

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

GRADO EN INGENIERÍA INFORMÁTICA



Trabajo Fin de Grado

"Estudio sobre herramientas de calidad de software"

Javier Díez Aguilar

2015

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

"Estudio sobre herramientas de calidad de software"

Autor: Javier Díez Aguilar

Director: Daniel Rodríguez García

TRIBUNAL

Presidente:

Vocal 1º:

Vocal 2º:

Calificación:

Fecha: de de

Índice

Índice	2
Índice de figuras	3
Índice de tablas	5
Resumen.....	6
Preface	6
Resumen extendido	7
1 Herramientas gráficas de calidad del software.....	8
1.1 Introducción	8
1.2 Planificación	10
1.3 Análisis de herramientas.....	11
1.3.1 Listado previo	11
1.3.2 Análisis breve de las herramientas.....	12
1.4 Análisis en profundidad de tres herramientas.....	16
1.4.1 Características comunes.....	16
1.4.2 SonarQube.....	17
1.4.3 Jira	41
1.4.4 MetricsGrimoire	54
1.5 Comparación de las herramientas	70
2 Implementación de Plug-In	73
2.1 Introducción	73
2.2 Entendiendo la arquitectura de SonarQube	74
2.3 Creación de un plugin.....	76
2.3.1 Creando el proyecto Maven	76
2.3.2 Definiendo la configuración disponible del widget.....	79
2.3.3 Describiendo las métricas	80
2.3.4 Implementando los analizadores: Clase Sensor.....	82
2.3.5 Implementando los analizadores: Clase Decorador.....	85
2.3.6 Creando el widget	86
4 Conclusiones.....	92
5 Trabajos futuros	93
6 Referencias.....	94

Índice de figuras

Figura 1. Arquitectura de SonarQube	17
Figura 2 Los siete ejes de SonarQube	18
Figura 3 Página principal de SonarQube	22
Figura 4 Dashboard por defecto	23
Figura 5 Size metrics widget.....	24
Figura 6 Events widget	24
Figura 7 Widget que muestra información general	25
Figura 8 Issues Widget	25
Figura 9 Cobertura Widget, de (http://www.adictosaltrabajo.com/)	26
Figura 10 Comentarios y líneas duplicadas	26
Figura 11 Métricas sobre la arquitectura, de (SonarQube in Action)	27
Figura 12 Métricas LCOM4 y RPC	27
Figura 13 Complejidad de las clases.....	28
Figura 14 Comportamiento de FindBugs, de (SonarQube in Action).....	31
Figura 15 Esquema SQALE.....	34
Figura 16 Widget calificación SQALE	34
Figura 17 Technical Debt Widget	35
Figura 18 Enlace a la documentación de la API.....	36
Figura 19 Métodos existentes en la API.....	37
Figura 20 Resultado de la consulta REST (I)	37
Figura 21 Resultado de la consulta REST (II)	38
Figura 22 Información exportada a excel.....	39
Figura 23 Arquitectura de Jira, de (Jira Essentials).....	42
Figura 24 Página principal de Jira	43
Figura 25 Búsqueda de cuadros de mando	43
Figura 26 Índice de proyectos existentes en el sistema.....	44
Figura 27 Detalle de un proyecto en Jira.....	44
Figura 28 Incidencias asignadas a un usuario	45
Figura 29 Pizarra SCRUM.....	46
Figura 30 Arquitectura subyacente a Jira.....	47
Figura 31 Flujo de trabajo entre MetricsGrimoire y vizGrimoire, de (http://vizgrimoire.bitergia.org/)	57
Figura 32 Dashboard de vizGrimoire para Eclipse (I), de (http://bitergia.com/)	58
Figura 33 Dashboard de vizGrimoire para Eclipse (II) , de (http://bitergia.com/)	58
Figura 34 Dashboard de vizGrimoire en detalle para Eclipse (I) , de (http://bitergia.com/)	59
Figura 35 Dashboard de vizGrimoire en detalle para Eclipse (II) , de (http://bitergia.com/)	60
Figura 36 Dashboard de vizGrimoire para openStack (I) , de (http://bitergia.com/)	60
Figura 37 Dashboard de vizGrimoire para openStack (II) , de (http://bitergia.com/)	61
Figura 38 Madurez de las herramientas	70
Figura 39 Ranking de BBDD más utilizadas, de (http://db-engines.com/en/ranking_trend)	72
Figura 40 Arquitectura SonarQube, de (SonarQube in Action).....	74
Figura 41 Creación de directorios usando Maven	76

Figura 42 Estructura de directorios inicial.....	76
Figura 43 Estructura de directorios final.....	77
Figura 44 Ficheros y directorios	77
Figura 45 Clase de entrada al plugin	79
Figura 46 Clase de definición de métricas.....	80
Figura 47 Nuestras métricas pertenecen al mismo grupo	81
Figura 48 Los componentes del constructor serán inyectados por SonarQube	82
Figura 49 Método para filtrar en qué tipos de proyectos se ejecuta nuestro sensor	83
Figura 50 Método donde se analizan y almacenan las métricas	84
Figura 51 Filtrado de ejecución de los decoradores	85
Figura 52 Método decorate	85
Figura 53 Método para utilizar otras métricas ya calculadas	86
Figura 54 Clase Java que configura el widget.....	87
Figura 55 Vista en SonarQube de los elementos definidos en nuestra clase Java	88
Figura 56 Parte de la clase Ruby on Rails	88
Figura 57 Métricas generales del proyecto.....	89
Figura 58 Métricas de tamaño del código.....	89
Figura 59 Gráfico generado mediante google chart	90
Figura 60 Gráfico circular y diagrama de Venn	90
Figura 61 Representación del widget en el dashboard de SonarQube.....	91

Índice de tablas

Tabla 1 Resumen de las herramientas	11
Tabla 2 Plugin disponibles y métricas que soportan	21
Tabla 3 Resumen CheckStyle.....	29
Tabla 4 Resumen PMD	30
Tabla 5 Resumen FindBugs.....	31
Tabla 6 Comparación herramientas de cobertura de test	32
Tabla 7 Características generales SonarQube.....	40
Tabla 8 Características técnicas SonarQube	40
Tabla 9 Características de la base de datos de SonarQube	40
Tabla 10 Tabla jiraissue	50
Tabla 11 Tabla de unión entre jiraissue y version/components	51
Tabla 12 Tabla issue links	51
Tabla 13 Tabla subtasks	51
Tabla 14 Características generales Jira.....	53
Tabla 15 Características técnicas Jira	53
Tabla 16 Características sobre la base de datos Jira	53
Tabla 17 Tabla scmlog	62
Tabla 18 Tabla scmlog	62
Tabla 19 Tabla actions	63
Tabla 20 Tabla branches.....	63
Tabla 21 Tabla Metrics	63
Tabla 22 Tabla people	64
Tabla 23 Tabla repositories	64
Tabla 24 Tabla commits lines	64
Tabla 25 Tabla files.....	64
Tabla 26 Tabla file copies	65
Tabla 27 Tabla file links	65
Tabla 28 Tabla tag revisions	65
Tabla 29 Tabla tags.....	65
Tabla 30 Tabla attachments	66
Tabla 31 Tabla Bugs.....	66
Tabla 32 Tabla Bugzilla Data.....	67
Tabla 33 Tabla Changes.....	67
Tabla 34 Tabla Comments.....	68
Tabla 35 Tabla General Info	68
Tabla 36 Características generales de MetricsGrimoire	69
Tabla 37 Características técnicas de MetricsGrimoire	69
Tabla 38 Características sobre la base de datos de MetricsGrimoire.....	69
Tabla 39 Comparativa características funcionales generales	71
Tabla 40 Bases de datos soportadas	71
Tabla 41 Diferencias entre decorador y sensor	82

Resumen

Este proyecto trata sobre la calidad de los proyectos software desde el estudio de las herramientas que hay actualmente en el mercado y que sirven de ayuda para la toma de decisiones. Nos centraremos en el estudio de herramientas enfocadas a presentar la información de forma gráfica, mediante dashboard o cuadros de mando, dada la importancia que tienen hoy en día estas herramientas en los proyectos software.

El trabajo finalizará con la implementación de un plugin para alguna de las herramientas estudiadas para demostrar cómo podemos extender estas herramientas para adaptarlas a nuestro entorno de trabajo.

Preface

This paper deals with the quality of software projects from the study of the tools currently on the market that are helpful for decision making. We will focus on the study of tools focused on presenting information graphically, by dashboard or scorecard, given the importance of these tools in software projects.

The paper ends with implementing a plugin for some of the tools studied to show how we can extend these tools to suit our environment.

Resumen extendido

Este trabajo trata sobre la calidad de los proyectos software desde el estudio de las herramientas de gestión de proyectos, herramientas de calidad del código y herramientas de gestión de incidencias y problemas.

La primera parte del trabajo, se centrará en realizar una búsqueda exhaustiva del mayor número de herramientas posible que tengan como finalidad ayudar a mejorar la calidad de los proyectos. En esta búsqueda, excluirémos aquellas herramientas que no presenten la información de forma gráfica, dado que la presentación de la información será uno de los puntos más importantes del trabajo.

Los datos, y una vez procesados, la información, son de vital importancia para el desarrollo de los procesos que intervienen en el desarrollo de los proyectos software. La presentación de forma gráfica ayuda a transmitir la información de un modo más expresivo, y nos va a permitir con sólo echar un vistazo entender lo que está pasando, como se está desarrollando el proyecto, observar sus características más importantes e incluso sacar alguna conclusión inicial.

La segunda parte del trabajo se centrará en el análisis de tres herramientas que serán escogidas del listado que se haya realizado previamente. Estas herramientas deberán cumplir una serie de características y el análisis se realizará en diferentes fases.

En primer lugar haremos una breve introducción a la herramienta, y expondremos las razones que nos han llevado a escoger esa herramienta. A continuación, analizaremos el front office de la herramienta, es decir, qué características funcionales nos ofrece la herramienta y cómo nos muestra la información visualmente.

El siguiente paso será analizar el back office de la herramienta para ver cómo funciona internamente, qué mecanismos internos implementa y qué métricas, metodologías o marcos de trabajo utiliza para ofrecernos unas u otras capacidades. Por último, estudiaremos la base de datos de la herramienta. Analizaremos cómo y dónde almacena la información obtenida anteriormente y cómo podremos acceder a ella.

Para finalizar esta segunda parte del trabajo, realizaremos un análisis en conjunto de las tres herramientas y las compararemos para ver qué características comunes tienen y en que difieren cada una de ellas.

Por último, estudiaremos cómo podemos extender las herramientas mediante plugins, para añadir funcionalidades que necesitemos en nuestro entorno de trabajo, e implementaremos un plugin para una de las herramientas.

1 Herramientas gráficas de calidad del software

1.1 Introducción

Obtener la máxima información posible es esencial para todo equipo que participe en el ciclo de desarrollo de un proyecto software. El problema de conseguir la información se acentúa, si tenemos en cuenta que el grado de externalización de los equipos es cada vez mayor debido al ahorro en costes que supone.

Este problema de información nos plantea tres cuestiones principales:

- ¿Qué datos necesitamos? Si bien es cierto que la falta de datos puede suponer un problema, es igualmente cierto que tener demasiada información sin ninguna organización, puede hacer que no nos centremos en los problemas principales y acabemos dispersándonos en problemas secundarios o de menor importancia. Por tanto, necesitamos asignar una importancia a los distintos datos e ir resolviendo los problemas en un orden de prioridad que establezcamos.
- ¿Cómo obtener estos datos? Es esencial que podamos obtener los datos de una forma automática y a ser posible, que se realice un análisis previo conforme a unos estándares o patrones, para que la información se presente de una forma estructurada y clara para un posterior análisis más profundo. Así mismo, es importante que la información se presente de forma gráfica.
- ¿Cuándo obtener los datos? Antes, durante y después de finalizar el desarrollo de un proyecto software.

Antes de continuar conviene dejar clara la diferencia entre datos e información. Los datos son representaciones simbólicas (numérica, alfabética, algorítmica, etc.) de un atributo o variable cuantitativa o cualitativa. Los datos aisladamente pueden no contener información relevante. Para que los datos nos aporten información útil, hay que analizarlos y procesarlos. A esto, es a lo que llamamos información.

Nuestro estudio se centrará en el análisis de herramientas que obtengan datos y que a su vez, procesen esos datos y nos aporten un primer nivel de información. Posteriormente, habrá que analizar los datos obtenidos para que nos aporten niveles de información más profundos ya que las herramientas de hoy en día no pueden aportar el mismo razonamiento y tener en cuenta las distintas variables que nos podemos encontrar en un entorno real.

El análisis de las herramientas se ha llevado a cabo mediante tres fases:

1. Planificación: En esta primera fase, se definirá de forma precisa el objetivo del análisis, y el procedimiento que usaremos para evaluar las herramientas.
2. Análisis: Se estudiarán las distintas herramientas y se escogerán algunas de ellas para un análisis más profundo siguiendo el procedimiento anteriormente definido.
3. Conclusiones: Finalmente, se sacaran una serie de conclusiones respecto a los resultados obtenidos en el análisis previo.

1.2 Planificación

Esta sección define el alcance de las herramientas que son relevantes para este estudio, así como el procedimiento para analizar dichas herramientas. Como ya se mencionó anteriormente, analizaremos herramientas relativas a la obtención de datos de los proyectos software.

Podemos dividir en dos grandes grupos el tipo de información que se puede obtener, aunque los dos tipos de información influirán directamente con la calidad final del proyecto. Por un lado tenemos la información orientada a la gestión, donde se evalúan los recursos y se estiman plazos y entregas, y por otro lado, tenemos las herramientas orientadas a la parte técnica que permiten llevar un seguimiento del proceso de desarrollo, de los errores e incidencias que van surgiendo o analizar el propio código para evaluar la calidad del desarrollo.

En este estudio se tendrán en cuenta los dos tipos de herramientas dado que el objetivo del trabajo consiste en obtener datos que influyan directamente en la calidad de un proyecto software. Habrá herramientas de uno u otro tipo, pero también nos encontraremos con herramientas que extraigan información tanto técnica como de gestión.

En un principio, se pensó en excluir del estudio todas las herramientas que no fueran open source, pero a medida que se ha ido avanzando en el mismo, nos hemos encontrado con multitud de herramientas no gratuitas, que tienen una gran relevancia en proyectos reales. Por tanto tendremos en cuenta las herramientas con licencias privativas, aunque a la hora de analizar en profundidad daremos más importancia a las herramientas open source, ya que dispondremos de más información y nos será más fácil adquirir la herramienta.

Otro requisito importante para el estudio, será que las herramientas puedan presentar la información de forma gráfica, en dashboard, mediante gráficos y otros elementos visuales que ayuden a tener una idea clara del estado actual del proyecto y faciliten la toma de decisiones. Además, este tipo de herramientas suelen ser más sencillas de usar que las herramientas de línea de comandos.

1.3 Análisis de herramientas

En este apartado se realizará un análisis de las herramientas en dos fases. En primer lugar, se describirá brevemente cada una de las herramientas descritas en la tabla 1 y a continuación se hará un análisis más profundo de tres de estas herramientas.

1.3.1 Listado previo

El listado que se muestra a continuación se ha obtenido consultando diferentes fuentes de información que pueden consultarse en la sección bibliografía.

YEAR	NOMBRE	URL	LICENCIA
2015	Bugzilla	https://www.bugzilla.org/	Libre
2014	Mantis BugTracker	https://www.mantisbt.org/	Libre
2013	Jira	https://es.atlassian.com	Privada
2012	Trac	http://trac.edgewall.org/	Libre
2014	Airbrake	https://airbrake.io/	Privada
2015	Sifter	https://sifterapp.com/	Privada
2014	Bugherd	http://bugherd.com/	Privada
2015	DoneDone	https://www.getdonedone.com/	Privada
2014	BugDigger	http://bugdigger.com/	Privada
2013	Bugloghq	http://www.bugloghq.com/	Libre
2010	FogBugz	http://www.fogcreek.com/fogbugz/	Privada
2011	BOTO	http://www.botoapp.com/	Privada
2012	LogDigger	http://logdigger.com/	Libre
2014	LightHouse	http://lighthouseapp.com/	Libre/Privada
2014	Bugify	https://bugify.com/	Libre/Privada
2015	Pivotal Tracker	http://www.pivotaltracker.com/	Privada
2014	Snowy Evening	https://snowy-evening.com/	Libre/Privada
2015	Redmine	http://www.redmine.org/	Libre
2015	Request Tracker	https://bestpractical.com/rt/	Libre
2015	OTRS	http://otrs.org/	Libre
2015	Fossil	http://www.fossil-scm.org	Libre
2014	The Bug Genie	http://www.thebuggenie.com/	Libre
2015	SonarQube	http://www.sonarqube.org/	Libre
¹ 2011	MetricsGrimoire	https://metricsgrimoire.github.io/	Libre

Tabla 1 Resumen de las herramientas

¹ MetricsGrimoire es un conjunto de herramientas con diferente fecha de actualización. Hemos hecho una media con la última versión de cada herramienta para obtener la fecha resultante de la tabla.

