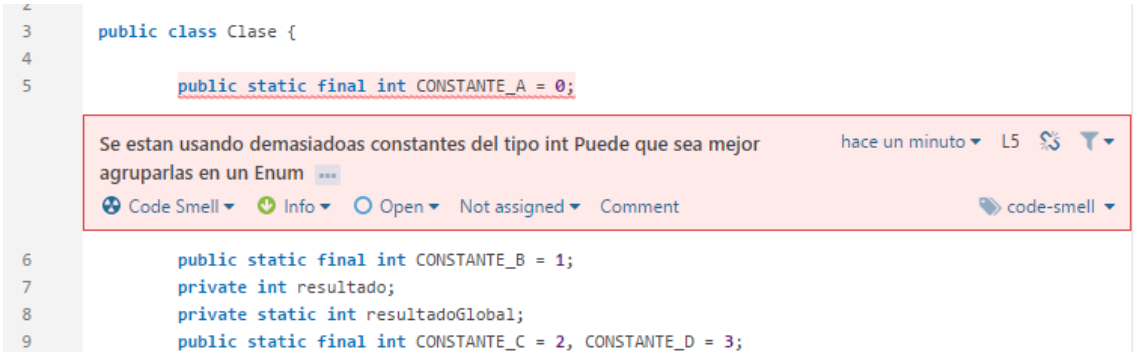


Ilustración 76. Web, issue de la nueva regla incrustado en el código donde falla

11. CONCLUSIONES

Después de meses analizando y estudiando cómo funciona y cómo se usa SonarQube, creo que es una herramienta que debería utilizarse en cada desarrollo software que se realice, al igual que sucede con los tests, que es algo imprescindible.

SonarQube te ayuda a encontrar problemas antes de que ocurran, te ayuda a que el código no se vuelva inmantenible, y después de casi ya 2 años que llevo trabajando en un proyecto en el que no existen test automáticos y mucho menos se tiene en cuenta la calidad del código, se ha convertido en una herramienta muy apreciada por mí. He pasado horas intentando entender un código, pues quien lo desarrolló no pensó que eso tendría que leerlo otro programador tiempo después. Si hubieran tenido implantado SonarQube en mi empresa, ese código no hubiera pasado desapercibido, obligando al desarrollador a hacerlo más mantenible.

Este proyecto me ha servido para aprender varias cosas sobre calidad de código, y conocer más herramientas que me pueden servir en mi día a día. No he podido investigar ni probar todo lo que tenía en mente. Tenía planificado el estudio de unos plugins de SonarQube para su integración con GitHub y Bitbucket, pensando que podría analizar proyectos almacenados en estos repositorios, pero en cuanto empecé a estudiar cómo se usaban, me di cuenta de que estaba equivocado en la funcionalidad de los mismos. No son para analizar proyectos almacenados en estas plataformas, si no para analizar pull-request que se hagan en dichos servicios.

El problema es que apenas tenía conocimiento de cómo funcionan los pull-request, por lo que el tiempo estimado para estas tareas lo superé con creces, sin sacar muchas conclusiones ni nada que funcionara.

Este TFG me ha resultado complicado poder completarlo ya que, como comentaba, llevo 2 años compaginándolo con el trabajo, que me mantiene once horas diarias fuera de casa, y es difícil dedicarle todos los días algo de tiempo después de pasar tantas horas usando un ordenador y programando. Los resultados no son tan buenos como esperaba en un principio, pero viendo todo lo que he aprendido y de lo que soy capaz de hacer si me lo propongo, me quedo satisfecho.

Mi siguiente reto es poder implantar esta herramienta en mi empresa y que se utilize de verdad.

Este proyecto tiene varias líneas futuras mucho más complejas de lo visto aquí, como por ejemplo desarrollar un plugin para poder hacer eso que creía que hacían los plugins que he estudiado: analizar proyectos almacenados en repositorios. No estoy muy seguro que sea algo viable, ya que a no ser que esté montado con maven, va a necesitar un fichero de configuración con los datos del proyecto. Y aun teniendo Maven, puede que necesite algunos datos que habría que estudiar cómo proporcionárselos.

Otra línea futura más realista es desarrollar conjuntos de nuevas reglas para SonarQube que se puedan usar en un determinado perfil de calidad ya que en este proyecto se ha aprendido como se pueden generar y se ha hecho una prueba de concepto. También sería interesante poder aplicar lo aquí aprendido y probado en la docencia.