1.3.2 Análisis breve de las herramientas

Bugzilla

Herramienta basada en web para el seguimiento de errores, originalmente desarrollada y usada por el proyecto Mozilla.

Mantis Bug Tracker

Mantis Bug Tracker es un software que constituye una solución completa para gestionar tareas en un equipo de trabajo. Es una aplicación OpenSource realizada con php y mysql que destaca por su facilidad y flexibilidad de instalar y configurar. Esta aplicación se utiliza para testear soluciones, hacer un registro histórico de alteraciones y gestionar equipos remotamente.

Jira

JIRA es una aplicación basada en web para el seguimiento de errores, de incidentes y para la gestión operativa de proyectos. Jira también se utiliza en áreas no técnicas para la administración de tareas.

Trac

Es una herramienta para la gestión de proyectos y el seguimiento de errores escrita en Python, inspirado en CVSTrac. Su nombre original era *svntrac*, debido a su fuerte dependencia de Subversion. Está desarrollado y mantenido por Edgewall Software, es software libre y de código abierto.

Airbrake

Aplicación para seguimiento y reporte de errores. Anteriormente conocida como Exceptional, está construido sobre una arquitectura robusta, que garantiza la seguridad de la información mediante una serie de filtros.

Sifter

Herramienta robusta para seguimiento de errores e incidencias para garantizar productos de calidad. Necesita menos configuración que otras herramientas, lo que permite dedicar más tiempo al producto software.

Bugherd

Herramienta que ofrece de una forma sencilla capturar la retroalimentación del cliente, resolver incidencias y gestionar los proyectos.

DoneDone

Es una herramienta simple y efectiva para seguimiento de errores, peticiones e ideas a través de todas las fases de desarrollo del software.

BugDigger

Es una herramienta para aplicaciones web, que permite capturar de forma inteligente datos de entorno, capturas de pantalla y el historial de uso de la web.

Bugloghq

Herramienta Open source para centralizar y automatizar la gestión de errores de múltiples aplicaciones.

FogBugz

Construido para seguimiento de errores, incidencias y soporte a clientes por medio de un sistema de tickets, durante todas las fases de desarrollo

Boto

Herramienta que ofrece un sistema de seguimiento de errores ágil con una interfaz de usuario intuitiva.

LogDigger

LogDigger ofrece un conjunto de herramientas para ayudar a recopilar y clasificar los informes de error y los registros de las aplicaciones Java, de forma que se puedan corregir los errores de una forma más rápida.

LightHouse

Ofrece seguimiento de problemas simples y sin esfuerzo de colaboración en proyectos para ayudar a mantener un seguimiento de su desarrollo del proyecto.

Bugify

Es un sistema simple pero potente de seguimiento de problemas, que cuenta con un sistema para marcar prioridades, los filtros de búsqueda, atajos de teclado, las notificaciones de correo electrónico y otras muchas características.

Pivotal Tracker

Es una herramienta de gestión de proyectos ágiles enfocada a dividir el proyecto en historias tamaño pequeño, y utiliza puntos para estimar la complejidad relativa de cada historia, y dar prioridad a la misma.

Snowy Evening

Es una aplicación de seguimiento de problemas que pretende ser lo suficientemente potente para que los desarrolladores, pero lo suficientemente simple para que los clientes la utilicen con facilidad.

Redmine

Está escrito en Ruby on Rails. Aparte de seguimiento de incidencias, ofrece un conjunto de funcionalidad de gestión de proyectos.

Request tracker

Escrito en Perl, esta herramienta puede usarse como un sistema de tickets para seguimiento de problemas.

OTRS

Escrito en Perl. Tiene todas las funcionalidades estándar de seguimiento de incidencias.

Fossil

Escrito en C, utiliza SQLite como base de datos. Aparte de seguimiento de errores, contiene una wiki. La instalación de la herramienta es de las más sencillas de este tipo de software.

The Bug Genie

Escrito en PHP, proporciona un software de seguimiento de errores basado en un asistente que ayuda gestionar dichos errores.

SonarQube

Plataforma de código abierto para inspeccionar de forma continua la calidad del código fuente.

MetricsGrimoire

Conjunto de herramientas para obtener datos de los repositorios relacionados con el desarrollo de software, como los sistemas de control de versiones y los relacionados con el seguimiento de incidencias.

1.4 Análisis en profundidad de tres herramientas

En esta sección haremos un análisis sobre tres herramientas de diferente tipo.

1.4.1 Características comunes

Al finalizar el estudio de las tres herramientas vamos a realizar una comparación sobre una serie de características que creemos importantes. Las características se dividen en tres grupos:

Características generales: Nombre de la herramienta, año de primera y última versión, tipo de licencia y otras. Con estas características pretendemos hacer un resumen de la herramienta estudiada que sirva de un primer filtro para escoger una u otra. Por ejemplo, podemos estar interesados únicamente en herramientas de licencia gratuita, y este resumen nos serviría para descartar algunas herramientas.

Características técnicas: Este tipo de características podemos dividirlo en dos tipos: por un lado las características funcionales generales que serán adquisición de datos, análisis de métricas y presentación de la información, y por otro lado, las características particulares de cada herramienta.

Esta división servirá por un lado para realizar una comparación entre las herramientas mediante las características funcionales generales, y por otro lado, para realizar un resumen de qué nos puede aportar cada herramienta en su campo de trabajo.

Otras características técnicas serán el lenguaje utilizado para el desarrollo de la aplicación y si disponemos de soporte profesional.

Características de la base de datos: Estudiaremos el modo de acceso a la base de datos y los gestores soportados por cada herramienta. Es importante conocer esta información dado que será donde guardemos todos nuestros datos.

1.4.2 SonarQube

1.4.2.1 ¿Por qué SonarQube?

Hemos elegido esta herramienta por los siguientes motivos:

- Es una herramienta open-source.
- Tiene detrás a una extensa comunidad, que sigue desarrollando y mejorando la herramienta, lo que hace que este actualizada día a día.
- Es compatible con, prácticamente, todos los lenguajes de programación.
- Dada la cantidad de plug-in existentes, ofrece muchas posibilidades y permite que se adapte a diferentes entornos de trabajo. Además, ofrece la posibilidad de desarrollar plug-in propios para usos más concretos.
- Es compatible con otras herramientas ampliamente usadas, como Jenkins, Subversion o Jira.
- A nuestro modo de ver, realiza los análisis más completos en cuanto a calidad de código se refiere, abarcando multitud de métricas y medidas.
- La interfaz gráfica es muy intuitiva y ofrece multitud de gráficos y representaciones de la información.
- Se puede asignar trabajo a los miembros del equipo, y llevar un seguimiento de dicho trabajo.

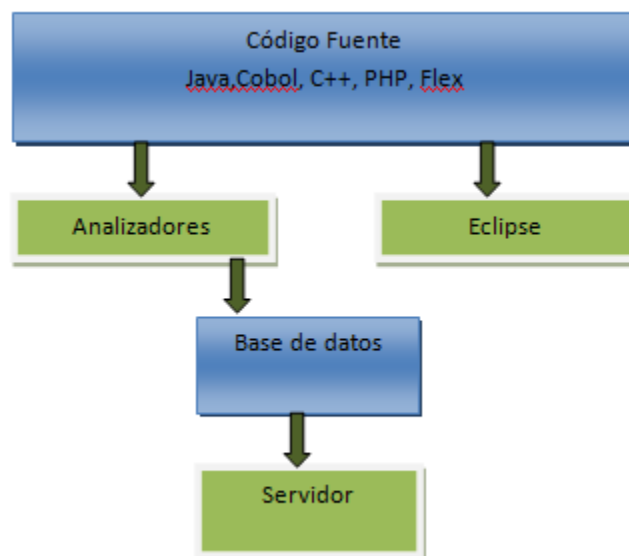


Figura 1. Arquitectura de SonarQube

1.4.2.2 Introducción a la herramienta

Sonarqube, anteriormente llamado Sonar, es una plataforma open source para medir la calidad del código software. Proporciona instantáneas o "snapshot" del proyecto y además, ofrece tendencias de retraso, *qué es lo que actualmente estamos haciendo mal*, y tendencias de futuro, *qué es probable que salga mal en el futuro*. Con estos indicadores podríamos responder a preguntas del tipo: *¿una cobertura de pruebas del 50% es un resultado positivo o negativo?* Sin más datos, podríamos pensar que es un resultado negativo, pero si el mes pasado sólo teníamos un 30% de cobertura de pruebas, sabremos que estamos en una tendencia positiva y que estamos haciendo las cosas de forma correcta. Sonar recopila la información y nos informa sobre estas tendencias, ya sean negativas o positivas.

La medición de qué es lo que se está haciendo mal, no es la única característica de SonarQube. Ofrece herramientas de administración para ayudar activamente en el proyecto como integración IDEs, integración para Jenkins, un servidor de integración continua y herramientas para revisión de código.

Otras herramientas de mercado, en el área de la calidad del software, que hemos estado estudiando para el proyecto ofrecen algunas de estas características pero basan, principalmente, el concepto de calidad en errores y complejidad, mientras que SonarQube extiende esa definición a lo que ellos denominan *los siete ejes de la calidad del software*. Abordaremos el estudio de estos ejes más adelante, pero brevemente son: Código comentado, arquitectura y diseño, número de líneas duplicadas, test unitarios, complejidad, errores potenciales y reglas de codificación.

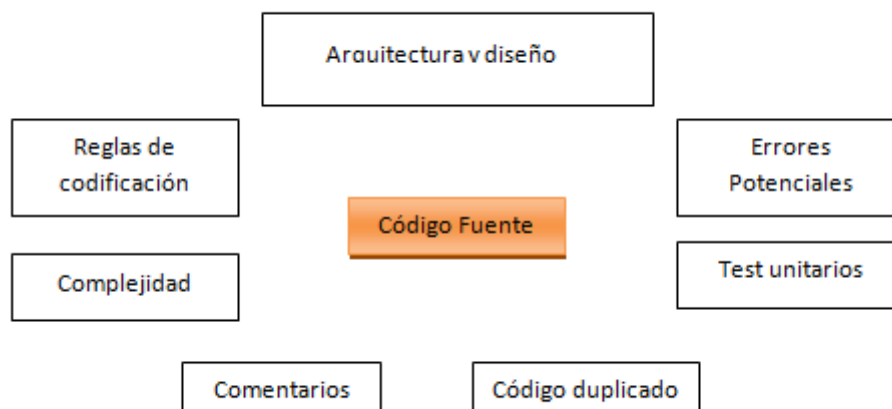


Figura 2 Los siete ejes de SonarQube

Esta herramienta tiene el potencial suficiente como para ser usada en un proyecto software para controlar y medir la calidad final de un producto. Los usuarios finales son los que valoraran en última instancia si el producto resultante es válido o no. Sin embargo, no sólo el usuario final es quién debe preocuparse por la calidad del producto. Todas las partes interesadas del sistema deberían preocuparse por la calidad y es donde Sonar cobra importancia. Si analizamos los distintos puntos de vista de las partes involucradas nos encontramos con lo siguiente:

- Desde el punto de vista de un tester, SonarQube es válido porque llega donde las pruebas automatizadas no pueden llegar. Además, sirve de apoyo en las pruebas manuales y de seguridad.
- Desde el punto de vista de un desarrollador, Sonar merece la pena porque te ayuda a crecer como programador, ya sea mediante ayudas sutiles sobre un lenguaje de programación específico o a mejorar temas de seguridad o gestión de recursos. Ya seas un programador experimentado o uno recién salido de la escuela, Sonar te ayudará en estas tareas, aunque sólo sea para hacer que tu código sea más legible y elegante.
- Desde el punto de vista de un arquitecto software, Sonar merece la pena porque ayuda a mantener la atención a si el diseño inicial del producto está siendo degradado con el tiempo, o a comprobar si la complejidad interna está creciendo y hay que reprogramar algunos módulos.
- Desde el punto de vista de un jefe de proyecto, Sonar es válido porque el testing automático no es suficiente. Esto sólo muestra la calidad externa del producto, es decir, si el producto hace lo que se supone que debe de hacer. Con Sonar se analiza la calidad interna, y se comprueba si el software se ejecuta de forma óptima, y si es posible mantener o extender en el futuro.
- Desde un punto de vista de negocios, Sonar merece la pena porque ofrece un fuerte retorno de inversión, dado que es un software libre, con coste de instalación bajos y con interfaces intuitivos que hacen que el coste de formación sean mínimos.
- Por último, desde el punto de vista de gestión, Sonar vale la pena porque ofrece métricas y medidas abstractas sobre la calidad del código.

1.4.2.3 *Front Office de SonarQube*

En esta sección vamos a analizar el front office de la aplicación para hacernos una idea de las posibilidades que ofrece. No pretende ser una guía de usuario extensa dado que está fuera del alcance de este estudio, y existen otras fuentes dedicadas a ello. Es importante destacar que todas las métricas que se van a describir se aplican, en su totalidad, únicamente al lenguaje Java. Empezaremos esta sección mostrando una tabla resumen de las métricas cubiertas para los lenguajes más usados.

- Lenguajes cubiertos por SonarQube

A continuación, podemos una lista de los lenguajes soportados por SonarQube junto con las métricas que se les aplican con los plugin actuales. Dado que la lista de plugin y lenguajes crece con el tiempo, la información aquí descrita puede variar con facilidad.

LANGUAGE	PAGO/LIBRE	METRICAS
ABAP	Pago	Tamaño, comentarios, complejidad, duplicaciones, problemas.
C	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas.
C++	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas.
C++	Pago	Tamaño, comentarios, complejidad, duplicaciones, problemas.
C#	Pago	Tamaño, comentarios, complejidad, duplicaciones, problemas, test.
Cobol	Pago	Tamaño, comentarios, complejidad, duplicaciones, problemas, test. Métricas específicas de Cobol.
Delphi	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas, test. y diseño
Drools	Libre	Tamaño, comentarios, problemas
Flex/ActionScript	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas, test.
Groovy	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas, test.
JavaScript	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas, test.
Natural	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas .
PHP	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas
PL/I	Pago	Tamaño, comentarios, complejidad, duplicaciones, problemas
PL/SQL	Pago	Tamaño, comentarios, complejidad, duplicaciones, problemas
Python	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas
Visual Basic 6	Pago	Tamaño, comentarios, complejidad, duplicaciones, problemas
Web (JSP, JSF, XHTML)	Libre	Tamaño, comentarios, complejidad, duplicaciones, problemas
XML	Libre	Tamaño y problemas

Tabla 2 Plugin disponibles y métricas que soportan

- Página principal de la aplicación:

La figura de más arriba muestra la pantalla por defecto de SonarQube. En la parte superior derecha tenemos la lista de proyectos analizados junto con algunos atributos clave, y en la parte inferior un mapa de árbol con los mismos proyectos. Se muestra en detalle en la figura 3.

Lista de proyectos

Mensaje de bienvenida con
enlaces de ayuda



Mapa en árbol de la lista de
proyectos

Figura 3 Página principal de SonarQube

- Dashboard

Pulsando sobre alguno de los proyectos de la lista llegamos al Dashboard de SonarQube. Se puede personalizar y agregar o quitar los widget que consideremos necesarios.



Figura 4 Dashboard por defecto

- Widget Generales

A continuación, vamos centrar nuestra atención en los widget generales, es decir, los que no entran en ninguna categoría de errores, si no que son generales a todos los proyectos.

El primero de ellos, hace referencia al **número de líneas de código**. En este widget, se pueden observar cuantas líneas, métodos clases y paquetes se han encontrado durante el análisis.

Size Metrics Delete						
Lines Of Code	Files		Functions			
18 ↗	1		2			
Java	Directories	Lines	Classes	Statements	Accessors	
	1	28 ↗	1	9 ↗	0	

Figura 5 Size metrics widget

Este widget nos muestra tanto las líneas de código (LoC), como el número de líneas físicas de todo el código. El número de líneas físicas hace referencia al número de veces que alguien ha pulsado la tecla ENTER, haya o no, contenido en esa línea, mientras que las líneas de código hacen referencia a las líneas de código "trabajadas". Aunque las líneas de código dependen del lenguaje usado, SonarQube las calcula para Java, restando todas las líneas en blanco y los comentarios.

La figura 6 muestra el **widget de eventos**. Este widget muestra una lista rápida de los eventos ocurridos en la aplicación. Hay muchos tipos de eventos, como el cambio de la versión de un proyecto.

Events Delete		
Events	All ▼	
Apr 05 2015	Version	1.0-SNAPSHOT

Figura 6 Events widget

Para finalizar con los widget generales, en la figura 7 podemos observar **el widget de descripción**. Este Widget muestra un pequeño resumen del proyecto cuando se realizó el análisis. Podemos ver el lenguaje, ID, perfil y una serie de enlaces de interés.



Figura 7 Widget que muestra información general

- Los siete ejes de calidad

Como mencionamos anteriormente, SonarQube cubre lo que los desarrolladores de la herramienta llaman *Los siete ejes de la calidad*. Vamos a ver en qué consiste cada uno de ellos y a ver en qué widget se representan. Recordemos que los siete ejes eran Código comentado, arquitectura y diseño, número de líneas duplicadas, test unitarios, complejidad, errores potenciales y reglas de codificación.

Comenzamos analizando los ejes **de errores potenciales**. La figura 8 muestra el widget de problemas.

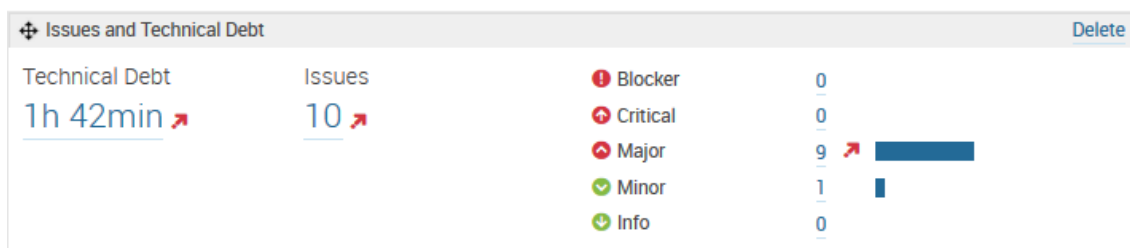


Figura 8 Issues Widget

Algunos de estos problemas que resolver pueden ser infracciones en el código, algunas de ellas más importantes que otras, y por eso se agrupan en grado de severidad como podemos observar en la imagen. Aunque este grado viene asignado por defecto, nosotros podemos cambiarlo y aumentar o disminuir la severidad de los tipos de error. El porcentaje de Rules compliance se basa en el número y severidad de los errores contra el número de líneas de código. Cuanto más alto sea este porcentaje mejor estaremos haciendo las cosas, mientras que el número de problemas nos interesa que este bajo.

El siguiente eje a analizar es el eje de **cobertura de test unitarios**. Se muestra en la figura 9.

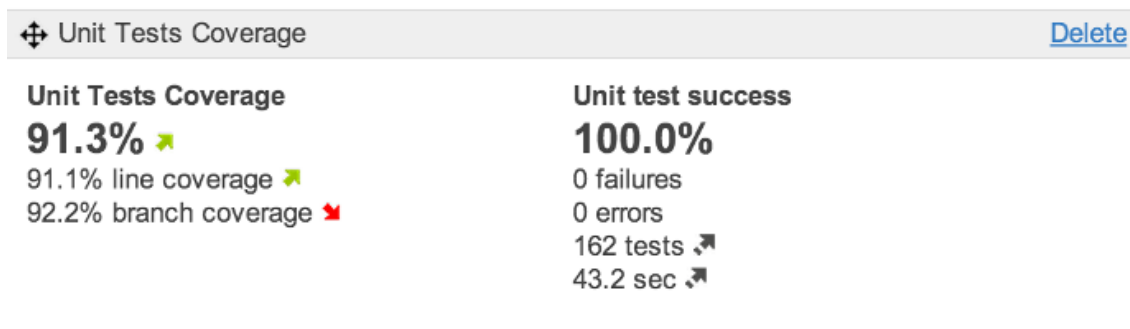


Figura 9 Cobertura Widget, de (<http://www.adictosaltrabajo.com/>)

Este widget muestra el porcentaje de test unitarios que pasa nuestra aplicación, es decir, que todas las unidades independientes funcionen correctamente.

Los siguientes ejes a analizar son los comentarios y la duplicidad de código que, como ocurriría con los ejes de errores potenciales y reglas de codificación, se agrupan en un mismo widget. Se muestra en la figura 10.

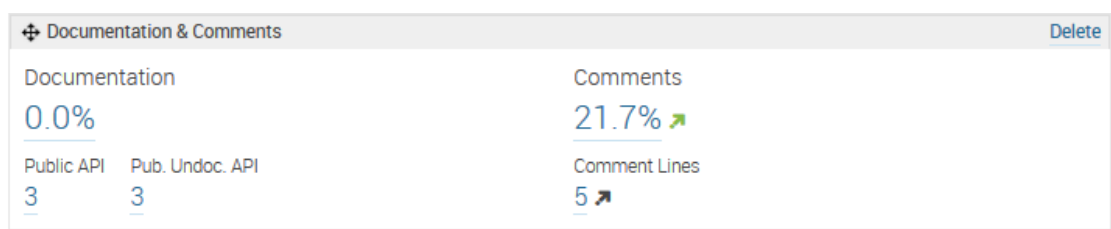


Figura 10 Comentarios y líneas duplicadas

Los comentarios se pueden dividir en dos tipos; aquellos que van dentro de los métodos y que pretenden explicar algo del funcionamiento lógico de la aplicación, y los comentarios externos al método, aquellos que definen el por qué y el cómo debemos usar dicho método. El segundo tipo de comentarios, conocido como documentación API es lo que mide Sonar. Es una medida de mantenibilidad, y se miden porque los comentarios hacen que un sistema sea más fácil de usar y de mantener. Los desarrolladores no deberían perder tiempo averiguando que es lo que pensaba y con qué motivo otro desarrollador cuando hizo ese método, sino, simplemente utilizar ese método si lo necesita.

Por otro lado, tenemos el código duplicado. SonarQube considera por defecto que el código está duplicado si tres líneas consecutivas de código se repiten varias veces durante el análisis. Esta medida se puede cambiar incrementando o disminuyendo el número de líneas que consideremos necesarias.

A continuación, analizamos el eje **de arquitectura y diseño**. Una de las partes de este eje mide la limpieza y la elegancia de la arquitectura. El plan original de diseño seguramente fuera elegante y simple, y lo que mide SonarQube es qué nivel ha alcanzado el desarrollo actual. La figura 11 muestra el widget indicado.

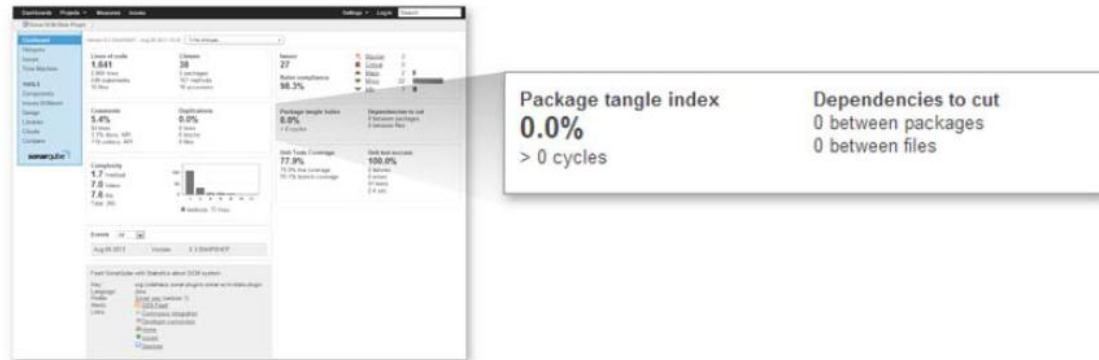


Figura 11 Métricas sobre la arquitectura, de (SonarQube in Action)

La otra parte del eje está compuesto por los widget LCOM4 y RFC, o Response for Class. En versiones anteriores estos widget se mostraban por defecto, pero en las nuevas versiones los desarrolladores creen que son métricas demasiado complejas, y, de hecho, el widget LCOM4 ha sido eliminado y los creadores de la herramienta no ofrecen soporte, debido a que en los proyectos en la vida real generan demasiados falsos positivos para ser usados.

En la figura 12 se muestra la representación para estas métricas. La representación de los gráficos indica que cuanto más grandes sean las barras a la izquierda, más simples y sencillas serán las clases de nuestro código.

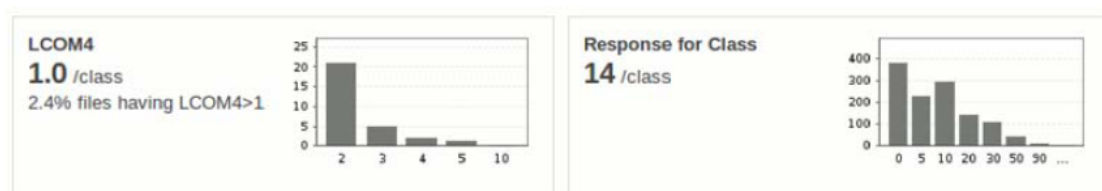


Figura 12 Métricas LCOM4 y RPC

Por último, hablamos del eje relacionado con la **complejidad**. Esta métrica lo que hace esencialmente es medir los pares de llaves de apertura y de cierre de un método. Cuanto más pares, más complejo será el método y por tanto, más compleja será la lógica del método. Como consecuencia, cuanto más compleja sea la lógica, será más complejo de mantener y de entender.



Figura 13 Complejidad de las clases

1.4.2.4 Back office de SonarQube

En esta sección, vamos a analizar qué herramientas utiliza SonarQube internamente, así como estándares de calidad y el método SQALE.

Solo vamos analizar las herramientas por defecto y las más comunes y usadas para el lenguaje Java. Existen multitud de plugin que añaden métricas para los distintos lenguajes pero queda fuera del alcance de este trabajo analizar todas ellas.

- PMD - CheckStyle - FindBugs

Empezamos por el trío de herramientas más conocidas que usa SonarQube. Son tres herramientas que se centran en aspectos diferentes de la calidad del software y que, por tanto, son totalmente complementarias. CheckStyle se centra en las **reglas de convención**, PMD en las **malas prácticas** y FindBugs se centra en los **errores potenciales**.

Los tipos de **convenciones** cubren nombres, comentarios y formato de convenciones. Algunos ejemplos son:

- ¿Hay JavaDoc en los métodos públicos?
- ¿El proyecto está siguiendo las convenciones de nombres de Sun?
- ¿El código está escrito con un formato consistente?

Los tipos de convenciones suelen tener a menudo una reputación bastante mala debido a que las reglas son muy simples. Es cierto que estas reglas no tienen ningún impacto en cuanto a estabilidad, rendimiento o confiabilidad en la aplicación, pero son el pegamento que permite que un equipo pueda trabajar junto y gastar sus energías en mejorar la herramienta en vez de perderla en entender la inconsistencia del código.

En la tabla 3 se encuentra el resumen de la herramienta

ATRIBUTOS	VALOR
Métricas implementadas	Violaciones de convenciones de código, code smells
Año de la primera versión	2001
Año de la última versión	2014
Licencia	<u>Lesser GNU General Public License</u>
Web	http://checkstyle.sourceforge.net/

Tabla 3 Resumen CheckStyle

Las **malas prácticas** consisten en comportamientos conocidos que llevan continuamente a lidiar con dificultades todo el tiempo. Algunos ejemplos son:

- Capturar una excepción sin realizar ninguna acción
- Tener código muerto
- Demasiados métodos complejos
- Usar directamente la implementación, en vez de las interfaces
- Implementar el método `hashCode()` sin el método `not equals(Object object)`

PMD te informa de este tipo de errores, informándote de las malas prácticas. PMD incluye CPD, *Copy Paste Detector*, para localizar código duplicado. En la tabla 4 vemos el esquema de caracterización de la herramienta.

ATRIBUTOS	VALOR
Métricas implementadas	Código duplicado, code smells
Año de la primera versión	2002
Año de la última versión	2014
Licencia	Lesser GNU General Public License
Web	http://pmd.sourceforge.net/

Tabla 4 Resumen PMD

Los tipos de errores potenciales ayudan a detectar que no está claramente visible en el código y a entender qué secuencias de código podrían tener errores potenciales. Algunos ejemplos son:

- Sincronización en booleanos, podrían llevar a puntos muertos (deadlock)
- Se puede llegar a exponer representaciones internas devolviendo referencias a objetos mutables
- Los métodos usan el mismo código para dos ramas

FindBugs utiliza un detector auxiliar conocido como Fb - Contrib. Añade algunas funciones extra a FindBugs como búsqueda de *if- else* complejos con multitud de condicionantes or en la condición, búsqueda de métodos públicos cuando podrían ser privados o problemas con literales de cadena creados manualmente. En la tabla 5 vemos el esquema de caracterización de la herramienta.

ATRIBUTOS	VALOR
Métricas implementadas	Code smells
Año de la primera versión	2003
Año de la última versión	2015
Licencia	Lesser GNU General Public License
Web	http://findbugs.sourceforge.net/

Tabla 5 Resumen FindBugs

Hay que tener en cuenta que FindBugs analiza el byte code, no el código Java. Esto puede ocasionar que en ocasiones SonarQube muestre el error en la línea equivocada. En ese caso habría que mirar varias líneas por encima o por debajo para ver realmente donde está el error.



Figura 14 Comportamiento de FindBugs, de (SonarQube in Action)

Además, para que la conexión sea total entre FindBugs y SonarQube, hay que poner el parámetro debug a ON cuando se ejecute el comando javac correspondiente, para que incruste el número de línea de forma correcta y pueda ser utilizado por FindBugs.

- Junit y las herramientas de cobertura de test unitarios de SonarQube

JUnit es, sin ninguna duda, la herramienta más usada y extendida en la comunidad testing en cuanto a pruebas unitarias se refiere y SonarQube no iba a ser la excepción.

Pero, como ya hemos dicho anteriormente, SonarQube nos ofrece un amplio conjunto de analíticas y resultados, y uno de ellos es la cobertura de tests unitarios. Hay cuatro plugin disponibles para este fin: Clover, Cobertura, Emma y JaCoCo. En la tabla 6 vemos un resumen de estos cuatro plugin.

	Clover	Cobertura	Emma	JaCoCo
Licencia	Comercial	GNU GPL	CPL	EPL
Última versión estable	3.0.2 (13 de abril 2010)	1.9.4.1 (3 de marzo 2010)	2.0.5312 (13 de junio 2005)	0.4.0 (4 de junio 2010)
Tipo de instrumentación				
Java	1.4 +	1.3 +	1.2 +	1.5 +
Cobertura por línea	Si	Si	Si	Si
Cobertura de ramas	Si	Si	No	No
Procesos integrados Con SonarQube	Instrumentación Compilación Ejecución Generador de informes Informe de análisis	Instrumentación Ejecución Generador de informes Informe de análisis	Instrumentación Ejecución Lectura de datos	Ejecución Lectura de datos

Tabla 6 Comparación herramientas de cobertura de test

A fecha del proyecto, el plugin para Emma está obsoleto desde la versión 4.2 de SonarQube y el plugin para JaCoCo, que recomiendan usarlo en vez de Emma, se encuentra igualmente obsoleto desde la versión 2.4 de Java. Hemos querido incluirlos en el análisis dado que en cualquier momento pueden volver a estar disponibles con el 100% de sus funcionalidades, dada la amplia comunidad que tiene detrás SonarQube.