Sin embargo, la línea futura más ambiciosa podría ser desarrollar un nuevo plugin para que SonarQube sea capaz de analizar otro lenguaje más que ahora mismo no está soportado, como por ejemplo uno que he escuchado mucho últimamente: Kotlin.

12. BIBLIOGRAFÍA

[1] milinaudara, «An Introduction to SonarQube», *Optimism*, 15-ene-2015. [En línea]. Disponible en: https://milinaudara.wordpress.com/2015/01/15/an-introduction-to-sonarqube/. [Accedido: 10-jul-2017].

[2] «Acyclic dependencies principle», *Wikipedia*, 27-ago-2015. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Acyclic_dependencies_principle&oldid=678047356. [Accedido: 11-jul-2017].

[3] A. Calero, «¿Qué es SonarQube?», *Antonio Calero*, 28-nov-2014. [En línea]. Disponible en: http://www.acalero.com/2014/11/28/que-es-sonarqube. [Accedido: 10-jul-2017].

[4] A. Enrikus, «SonarQube: instalación y configuración», *Enrikus' Blog*, 28-dic-2013. [En línea]. Disponible en: http://enrikusblog.com/sonarqube-instalacion-y-configuracion/. [Accedido: 10-jul-2017].

[5] «Analyzing with SonarQube Scanner - Scanners - Doc SonarQube». [En línea]. Disponible en: https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner. [Accedido: 10-jul-2017].

[6] «Analyzing with SonarQube Scanner for Ant - Scanners - Doc SonarQube». [En línea]. Disponible en: https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+Ant. [Accedido: 10-jul-2017].

[7] «Analyzing with SonarQube Scanner for Maven - Scanners - Doc SonarQube». [En línea]. Disponible en: https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+Maven. [Accedido: 10-jul-2017].

[8] «PMD (software)», *Wikipedia*, 29-abr-2017. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=PMD_(software)&oldid=777804052. [Accedido: 10-jul-2017]

[9] «PMD». [En línea]. Disponible en: https://pmd.github.io/. [Accedido: 10-jul-2017].

[10] «FindBugs™ - Find Bugs in Java Programs». [En línea]. Disponible en: http://findbugs.sourceforge.net/. [Accedido: 10-jul-2017].

[11] «FindBugs», *Wikipedia*, 05-oct-2016. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=FindBugs&oldid=742697797. [Accedido: 10-jul-2017]

[12] «Checkstyle», *Wikipedia*, 24-feb-2017. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Checkstyle&oldid=767229189. [Accedido: 10-jul-2017]

[13] «checkstyle: Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. By default it supports the Google Java Style Guide and Sun Code Conventions, but..». [En línea]. Disponible en: https://github.com/checkstyle/checkstyle. [Accedido: 10-jul-2017]

[14] alfonso, «¿Qué es un pull request?», *Aprende GIT*, 14-feb-2013. [En línea]. Disponible en: http://aprendegit.com/que-es-un-pull-request/. [Accedido: 10-jul-2017].

[15] «JHotDraw Start Page». [En línea]. Disponible en: http://www.jhotdraw.org/. [Accedido: 10-jul-2017].

[16] «Metric Definitions - SonarQube Documentation - Doc SonarQube». [En línea]. Disponible en: https://docs.sonarqube.org/display/SONAR/Metric+Definitions. [Accedido: 10-jul-2017].

[17] «Complejidad ciclomática ¿Nuestro código es fácil de mantener y probar?», *JoaquinOriente.com*, 15-nov-2012. [En línea]. Disponible en: http://joaquinoriente.com/2012/11/15/complejidad-ciclomatica-nuestro-codigo-es-facil-de-mantener-y-probar/. [Accedido: 10-jul-2017].

[18] «Complejidad ciclomática», *Wikipedia, la enciclopedia libre*, 19-abr-2016. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Complejidad_ciclom%C3%A1tica&oldid=90567000. [Accedido: 10-jul-2017]

[19] «What is the highest Cyclomatic Complexity of any function you maintain? And how would you go about refactoring it? - Stack Overflow». [En línea]. Disponible en: https://stackoverflow.com/questions/1364946/what-is-the-highest-cyclomatic-complexity-of-any-function-you-maintain-and-how. [Accedido: 10-jul-2017].