- SQALE (Software Quality Assessment based on Lifecycle Expectations)

En esta sección vamos a hablar sobre SQALE. SQALE es un método que permite evaluar el código de un proyecto para software para determinar la calidad del mismo. Está especialmente dedicada a la gestión de la deuda técnica. Permite:

- Define claramente el origen de la deuda técnica
- Ayuda estimar correctamente esta deuda
- Ayuda analizar la deuda desde una perspectiva técnica y de negocio
- Ofrece diferentes estrategias de priorización permitiendo establecer métodos óptimos de recuperación debido a la deuda técnica

Se basa en cuatro conceptos:

1. El modelo de calidad: Se usa para formular y organizar los requisitos no funcionales relativos a la calidad del código.
2. El modelo de análisis: Contiene por un lado las reglas que son usadas para normalizar las medidas y los controles del código, y por otro lado las reglas para agregar los valores normalizados.
3. Los índices: representan los costes. Los costes pueden ser calculados en unidad de trabajo, de tiempo o de dinero. En todos ellos, los índices de valor se representan de una escala de tipo ratio. Los índices son:
 - SQALE Testability Index : STI
 - SQALE Reliability Index : SRI
 - SQALE Changeability Index : SCI
 - SQALE Efficiency Index : SEI
 - SQALE Security Index : SSI
 - SQALE Maintainability Index : SMI
 - SQALE Portability Index : SPI
 - SQALE Reusability Index : SRUI
4. Los indicadores: SQALE define cuatro indicadores sintetizados. Proporcionan capacidades de análisis y apoyo a la toma de decisiones para la gestión de la calidad del código fuente y de la deuda técnica.

En la figura 15 vemos un esquema de los índices. Como indica la flecha, los índices más importantes se encuentran en los niveles inferiores y los menos importantes en los superiores. Así, tenemos que, la testeabilidad es el más importante y la reusabilidad el que menos.

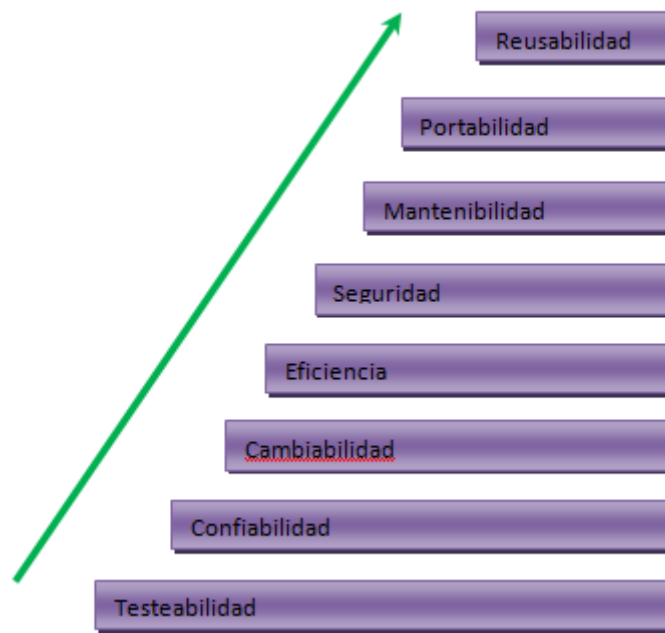


Figura 15 Esquema SQALE

SonarQube provee varios widget donde podemos visualizar esta escala. La figura 16 muestra el widget por defecto que viene con SonarQube.

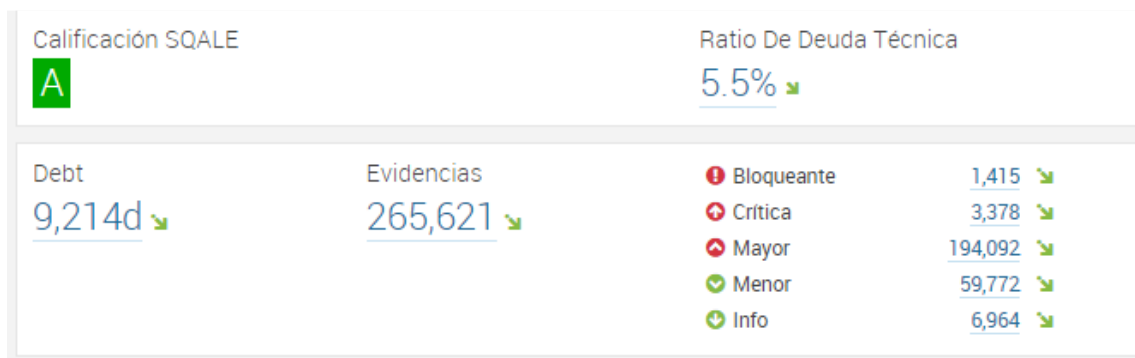


Figura 16 Widget calificación SQALE

La calificación SQALE es una escala que va desde la letra A, mejor grado, hasta la E. El ratio de deuda técnica es un ratio entre la actual deuda técnica y el esfuerzo que supondría reescribir el código desde 0. La fórmula para calcular el ratio es:

$$\text{"technical_debt"} / \text{"estimated_development_cost"}$$

donde "estimated_development_cost" es igual a:

$$\text{"LOC x 30 minutes"}$$

Por último, se muestra en la figura 23 un widget que indica donde empezar a pagar la deuda técnica. La forma de leer el widget es de abajo a arriba.

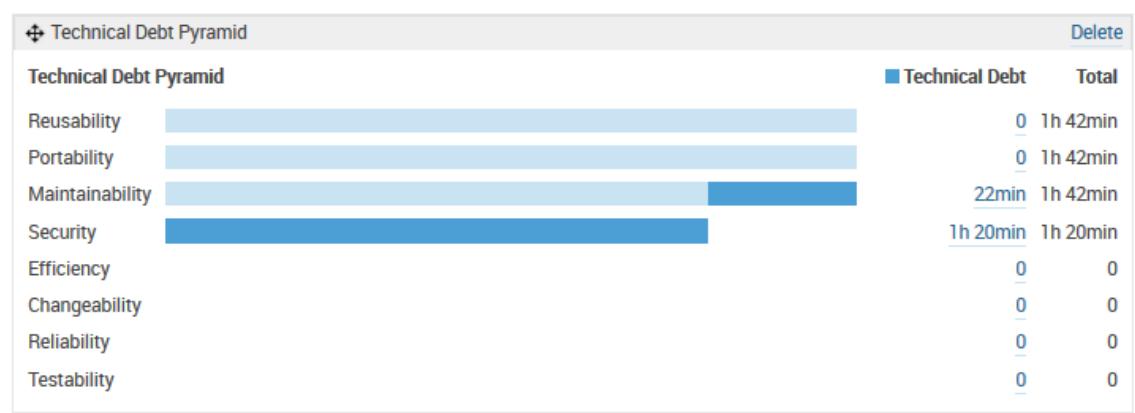


Figura 17 Technical Debt Widget

1.4.2.5 Base de datos

En la última sección del análisis de SonarQube vamos a analizar la bases de datos, y como acceder a ella.

La estructura de la base de datos de SonarQube está deliberadamente indocumentada por el equipo de desarrollo, ya que aconsejan utilizar REST para extraer la información necesaria. Por tanto, primero mostraremos las tablas e índices de la base de datos y a continuación, como acceder mediante REST.

- Esquema de la base de datos

Como hemos mencionado anteriormente, se recomienda acceder a la base de datos mediante REST. Aún así, todos los objetos que componen la base de datos se pueden encontrar en el siguiente enlace.

<https://github.com/SonarSource/sonarqube/blob/master/sonar-core/src/main/resources/org/sonar/core/persistence/schema-h2.ddl>

- Como acceder usando REST

En la página principal de SonarQube hay un enlace con toda la información referente al Web-Service-API

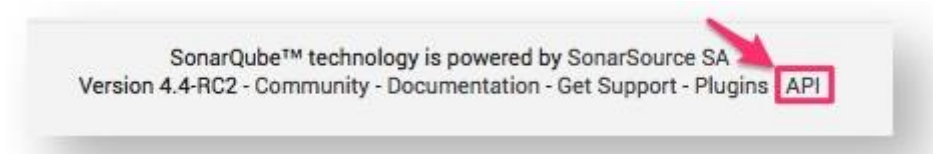


Figura 18 Enlace a la documentación de la API

Web Service API ☐ Show Internals

api/action_plans
Action plans management

- api/authentication
Check authentication credentials
- api/coverage
Display coverage information
- api/duplications
Display duplications information
- api/events
Project's events management
- api/favorites
User's favorites management
- api/issue_filters
Issue Filters management
- api/issues
Coding rule issues
- api/languages
Programming languages

api/action_plans

POST api/action_plans/close
Since 3.6
Close an action plan. Requires Administer permission on project

Parameters

format optional	Response format can be set through: <ul style="list-style-type: none"> Parameter format: xml json Or the 'Accept' property in the HTTP header: <ul style="list-style-type: none"> Accept:text/xml Accept:application/json If nothing is set, json is used Possible values: json , xml
key required	Key of the action plan Example value: 3f19de90-1521-4482-a737-a311758ff513

POST api/action_plans/create
Since 3.6
Create an action plan. Requires Administer permission on project

Parameters

deadLine optional	Due date of the action plan. Format: YYYY-MM-DD Example value: 2013-12-31
description optional	Description of the action plan Example value: Version 3.6

Figura 19 Métodos existentes en la API

Pulsando sobre el enlace tenemos la información que aparece en la figura 19. Vamos a ver algunos ejemplos de información que podemos obtener utilizando este servicio. Para las pruebas se ha utilizado el repositorio de ejemplo <http://nemo.sonarqube.org/>

Para comenzar, vamos a obtener una lista de todos los recursos disponibles en ese repositorio. Para ello accedemos al siguiente enlace:

<http://nemo.sonarqube.org/api/resources?>

Obtenemos el resultado de la figura 20.

```
<resources>
  <resource>
    <id>495453</id>
    <key>org.nuxeo:nuxeo-ecm</key>
    <name>Nuxeo ECM Projects</name>
    <lname>Nuxeo ECM Projects</lname>
    <scope>PRJ</scope>
    <qualifier>TRK</qualifier>
    <version>5.9.6-SNAPSHOT</version>
    <date>2014-09-06T11:49:32+0000</date>
    <creationDate>2013-01-02T12:47:15+0000</creationDate>
  </resource>
  <resource>
    <id>117236</id>
    <key>org.apache.excalibur:excalibur</key>
    <name>Excalibur</name>
    <lname>Excalibur</lname>
    <scope>PRJ</scope>
    <qualifier>TRK</qualifier>
    <version>4-SNAPSHOT</version>
    <date>2014-10-04T21:29:39+0000</date>
    <creationDate></creationDate>
  </resource>
</resources>
```

Figura 20 Resultado de la consulta REST (I)

Ahora vamos a acceder a un recurso en particular y a obtener información referente a varias métricas. Para ello accedemos al siguiente enlace:

http://nemo.sonarqube.org/api/resources?resource=org.apache.excalibur:excalibur&verbose=true&metrics=lines,violations,duplicated_lines

Obtenemos el resultado de la figura 21.

```
<resources>
  <resource>
    <id>117236</id>
    <key>org.apache.excalibur:excalibur</key>
    <name>Excalibur</name>
    <lname>Excalibur</lname>
    <scope>PRJ</scope>
    <qualifier>TRK</qualifier>
    <version>4-SNAPSHOT</version>
    <date>2014-10-04T21:29:39+0000</date>
    <creationDate/>
    <msr>
      <key>duplicated_lines</key>
      <name>Lineas duplicadas</name>
      <val>6867.0</val>
      <frmt_val>6,867</frmt_val>
    </msr>
    <msr>
      <key>violations</key>
      <name>Evidencias</name>
      <val>16978.0</val>
      <frmt_val>16,978</frmt_val>
    </msr>
    <msr>
      <key>lines</key>
      <name>Lineas</name>
      <val>117395.0</val>
      <frmt_val>117,395</frmt_val>
    </msr>
  </resource>
</resources>
```

Figura 21 Resultado de la consulta REST (II)

Por último, vamos a exportar esta información a un excel para poder realizar informes o presentar la información de otra manera. Accedemos al siguiente enlace:

http://nemo.sonarqube.org/api/timemachine?resource=org.apache.excalibur:excalibur&verbose=true&metrics=lines,violations,duplicated_line&format=csv

El resultado será un archivo que nuestro navegador descargará en formato csv. Se muestra en la figura 22.

	A	B
1	date,lines,violations,duplicated_lines	
2	2009-07-04T15:19:50+0000,121623,2266,3664	
3	2009-08-02T10:53:53+0000,115826,2154,3426	
4	2009-09-06T09:53:44+0000,115826,2225,3428	
5	2009-10-18T08:59:44+0000,115826,2225,3428	
6	2009-10-26T23:01:06+0000,115826,2225,3428	
7	2009-11-08T07:24:51+0000,115826,2134,3428	
8	2009-11-15T06:24:10+0000,115826,2134,3428	
9	2009-12-06T19:19:31+0000,115855,2041,3432	
10	2010-01-03T08:27:03+0000,115855,2041,3432	
11	2010-02-07T07:41:35+0000,115855,2041,3432	
12	2010-03-04T20:31:25+0000,115855,2014,3432	
13	2010-04-04T05:55:04+0000,115855,2101,3432	

Figura 22 Información exportada a excel

También podríamos obtener la misma información en formato JSON cambiando el parámetro "format":

http://nemo.sonarqube.org/api/timemachine?resource=org.apache.excalibur:excalibur&verbose=true&metrics=lines,violations,duplicated_line&format=json

1.4.2.6 Resumen

Una vez que hemos analizado la herramienta en profundidad, vamos a realizar un resumen desde tres puntos de vista diferentes: Resumen sobre características generales de la herramienta, resumen de características técnicas y un resumen sobre la base de datos.

La tabla 7 muestra las características generales de la herramienta. Destaca la madurez de la herramienta con una edad de ocho años desde la primera versión.

CARACTERÍSTICAS	VALOR
Nombre	SonarQube
Año primera versión	2007
Año última versión	2015
Licencia	LGPL v3
Software Libre/Pago	Libre
Web	http://www.sonarqube.org/

Tabla 7 Características generales SonarQube

La tabla 8 muestra las características técnicas de SonarQube. Aunque es una herramienta de libre uso, dispone de soporte profesional para empresas por medio de SonarSource.

Entre las métricas que implementa se encuentran la complejidad del código, tamaño del código, comentarios, dependencias, cobertura de test unitarios, etc.

CARACTERÍSTICAS	VALOR
Lenguaje utilizado para su desarrollo	Java
Soporte profesional	SonarSource
Características funcionales generales	Adquisición de datos, análisis de métricas, presentación de la información
Características funcionales particulares	Implementación de métricas, implementación interna de modelos de calidad

Tabla 8 Características técnicas SonarQube

Por último, en la tabla 9 encontramos la información sobre la base de datos. Aparte de las bases de datos enumeradas en la tabla, dispone de una base de datos que viene instalada por defecto y que sólo es recomendable su uso para pruebas.

CARACTERÍSTICAS	VALOR
Bases de datos soportadas	Microsoft SQL Server, MySQL, Oracle, PostgreSQL
Acceso a la base de datos	Mediante REST

Tabla 9 Características de la base de datos de SonarQube

1.4.3 Jira

1.4.3.1 ¿Por qué Jira?

Hemos elegido Jira por los siguientes motivos:

- Es una de las herramientas más usadas en cuanto a seguimiento de incidencias, errores y gestión de proyectos se refiere, usada por compañías muy importantes como ebay, adobe y la NASA.
- Aunque es una herramienta con licencia privativa, el coste nos parece muy asequible para todo tipo de empresas, ya sean pequeñas o multinacionales. Por 10€/mes o 20€/mes, según que funcionalidades necesitemos, podemos tener la herramienta para 10 usuarios. Además, la compañía ofrece licencias gratuitas para proyectos open source que cumplan unos determinados requisitos.
- La herramienta dispone de multitud de funcionalidades que hace que pueda adaptarse a diferente entornos de trabajo, tanto para gestión de proyectos, para tareas de desarrollo o para seguimiento de incidencias.
- Se pueden agregar workflow de procesos ajustando la herramienta al equipo de trabajo, y no el equipo a la herramienta.
- Se puede planificar el trabajo del equipo de una forma clara y realizar un seguimiento de lo que ya este hecho. Además, tiene extensiones para usar metodologías de trabajo ágiles.
- Se puede integrar y utilizar con otras herramientas muy conocidas y usadas como Jenkins o SonarQube.

1.4.3.2 Introducción a la herramienta

JIRA es una aplicación basada en web para el seguimiento de errores, incidencias y para la gestión operativa de proyectos. Jira también se utiliza en áreas no técnicas para la administración de tareas. La herramienta fue desarrollada por la empresa australiana Atlassian. Inicialmente, Jira se utilizó para el desarrollo de software, sirviendo de apoyo para la gestión de requisitos, seguimiento del estatus y más tarde para el seguimiento de errores.

Como dato curioso, el nombre se deriva del nombre japonés para Godzilla, Gojira. Los desarrolladores de JIRA querían que el término estuviera relacionado con Bugzilla para concluir con Gojira.

Al ser una aplicación web no necesita la instalación de ningún componente adicional. Los componentes más importantes de la arquitectura son los mostrados en la figura 23.

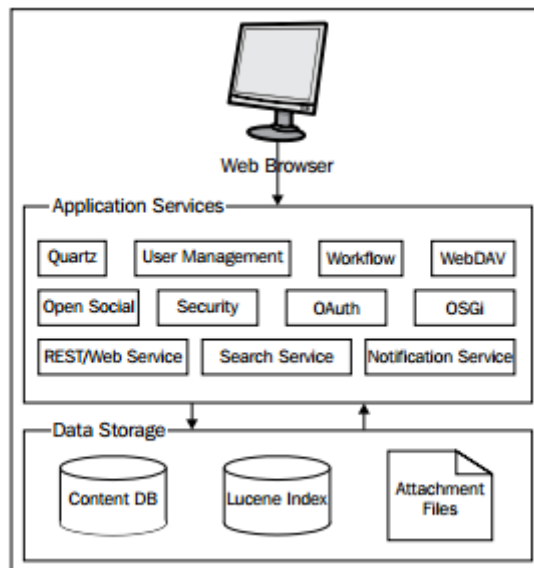


Figura 23 Arquitectura de Jira, de (Jira Essentials)

1.4.3.3 Front Office de Jira

En esta sección, vamos a echar un vistazo al dashboard de Jira para conocer las opciones que nos ofrece, y para ver cómo nos presenta la información en las distintas pantallas.

La página principal por defecto de la aplicación se muestra en la Figura 24.

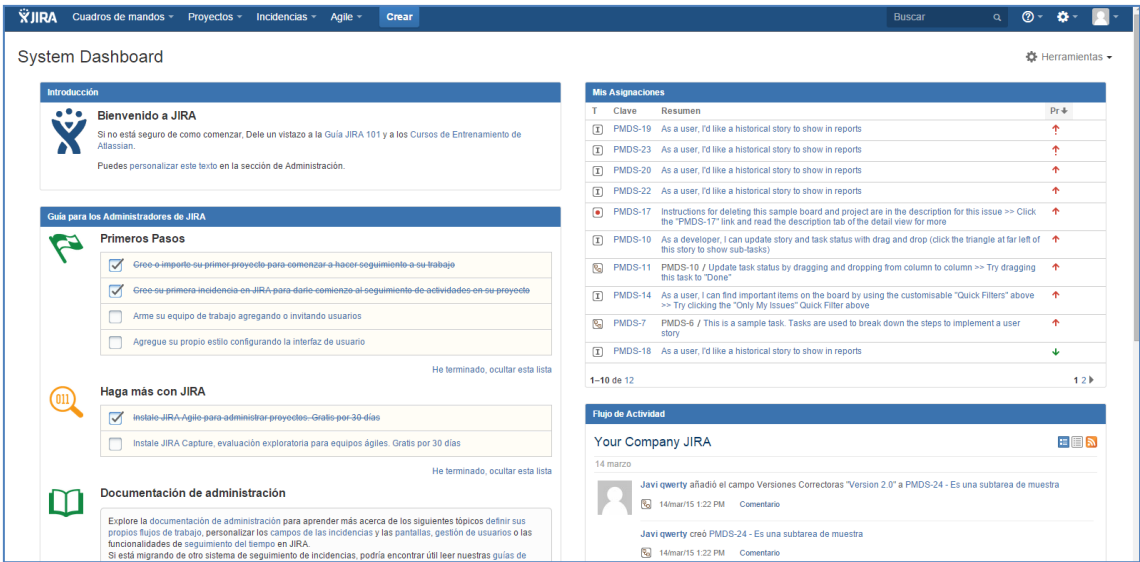


Figura 24 Página principal de Jira

En la parte superior de la imagen podemos ver los distintos menús a los que podemos acceder; Cuadros de mando, Proyectos, Incidencias y el menú para administrar los proyectos ágiles.

Las distintas secciones se explican a continuación:

- Menú Cuadros de mando

Esta opción permite administrar los cuadros de mando del sistema. Se puede crear nuevos cuadros de mando, exportar, importar o modificar los existentes.

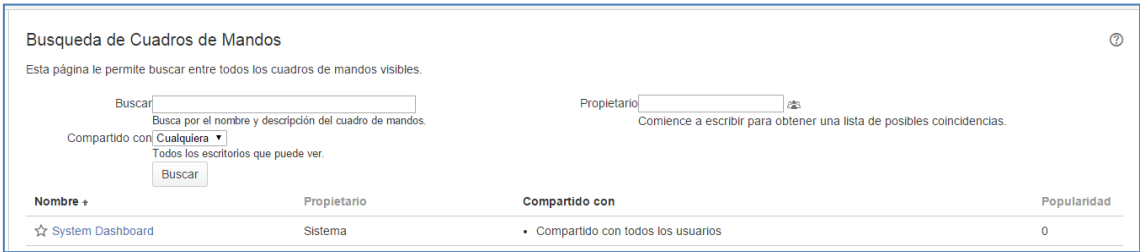


Figura 25 Búsqueda de cuadros de mando

- Menú Proyectos

En este menú podemos navegar por todos los proyectos existentes en el sistema o crear uno nuevo.



Navegar por los Proyectos			
Proyectos recientes	Todos los proyectos		
Todos los proyectos	Proyecto	Clave	Lider del Proyecto
	 Proyecto muestra de Scrum	PMDS	Javi qwerty
	 Proyecto Prueba	PRPRUEBA	Javi qwerty

Figura 26 Índice de proyectos existentes en el sistema

Al pinchar sobre un proyecto accedemos a la página principal del mismo. En la parte izquierda de la figura 27 podemos ver el resumen del proyecto, las versiones y un gráfico con las incidencias.



Figura 27 Detalle de un proyecto en Jira

En la parte derecha de la figura, podemos ver el flujo de actividad del proyecto, es decir, todas las acciones que se realizan sobre el proyecto junto con la información de registro del evento.

- Menú incidencias

Dicho menú, contiene toda la información necesaria para administrar las incidencias del proyecto. Dependiendo de nuestro rol en el proyecto, tendremos unos permisos asignados que nos permitirán acceder a un conjunto distinto de incidencias.

Como vemos en la figura 28, la pantalla de gestión de incidencias es muy intuitiva

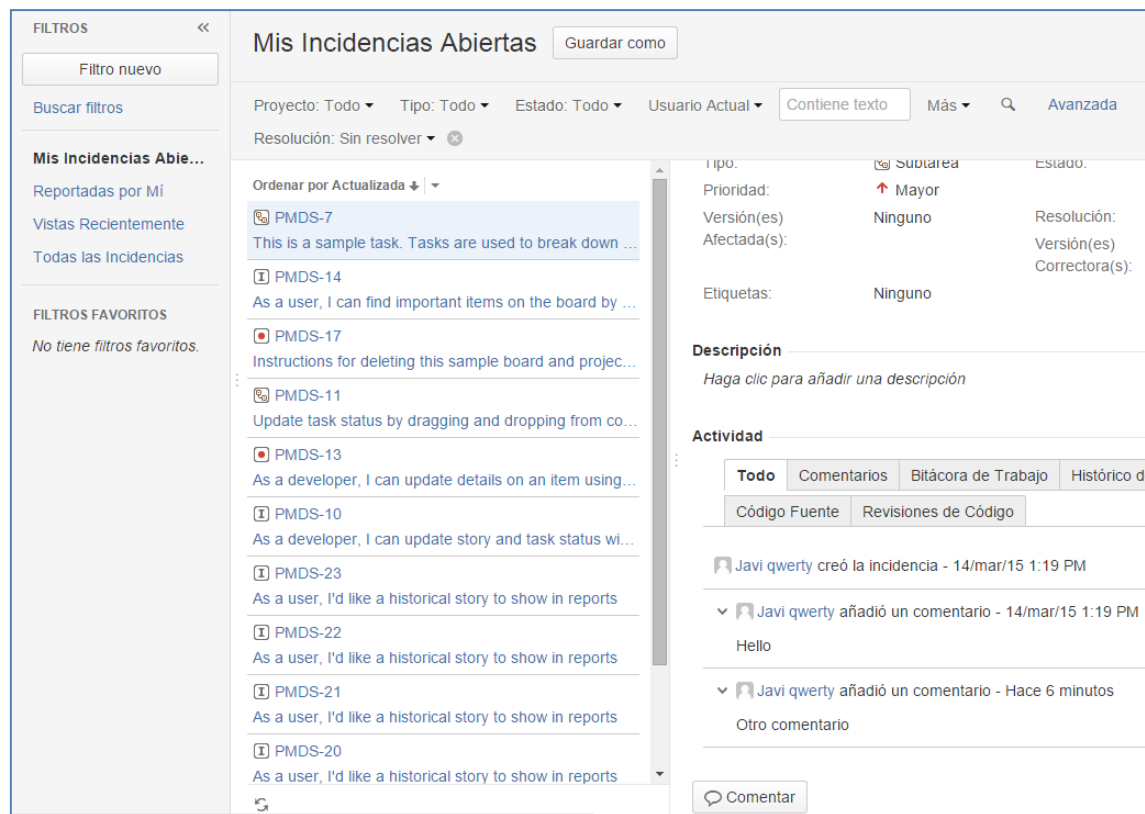


Figura 28 Incidencias asignadas a un usuario

En el centro de la imagen podemos ver todas las incidencias ordenadas por fecha de actualización, y en el lado derecho todas las acciones posibles sobre cada incidencia. Algunas opciones de las incidencias como *Código Fuente*, requieren de la instalación previa de otras herramientas adicionales. Estas herramientas son:

- *Stash*: Se utiliza para administrar repositorios Git
- *Bitbucket*: Utilizado para almacenamiento de repositorios Git
- *FishEye*: Se utiliza para administrar y explorar todos los cambios realizados en los repositorios de código fuente.

- Menú Agile

Por último, tenemos el menú para administrar proyectos ágiles. Tiene dos modos principales, Scrum y Kanban. Scrum es, posiblemente, la metodología de desarrollo ágil más conocida y usada actualmente, mientras que Kanban permite crear tareas en función de restricciones.

Jira permite crear pizarras para Scrum o para Kanban donde podemos introducir toda la información necesaria sobre la iteración. En la figura 29 vemos un ejemplo de SCRUM.

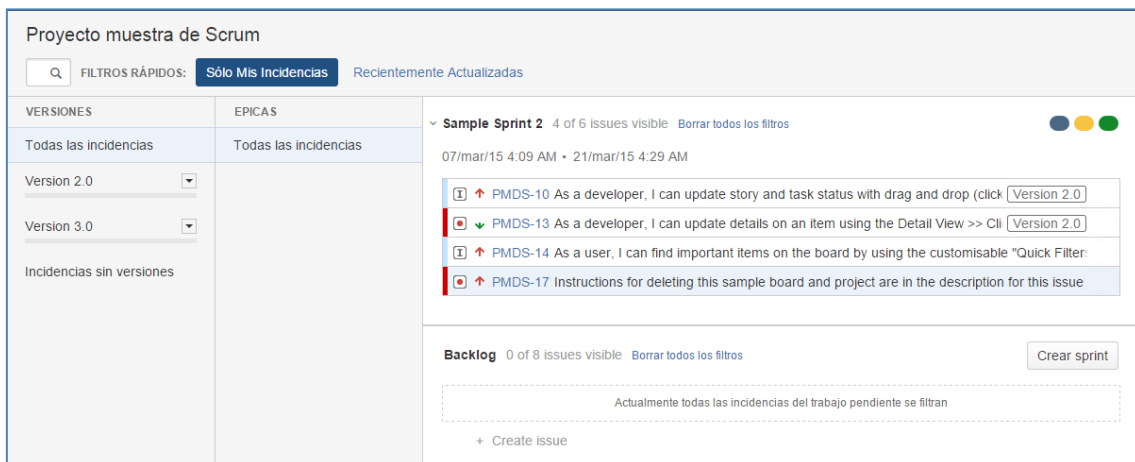


Figura 29 Pizarra SCRUM

1.4.3.4 Back Office de Jira

En esta sección vamos a ver analizar técnicamente el funcionamiento de Jira.

Jira es una aplicación escrita en Java. Se implementa como un archivo WAR estándar en un contenedor servlet como Tomcat. En la figura 30 vemos toda la arquitectura subyacente a Jira.

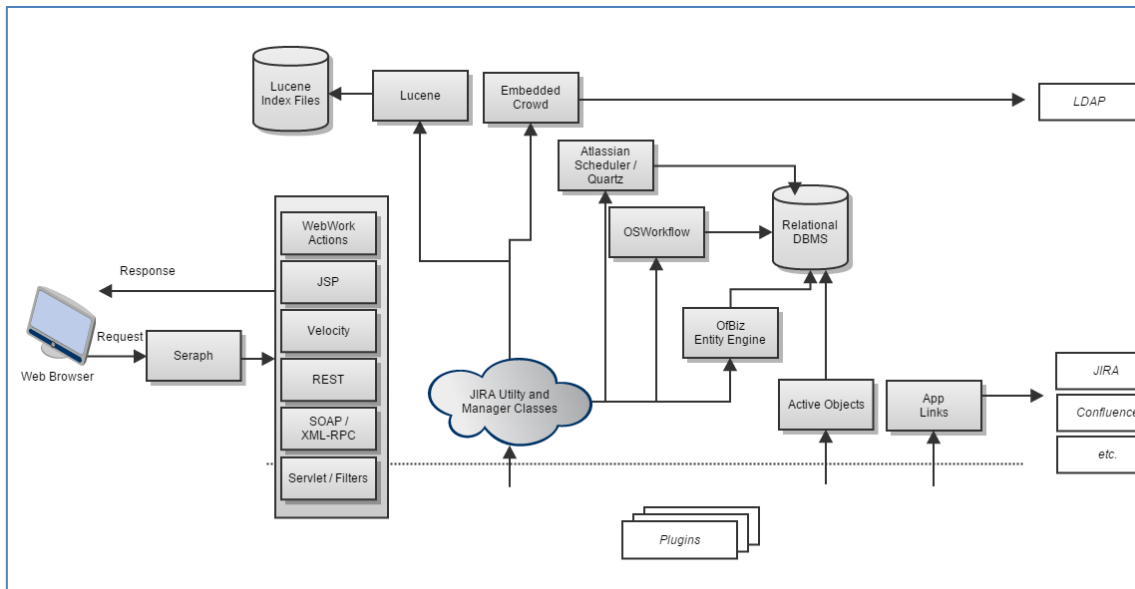


Figura 30 Arquitectura subyacente a Jira²

Vamos a ver más en detalle algunas características del esquema anterior.

- WebWork

Dado que Jira es una aplicación web, los usuarios interactúan con Jira utilizando un navegador. Jira utiliza **WebWork 1** para procesar las peticiones de los usuarios. WebWork 1 es un framework basado en el modelo vista controlador (MVC), similar a Struts. Cada solicitud es manejada por una acción WebWork que generalmente utiliza otros objetos.

Jira utiliza JSP para la capa de presentación. Por tanto, la mayoría del código HTML como respuesta a las solicitudes es generado por JSP.

² Fuente: <https://developer.atlassian.com/>

- Seraph

Prácticamente, todas las autenticaciones en Jira se realizan a través de Seraph, un **framework open source** de Atlassian. El objetivo de Seraph es proporcionar un sistema de autenticación simple, extensible y que pueda ser utilizado en cualquier servidor de aplicaciones.

Seraph se implementa como un servlet de filtro. Su único trabajo es, dado una petición web, asociarla a un usuario en particular. Soporta multitud de protocolos de autenticación, incluyendo la autenticación básica HTTP, autenticación basada en formularios, y búsqueda de credenciales almacenados en una sesión de usuario, como cookies.

Seraph no realiza por sí mismo la gestión de usuarios, si no que comprueba la petición entrante y delega las funciones a la gestión de usuarios de Jira, llamado Jira: Embedded Crowd.

Otra funcionalidad muy importante de Seraph, es administrar que sólo los usuarios con permisos de administrador puedan realizar tareas de administración. Estas acciones WebWork son accesibles a través del navegador mediante URLs que comiencen por */admin*.