[20] «JFreeChart». [En línea]. Disponible en: http://www.jfree.org/jfreechart/. [Accedido: 10-jul-2017].

[21] «Maven: skip GPG sign process». [En línea]. Disponible en: http://www.michaelpollmeier.com/maven-skip-pgp-sign-process. [Accedido: 10-jul-2017].

[22] «Apache Maven GPG Plugin – Introduction». [En línea]. Disponible en: http://maven.apache.org/plugins/maven-gpg-plugin/. [Accedido: 10-jul-2017].

[23] J. Dobie, «Unit Test Code Coverage With Maven And Jacoco», *JavaWorld*, 16-ene-2012. [En línea]. Disponible en: http://www.javaworld.com/article/2074515/core-java/unit-test-code-coverage-with-maven-and-jacoco.html. [Accedido: 10-jul-2017].

[24] «Code Coverage by Unit Tests for Java Project - Plugins - Doc SonarQube». [En línea]. Disponible en: https://docs.sonarqube.org/display/PLUG/Code+Coverage+by+Unit+Tests+for+Java+Project. [Accedido: 10-jul-2017].

[25] «XPath Tutorial». [En línea]. Disponible en: https://www.w3schools.com/xml/xpath_intro.asp. [Accedido: 10-jul-2017].

[26] «XPath», *Wikipedia, la enciclopedia libre*, 22-abr-2017. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=XPath&oldid=98545994. [Accedido: 10-jul-2017]

[27] «Tutorial de XPath. Ver. 1.0». [En línea]. Disponible en: http://geneura.ugr.es/~victor/cursillos/xml/XPath/. [Accedido: 10-jul-2017].

[28] «Una guía rápida para XPath». [En línea]. Disponible en: http://gpd.sip.ucm.es/rafa/docencia/bdsi/apuntes/XPath.pdf. [Accedido: 10-jul-2017].

[29] «java - Is Xpath rule deprecated in Sonar 4.5.1 - Stack Overflow». [En línea]. Disponible en: https://stackoverflow.com/questions/34854611/is-xpath-rule-deprecated-in-sonar-4-5-1. [Accedido: 10-jul-2017].

[30] «Java XPath rules in Sonarqube 5.2 - Stack Overflow». [En línea]. Disponible en: https://stackoverflow.com/questions/33765076/java-xpath-rules-in-sonarqube-5-2. [Accedido: 10-jul-2017].

[31] «Release 6.1 Upgrade Notes - SonarQube Documentation - Doc SonarQube». [En línea]. Disponible en: https://docs.sonarqube.org/display/SONAR/Release+6.1+Upgrade+Notes. [Accedido: 10-jul-2017].

[32] «ModuleFileSystem (SonarQube 4.3 API)». [En línea]. Disponible en: http://javadocs.sonarsource.org/4.3/apidocs/org/sonar/api/scan/filesystem/ModuleFileSystem.html. [Accedido: 10-jul-2017].

[33] «Downloads | SonarQube». [En línea]. Disponible en: https://www.sonarqube.org/downloads/. [Accedido: 10-jul-2017].

[34] «Árbol de sintaxis abstracta», *Wikipedia, la enciclopedia libre*, 07-may-2017. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=%C3%81rbol_de_sintaxis_abstracta&oldid=98936913. [Accedido: 10-jul-2017]

[35] «java - Why use Enums instead of Constants? - Stack Overflow». [En línea]. Disponible en: https://stackoverflow.com/questions/11575376/why-use-enums-instead-of-constants. [Accedido: 10-jul-2017].

[36] «Adding Coding Rules - Extend - Doc SonarQube». [En línea]. Disponible en: https://docs.sonarqube.org/display/DEV/Adding+Coding+Rules. [Accedido: 10-jul-2017].

[37] «Writing Custom Java Rules 101 - Plugins - Doc SonarQube». [En línea]. Disponible en: https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101. [Accedido: 10-jul-2017].

13. CONTENIDO DEL CD

- Memoria del TFG en formato PDF.
- Carpéta código fuente:
    - Proyecto maven con el desarrollo Java.
    - Fichero de texto con el desarrollo XPath.
- Plugin para SonarQube con la regla desarrollada en Java en formato JAR.