- Embedded Crowd

Crowd es una herramienta open source para la gestión de identidades y autenticación SSO, Single Sign On. Provee las siguientes funcionalidades:

- Almacenamiento de usuarios y grupos en la base de datos.
- Almacenamiento de a que grupos pertenecen los usuarios.
- Autenticación de usuarios. Comprueba que la contraseña introducida coincide con la almacenada.
- Provee una API para la administración de usuarios y grupos. Creación, modificación y asignación de usuarios a los grupos.
- Permite a Jira conectarse con sistemas externos para reunir información sobre las credenciales de los usuarios. Por ejemplo, mediante el acceso a un LDAP.
- Almacena una copia de cualquier información externa en el almacenamiento local para que se procesen más rápidas las peticiones y sincroniza las copia con la fuente original.

- PropertySet

Jira también usa JIRA: PropertySet para almacenar las preferencias de los usuarios. Se almacenan en forma de parejas de clave/valor de una identidad con único identificador.

En Jira, las preferencias incluyen:

- Si el usuario prefiere recibir en formato HTML o texto los e-mails.
- Número de problemas que se muestran por defecto en el navegador.
- Idioma local del usuario
- Si el usuario desea recibir las actualizaciones que se produzcan en sus incidencias

- Jira utiliza y administra clases en lenguaje Java

La lógica de la aplicación se implementa en cientos de clases Java. Las clases pueden ser simplemente clases de utilidad u objetos de gestión.

Los objetos de gestión en Jira, suelen tener un objetivo específico, como por ejemplo, la gestión de versiones o de idiomas.

Los objetos de gestión utilizan multitud de dependencias externas. Muchas de ellas de código abierto y otras desarrolladas por Atlassian, y compartidas por multitud de sus productos.

1.4.3.5 Base de datos

En el último apartado sobre Jira vamos a analizar la base de datos. En primer lugar vamos a ver las tablas de la base de datos y a continuación, vamos a analizar algunos aspectos importantes como el acceso mediante REST.

- Estructura de la base de datos

Debido al tamaño de la base de datos, no vamos a mostrar todas las tablas y relaciones que tiene, si no que vamos a mostrar algunas de las tablas más importantes. Para consultar la base de datos completa se puede acceder a la documentación oficial de Jira.

A continuación mostramos algunas tablas:

- Tabla jiraissue

En esta tabla se guardan todos los problemas registrados por los usuarios:

Campo	Tipo
ID	Decimal(18,0)
Pkey	Varchar(255)
PROJECT	Decimal(18,0)
REPORTER	Varchar(255)
ASSIGNEE	Varchar(255)
Issuetype	Varchar(255)
SUMMARY	Varchar(255)
DESCRIPTION	Longtext
ENVIRONMENT	LongText
PRIORITY	Varchar(255)
RESOLUTION	Varchar(255)
Issuestatus	Varchar(255)
CREATED	Datetime
UPDATED	Datetime
DUE DATE	Datetime
RESOLUTIONDATE	Datetime
VOTES	Decimal(18,0)
WATCHES	Decimal(18,0)
TIMEORIGINALESTIMATE	Decimal(18,0)
TIMEESTIMATE	Decimal(18,0)
TIMESPENT	Decimal(18,0)
WORKFLOW_ID	Decimal(18,0)
SECURITY	Decimal(18,0)
FIXFOR	Decimal(18,0)
COMPONENT	Decimal(18,0)

Tabla 10 Tabla jiraissue

➤ Tabla user_details

Dado que cada problema tiene diferentes versiones/componentes, hay una tabla de unión entre la tabla jiraissue y la tabla version/components.

Campo	Tipo
SOURCE_NODE_ID	Decimal(18,0)
SOURCE_NODE_ENTITY	Varchar(60)
SINK_NODE_ID	Decimal(18,0)
SINK_NODE_ENTITY	Varchar(60)
ASSOCIATION_TYPE	Varchar(60)
SEQUENCE	Decimal(18,0)

Tabla 11 Tabla de unión entre jiraissue y version/components

➤ Tabla issue Links

La siguiente tabla enlaza dos problemas juntos, y guardar el tipo de problema.

Campo	Tipo
ID	Decimal(18,0)
LINKTYPE	Decimal(18,0)
SOURCE	Decimal(18,0)
DESTINATION	Decimal(18,0)
SEQUENCE	Decimal(18,0)

Tabla 12 Tabla issue links

➤ Tabla Subtasks

La última tabla guarda los campos personalizados.

Campo	Tipo
ID	Decimal(18,0)
ISSUE	Decimal(18,0)
CUSTOMFIELD	Decimal(18,0)
PARENTKEY	Varchar(255)
STRINGVALUE	Varchar(255)
NUMBERVALUE	Decimal(18,6)
TEXTVALUE	Longtext
DATEVALUE	Datetime
VALUETYPE	Varchar(255)

Tabla 13 Tabla subtasks

- Jira Rest API

Al igual que SonarQube, con Jira se puede obtener información de la base de datos a través de REST. Jira usa el plugin de Atlassian para REST para implementar el API de Jira.

El formato de todas las peticiones REST para Jira es el siguiente:

http://hostname/rest/<api-name>/<api-version>/<resource-name>

1.4.3.6 Resumen

Vamos a acabar este capítulo con un resumen de la herramienta.

En primera lugar podemos ver las características generales de Jira en la tabla 14.

CARACTERÍSTICAS	VALOR
Nombre	Jira
Año primera versión	2004
Año última versión	2013
Licencia	Atlassian
Software Libre/Pago	Pago
Web	https://es.atlassian.com

Tabla 14 Características generales Jira

En la tabla 15 vemos las características técnicas de Jira. Destaca la variedad de áreas en las que es de utilidad.

CARACTERÍSTICAS	VALOR
Lenguaje utilizado para su desarrollo	Java
Soporte profesional	Atlassian
Características funcionales generales	Adquisición de datos, presentación de la información
Características funcionales particulares	Seguimiento de errores, seguimiento de incidencias, gestión de proyectos

Tabla 15 Características técnicas Jira

Por último las características referentes a la base de datos se muestran en la tabla 16.

CARACTERÍSTICAS	VALOR
Bases de datos soportadas	Microsoft SQL Server, MySQL, Oracle, PostgreSQL, HSQLDB
Acceso a la base de datos	Mediante REST

Tabla 16 Características sobre la base de datos Jira

1.4.4 MetricsGrimoire

1.4.4.1 ¿Por qué MetricsGrimoire?

Hemos elegido esta herramienta por los siguientes motivos:

- Es un conjunto de herramientas que trabaja de una forma diferente a SonarQube y a Jira. MetricsGrimoire trabaja con repositorios externos, mientras que las otras dos herramientas tienen su propio repositorio.
- Es un proyecto que comenzó en una universidad Española.
- Es una herramienta menos conocida que las anteriores, lo que permite tener un punto de vista diferente.
- Era una herramienta que no conocíamos, y al realizar un análisis sobre ella nos permite adquirir nuevos conocimientos.

1.4.4.2 Introducción a la herramienta

Dicho conjunto de herramientas tiene su origen en la universidad Rey Juan Carlos, con el nombre de *Libresoft Tools*. Poco a poco, empezó a crecer una comunidad alrededor de estas herramientas, y varios de los creadores de estas herramientas fundaron una empresa, Bitergia, para dar soporte profesional. El proyecto fue evolucionando hasta adquirir el nombre de MetricsGrimoire.

Como hemos dicho anteriormente, MetricsGrimoire está compuesto por un conjunto de herramientas que se detallan a continuación:

- CVSanaly

Recupera y organiza información de los sistemas de control de versiones. Actualmente soporta repositorios CVS, Subversion y Git.

Repositorio en GitHub: <https://github.com/MetricsGrimoire/CVSanaly>

- Bicho

Recupera y organiza información de los sistemas de seguimiento de problemas. En la actualidad soporta Bugzilla, Jira y los repositorios de código Allura, GitHub, Google Code y LaunchPad.

Repositorio en GitHub: <https://github.com/MetricsGrimoire/Bicho>

- MailingListStats

Recupera y organiza información de los correos. Soporta archivos locales en formato mbox o archivos web accesibles.

Repositorio en GitHub: <https://github.com/MetricsGrimoire/MailingListStats>

- Repository Handler

Librería para manejar repositorios de código fuente.

Repositorio en GitHub: <https://github.com/MetricsGrimoire/RepositoryHandler>

- CMetrics

Extrae algunas métricas de código escrito en C.

Repositorio en GitHub: <https://github.com/MetricsGrimoire/CMetrics>

- Sibyl

Esta herramienta extrae información de páginas web que siguen un formato de *question-and-answer* y almacena la información en la base de datos. Actualmente soporta AskBot.

Repositorio en GitHub: <https://github.com/MetricsGrimoire/Sibyl>

Hoy en día, de todas las herramientas descritas anteriormente, CVSanaly y Bicho son las que tienen un desarrollo y una aceptación mayor, por tanto, nuestro estudio se centrará en esas dos herramientas.

Por otro lado, hay que hablar sobre **vizGrimoire**. vizGrimoire es un proyecto para analizar y presentar información de desarrollos de software. Actualmente se centra en la información extraída por las herramientas CVSanaly, Bicho y MailingStats.

Podríamos decir que vizGrimoire es la presentación de los datos que extraemos y almacenamos con MetricsGrimore.

1.4.4.3 Front Office de MetricsGrimoire

Como anteriormente hemos citado, MetricsGrimoire no tiene un front-office propio, si no que utiliza vizGrimoire para presentar la información. En la figura 31 vemos un esquema de cómo funcionan estas herramientas de forma conjunta.

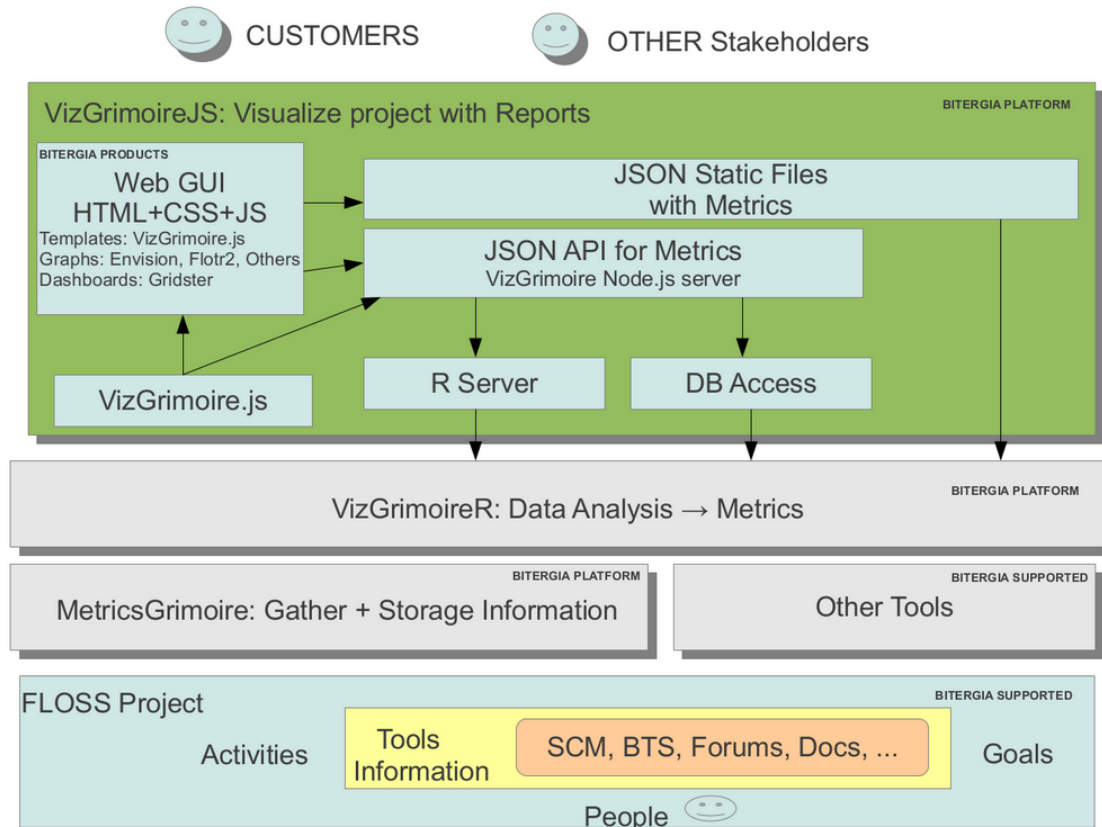


Figura 31 Flujo de trabajo entre MetricsGrimoire y vizGrimoire, de (<http://vizgrimoire.bitergia.org/>)

Como podemos apreciar en la imagen, las herramientas de MetricsGrimoire extraen la información y la almacenan, para posteriormente, ser analizada y mostrada por vizGrimoire.

vizGrimoire se ha utilizado para analizar proyectos como eclipse, Red Hat u OpenStack. A continuación se muestran algunos ejemplos de dashboard para los proyectos citados.

- Eclipse Foundation

En estos dashboard podemos ver las estadísticas referentes a cuantos commits se han realizado, el número de desarrolladores o información referentes a los correos electrónicos.

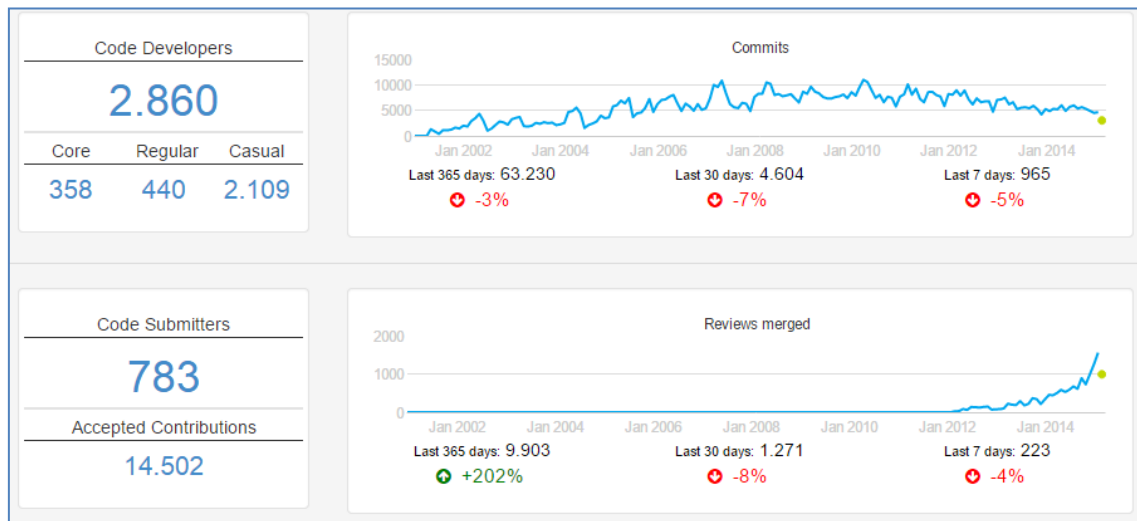


Figura 32 Dashboard de vizGrimoire para Eclipse (I), de (<http://bitergia.com/>)

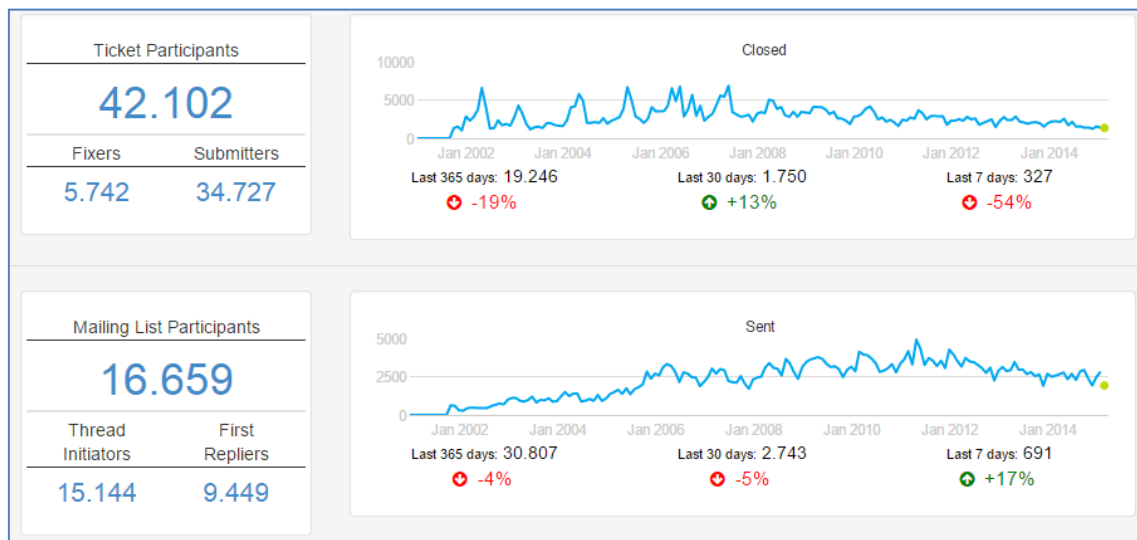


Figura 33 Dashboard de vizGrimoire para Eclipse (II) , de (<http://bitergia.com/>)

También existe la opción de ver más en detalle la información referente a cada apartado. Por ejemplo, en la figura 34 vemos información en detalle sobre los desarrolladores.

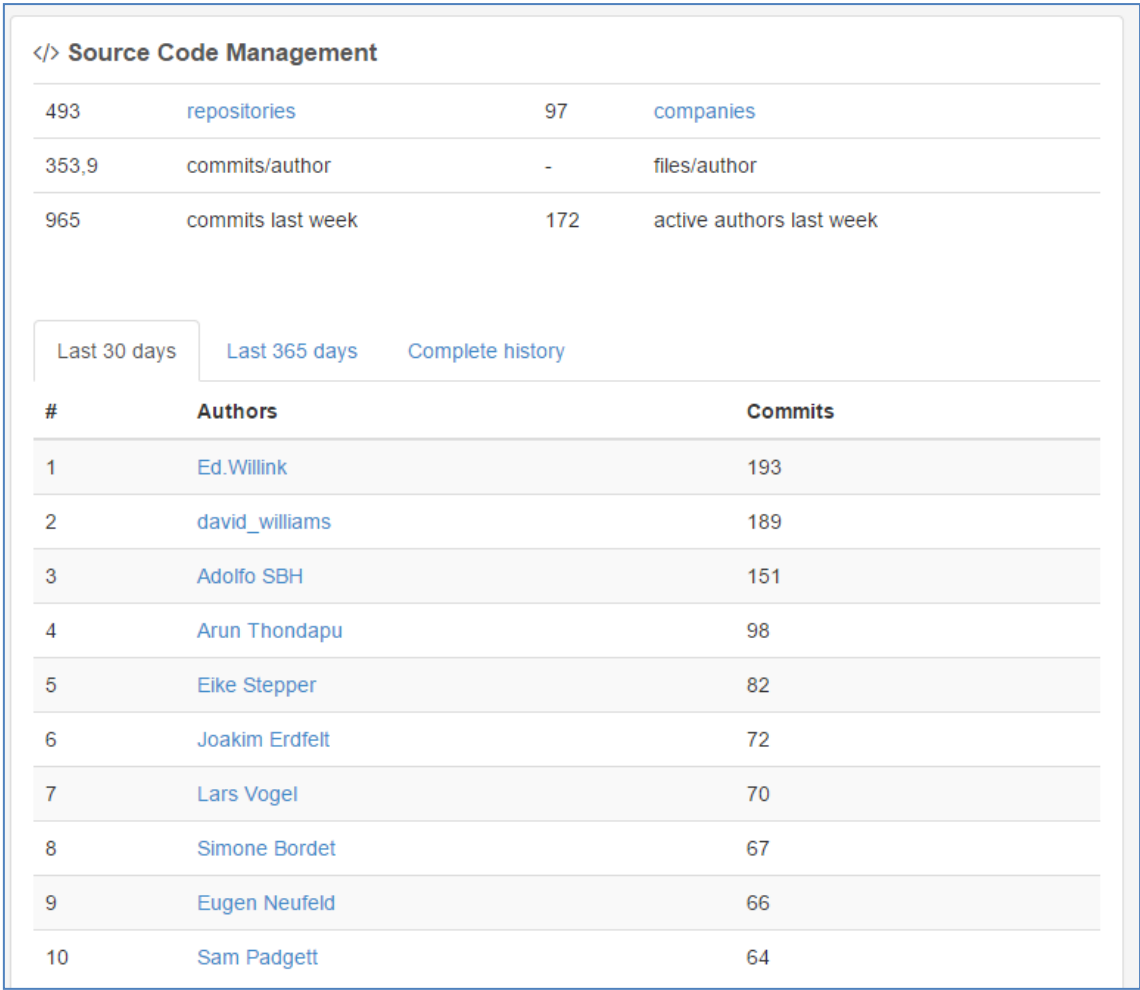


Figura 34 Dashboard de vizGrimoire en detalle para Eclipse (I) , de (<http://bitergia.com/>)

Y en la figura 35 podemos ver estadísticas por zonas horarias, y otros gráficos que muestran las líneas añadidas contra las líneas eliminadas.

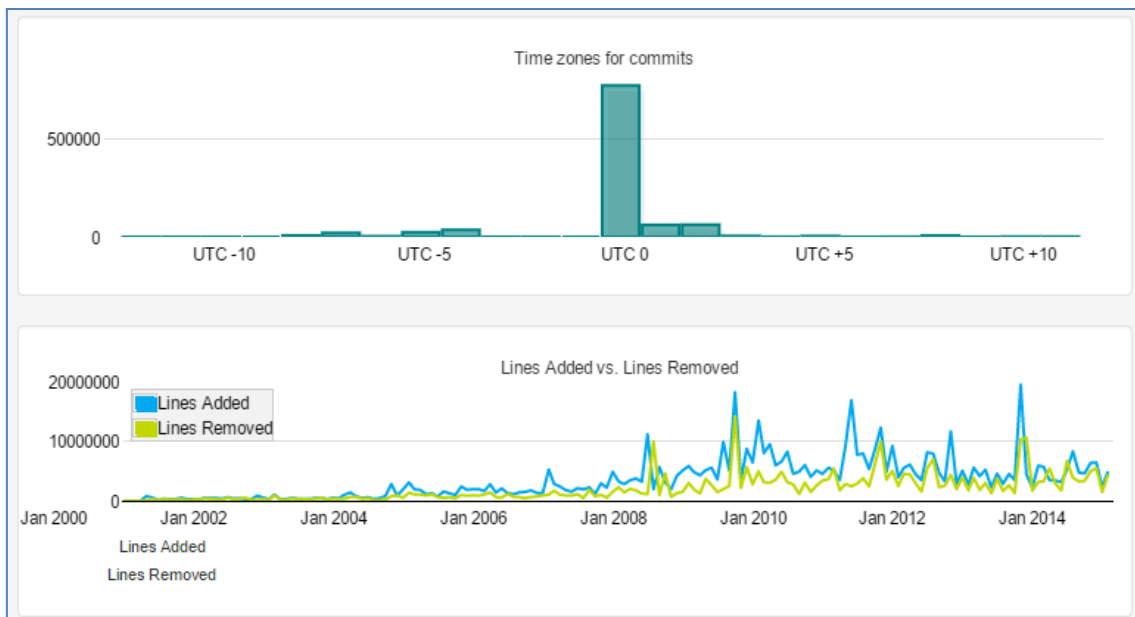


Figura 35 Dashboard de vizGrimoire en detalle para Eclipse (II) , de (<http://bitergia.com/>)

- OpenStack

En este proyecto podemos visualizar las mismas métricas que en el anterior, pero además, podemos ver información referente al uso del canal IRC.

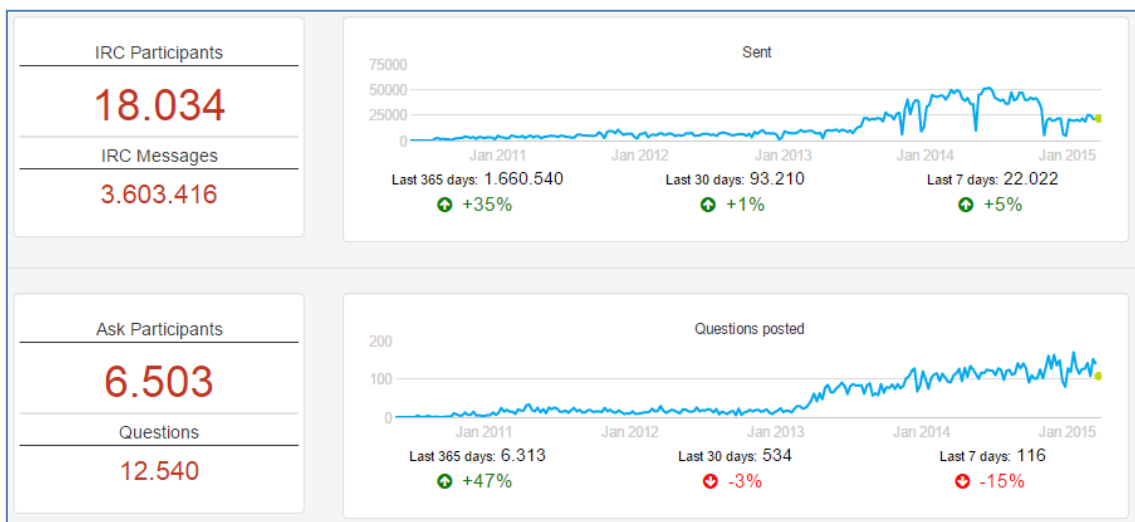


Figura 36 Dashboard de vizGrimoire para openStack (I) , de (<http://bitergia.com/>)

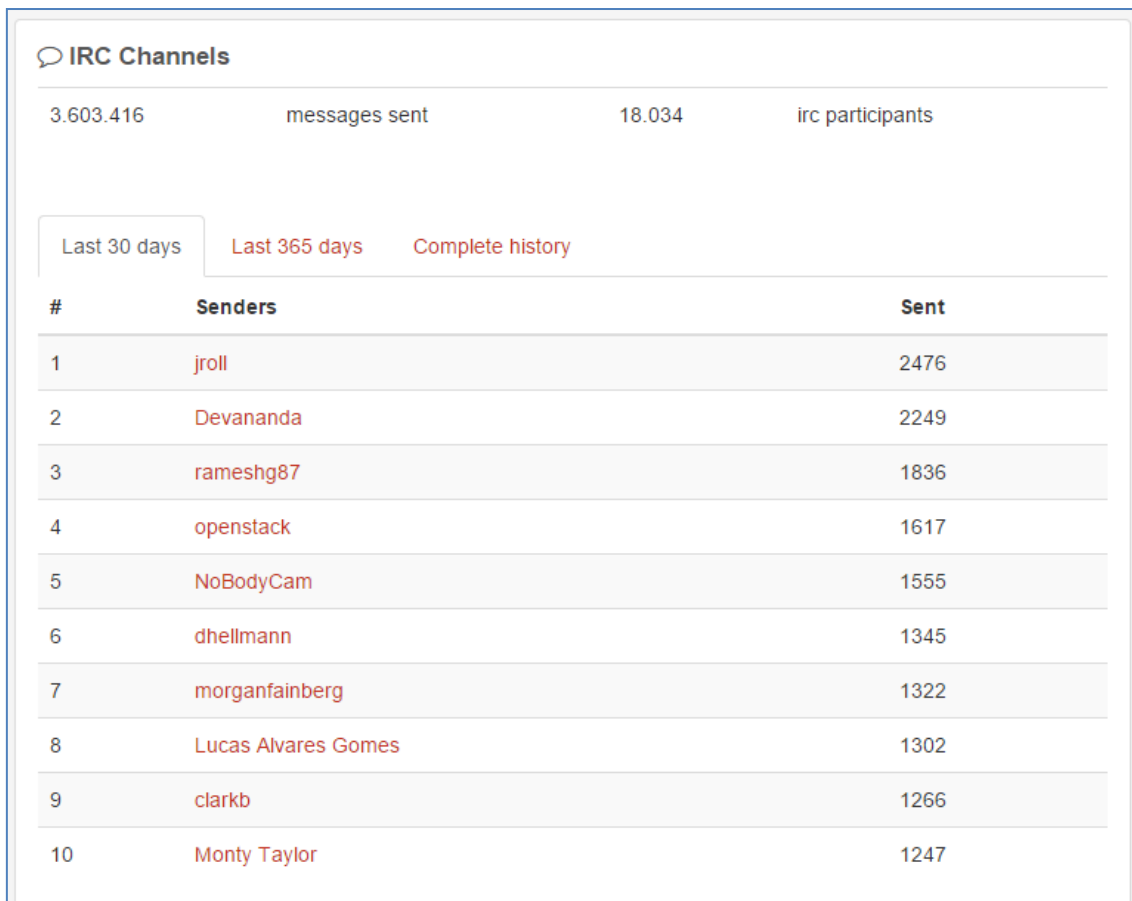


Figura 37 Dashboard de vizGrimoire para openStack (II) , de (<http://bitergia.com/>)

1.4.4.4 Back office y base de datos de MetricsGrimoire

En esta sección vamos a ver las tablas que componen la base de datos de las dos herramientas que estamos analizando de MetricsGrimoire: CVSAly y Bicho.

Además, veremos algunas consultas de ejemplo donde para obtener la información que hemos recogido de los repositorios. A diferencia de las dos herramientas analizadas anteriormente, en estas no tenemos que acceder mediante REST y podemos ejecutar consultas contra la base de datos.

- CVSAly

A continuación se muestra el listado de tablas:

- **Scmlog:** Esta tabla contiene información general sobre los commits. Cada commit en el repositorio se representa por un registro en la tabla scmlog.

Campo	Valor
Id	INTEGER
Repository id	INTEGER
Author id	INTEGER
Committer id	INTEGER
Rev	MEDIUMTEXT
Date	DATETIME
Message	LONGTEXT
Composed rev	BOOL

Tabla 17 Tabla scmlog

- **File types:** Esta tabla contiene un registro por cada tipo de archivo que podría ser encontrado en el repositorio como archivos de documentación de código fuente, imágenes, etc.

Campo	Valor
Id	INTEGER
File_id	INTEGER
Type_id	MEDIUMTEXT

Tabla 18 Tabla scmlog

- **Actions:** Contiene información sobre las diferentes acciones que se realizan en un commit. En sistemas como CVS, donde el commit se limita a un archivo, sólo habrá un registro en la tabla *actions*.

Campo	Valor
Id	INTEGER
Commit_id	INTEGER
File_id	INTEGER
Branch_id	INTEGER
Type	VARCHAR(1)

Tabla 19 Tabla actions

- **Branches:** Contiene las distintas ramas de un repositorio.

Campo	Valor
Id	INTEGER
Name	VARCHAR(255)

Tabla 20 Tabla branches

- **Metrics:** Contiene las distintas métricas de un archivo. Esta tabla añade métricas sobre el código fuente por cada revisión de cada archivo encontrado en el repositorio. Debido a que esta tabla contiene información sobre el código fuente, usa la tabla *FileTypes* para encontrar sólo los archivos de código.

Campo	Valor
Id	INTEGER
File_id	INTEGER
Commit_id	INTEGER
Lang	TINYTEXT
Sloc	INTEGER
Loc	INTEGER
Ncomment	INTEGER
Lcomment	INTEGER
Lblank	INTEGER
Mccabe_min	INTEGER
Nfunctions	INTEGER
Mccabe_max	INTEGER
Mccabe_sum	INTEGER
Mccabe_mean	INTEGER
Mccabe_median	INTEGER
Halstead_length	INTEGER
Halstead_vol	INTEGER
Halstead_level	DOUBLE
Halstead_md	INTEGER

Tabla 21 Tabla Metrics

- **People:** Contiene información sobre la gente que ha trabajado en el repositorio. Contiene el nombre y el email, cuando esté disponible, de la gente involucrada en el repositorio.

Campo	Valor
People_id	INTEGER
Name	VARCHAR(255)
mail	VARCHAR(255)

Tabla 22 Tabla people

- **Repositories:** Contiene URLs de los repositorios analizados.

Campo	Valor
Id	INTEGER
url	VARCHAR(255)
Name	VARCHAR(255)
Type_2	VARCHAR(255)

Tabla 23 Tabla repositories

- **Commits lines:** Contiene información sobre las líneas añadidas y eliminadas por un commit.

Campo	Valor
Id	INTEGER
Commit_id	INTEGER
Added_name	INTEGER

Tabla 24 Tabla commits lines

- **Files:** Contiene información general sobre los archivos encontrados en el repositorio.

Campo	Valor
Id	INTEGER
Repository_id	INTEGER

Tabla 25 Tabla files

- **File copies:** Contiene información general sobre los archivos copiados. Se usa para almacenar adicional información sobre las acciones que involucren más de un archivo. Copias, movimientos y reemplazos son acciones que se llevan a cabo sobre dos o más archivos.

Campo	Valor
Id	INTEGER
From_id	INTEGER
From_commit_id	INTEGER
To_id	INTEGER
Action_id	INTEGER
New_file_name	MEDIUMTEXT

Tabla 26 Tabla file copies

- **Files links:** Contiene información general sobre la tipología de los archivos. La relación entre dos archivos es siempre padre - hijo.

Campo	Valor
Id	INTEGER
File_id	INTEGER
Parent_id	INTEGER
Commit_id	INTEGER

Tabla 27 Tabla file links

- **Tag revisions:** Contiene información general sobre la lista de revisiones apuntando a cada etiqueta.

Campo	Valor
Id	INTEGER
Datasource_id	INTEGER
Commit_id	INTEGER
Tag_id	INTEGER

Tabla 28 Tabla tag revisions

- **Tags:** Contiene información general sobre el nombre de las etiquetas.

Campo	Valor
People_id	INTEGER
Name	VARCHAR(255)

Tabla 29 Tabla tags

- Bicho

- **Attachments:** Esta tabla contiene información general sobre la lista de archivo adjunto.

Campo	Valor
Id	INTEGER
idBug	VARCHAR(128)
Name	VARCHAR(256)
Description	TEXT
Url	VARCHAR(256)

Tabla 30 Tabla attachments

- **Bugs:** Contiene información general sobre la lista de problemas encontrados.

Campo	Valor
idBug	VARCHAR(128)
Summary	TEXT
Description	TEXT
DateSubmitted	VARCHAR(128)
Status	VARCHAR(64)
Resolution	VARCHAR(64)
Priority	VARCHAR(64)
Category	VARCHAR(128)
AssignedTo	VARCHAR(128)
SubmittedBy	VARCHAR(128)
iGroup	VARCHAR(128)

Tabla 31 Tabla Bugs

- **Bugzilla Data:** Contiene más información sobre problemas.

Campo	Valor
idBug	MEDIUMINT(9)
Delta ts	VARCHAR(128)
Reporter accessible	VARCHAR(128)
Cclist accessible	VARCHAR(128)
Classification id	VARCHAR(128)
Classification	VARCHAR(128)
Product	VARCHAR(128)
Component	VARCHAR(128)
Version	VARCHAR(128)
Rep platform	VARCHAR(128)
Op sys	VARCHAR(128)
Bug severity	VARCHAR(128)
Target milestone	VARCHAR(128)
Cc	VARCHAR(128)

Tabla 32 Tabla Bugzilla Data

- **Changes:** Contiene información general sobre la lista de cambios realizados en los problemas.

Campo	Valor
Id	INTEGER(11)
idBug	VARCHAR(128)
Field	VARCHAR(256)
OldValue	VARCHAR(256)
Date	VARCHAR(256)
SubmittedBy	VARCHAR(256)

Tabla 33 Tabla Changes

- **Comments:** Contiene información sobre los comentarios en los problemas.

Campo	Valor
Id	INTEGER(11)
idBug	VARCHAR(128)
DateSubmitted	VARCHAR(128)
SubmittedBy	VARCHAR(128)
Comment	TEXT

Tabla 34 Tabla Comments

- **GeneralInfo:** Esta tabla contiene información general sobre el tracker recuperado.

Campo	Valor
Id	INTEGER(11)
Project	VARCHAR(256)
Url	VARCHAR(256)
Tracker	VARCHAR(256)
Date	VARCHAR(128)

Tabla 35 Tabla General Info

1.4.4.5 Resumen

A continuación podemos ver tres tablas con las características de la herramienta.

En la tabla 36 vemos se detallan las características generales de la herramienta.

CARACTERÍSTICAS	VALOR
Nombre	MetricsGrimoire
Año primera versión	2004
Año última versión	2011
Licencia	LGPL v3
Software Libre/Pago	Libre
Web	https://metricsgrimoire.github.io/

Tabla 36 Características generales de MetricsGrimoire

En la tabla 37 podemos ver las características técnicas de MetricsGrimoire.

CARACTERÍSTICAS	VALOR
Lenguaje utilizado para su desarrollo	Python
Soporte profesional	Bitergia
Características funcionales generales	Adquisición de datos, presentación de la información
Características funcionales particulares	Dashboard de proyectos, analíticas y reportes de proyectos e identificación de buenas / malas prácticas.

Tabla 37 Características técnicas de MetricsGrimoire

Por último, vemos las características sobre la BBDD de MetricsGrimoire.

CARACTERÍSTICAS	VALOR
Bases de datos soportadas	MySQL, PostgreSQL
Acceso a la base de datos	Mediante REST

Tabla 38 Características sobre la base de datos de MetricsGrimoire

1.5 Comparación de las herramientas

En este apartado, vamos a comparar las tres herramientas analizadas anteriormente y así ver las diferencias y lo que pueden aportarnos cada una de ellas.

Empezaremos por las características generales. Aquí la característica más importante es la madurez del proyecto. En la figura 38 vemos una gráfica con los años de desarrollo de cada proyecto.

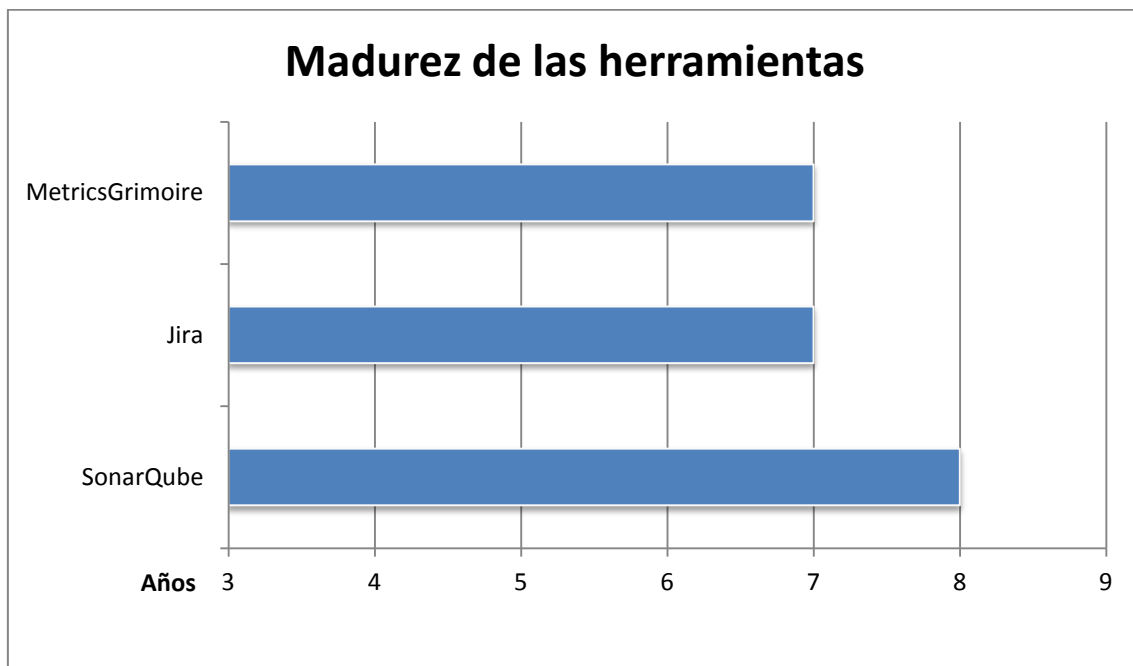


Figura 38 Madurez de las herramientas

No sólo tenemos que tener en cuenta los años de madurez del proyecto, ya que una herramienta con diez años de desarrollo cuya última versión fue liberada hace quince años, es muy probable que se haya quedado anticuada dado que el mundo tecnológico evoluciona a un ritmo muy alto. Por tanto, es importante fijarnos en el año de la última versión de cada herramienta. MetricsGrimoire y Jira publicaron sus últimas versiones en el año 2011, mientras que SonarQube publicó su última versión este mismo año, en 2015.

Cómo vemos, destaca SonarQube en este aspecto tanto en años de madurez, como en la fecha de publicación de la última versión.

Respecto a las otras características generales, vemos que tenemos dos herramientas de uso libre como son SonarQube y MetricsGrimoire, y otra de uso privativo, Jira, aunque es cierto que el coste para uso personal no es muy elevado y ofrece licencias gratuitas para comunidades de desarrollo de software libre.

Vemos ahora las características técnicas de las herramientas. En primer lugar veamos lo que aportan de forma general las herramientas en el aspecto funcional. La tabla 39 muestra esta información.

	ADQUISICIÓN DE DATOS	PRESENTACIÓN DE LA INFORMACIÓN	ANÁLISIS DE LAS MÉTRICAS
MetricsGrimoire	X	X	
Jira		X	
SonarQube	X	X	X

Tabla 39 Comparativa características funcionales generales

Como vemos, las tres herramientas tienen dos de las tres características funcionales en común, mientras que SonarQube además, realiza un análisis automático de los datos que recopila.

MetricsGrimoire recopila información de distintos repositorios y los almacena en la base de datos. A continuación, y haciendo uso de vizGrimoire, muestra distintos gráficos con los datos obtenidos pero no utiliza ninguna métrica, parámetros o metodología para decidir si los datos son una representación positiva o negativa de lo que se está haciendo.

Jira por su parte, tiene un comportamiento parecido al de MetricsGrimoire en el sentido que la información que almacena y utiliza debe ser analizada para tener una visión clara de cómo está evolucionando el proyecto.

SonarQube por su parte, si que realiza ese análisis, y utiliza métricas y métodos como SQALE o el estándar ISO 9126. Además muestra tendencias de cómo van evolucionando los resultados obtenidos, lo que facilita y automatiza el trabajo de decisión y gestión de un proyecto.

Aunque en áreas distintas, en este sentido, SonarQube vuelve a estar un escalón por encima de las otras herramientas.

Por último, vamos a comparar las bases de datos soportadas por cada herramienta. La tabla 41 muestra esta comparación.

	MICROSOFT SQL SERVER	MYSQL	ORACLE	POSTGRESQL
MetricsGrimoire		X		X
Jira	X	X	X	X
SonarQube	X	X	X	X

Tabla 40 Bases de datos soportadas

Las tres herramientas tienen en común el soporte para las dos bases de datos gratuitas, que son además, dos de las bases de datos más utilizadas. Jira y Sonarqube también tienen soporte para SQL Server y Oracle.

En este sentido no nos parece que una herramienta sea superior a otra, dado que todas permiten la utilización de bases de datos ampliamente utilizadas y por tanto no podemos poner una por encima de la otra. El hecho de que dos de las herramientas permitan el uso de

bases de datos de pago es un añadido, y podría ser una ventana respecto a MetricsGrimoire si ésta no tuviera el soporte para bases de datos muy utilizadas, pero no es el caso, dado que MySQL y PostgreSQL son la segunda y la quinta base de datos más utilizada en el mundo.

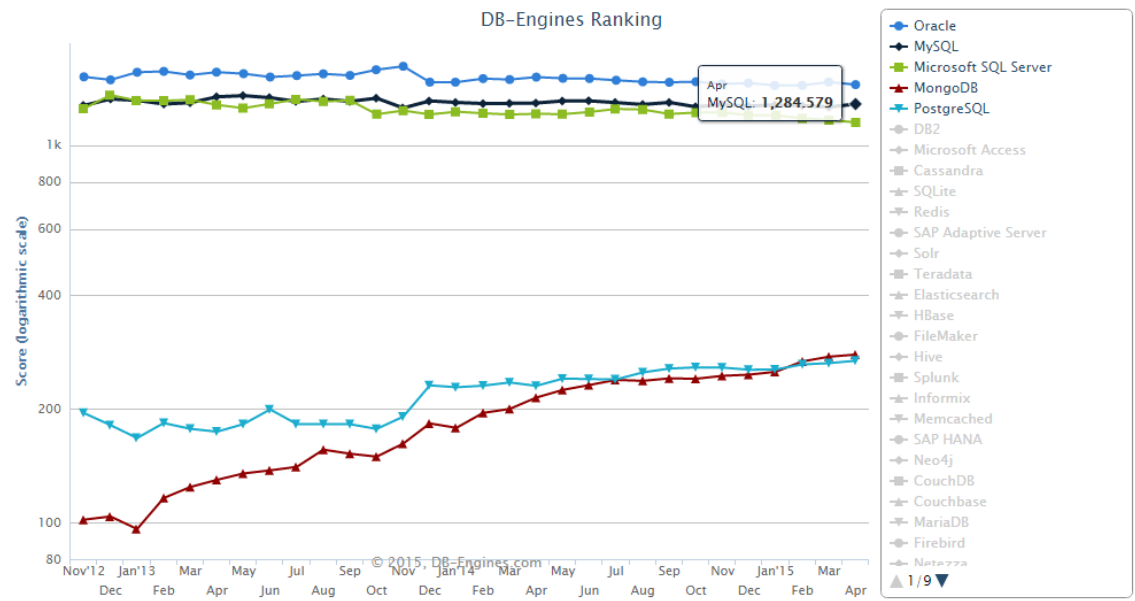


Figura 39 Ranking de BBDD más utilizadas³

³ Fuente: <http://db-engines.com/en/>

2 Implementación de Plug-In

2.1 Introducción

En la segunda parte del trabajo vamos a describir detalladamente cómo construir plugin para SonarQube.

En primera lugar vamos detallar cómo funciona la arquitectura de SonarQube cuando realizamos análisis de los proyectos software, para ver cómo interactúan las distintas partes de la herramienta.

En segundo, y último lugar, explicaremos como implementar un plugin paso a paso, junto con las capturas del plugin que vamos a crear.

El plugin consistirá en dos tablas y dos gráficas. La primera tabla nos mostrará información general de la herramienta, mientras que en la segunda visualizaremos algunas métricas de tamaño del código junto con la explicación detallada de que es lo que cuenta cada métrica.

Por último, podremos ver dos gráficos sobre la información que habremos almacenado del proyecto software analizado.

2.2 Entendiendo la arquitectura de SonarQube

Vamos a comenzar esta sección describiendo cómo funciona la arquitectura de Sonar y el proceso de análisis. En la figura 40 vemos un esquema del flujo que sigue un análisis en SonarQube.

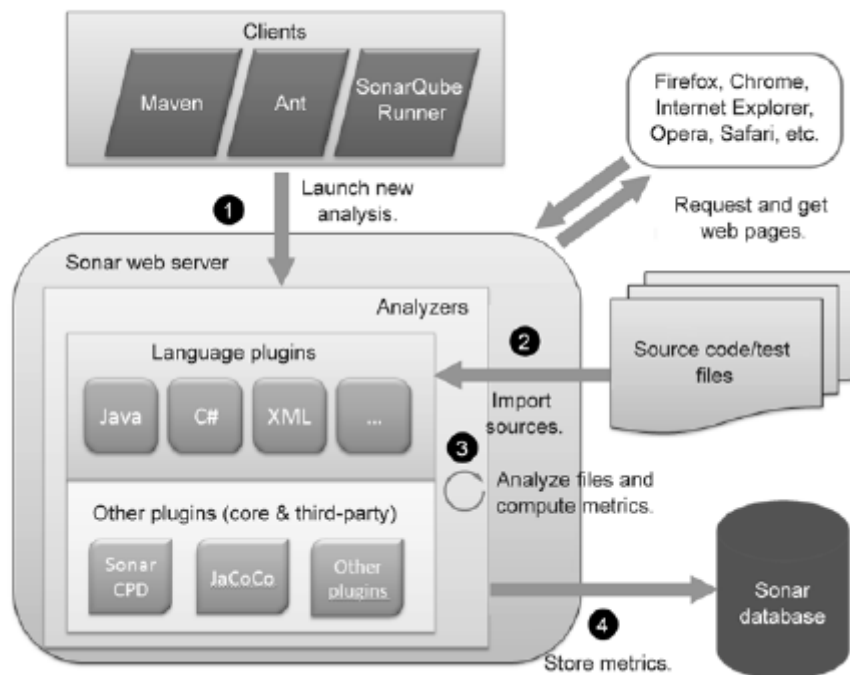


Figura 40 Arquitectura SonarQube, de (SonarQube in Action)

En primer lugar tenemos que lanzar un análisis. Existen tres herramientas principalmente para lanzar éste análisis: Maven, Ant y SonarQube Runner. Éste último es el recomendado por SonarSource, aunque en este estudio hemos optado por utilizar Maven, dado que ofrece otras funcionalidades adicionales, como la creación de la estructura del proyecto.

En segundo lugar, tan pronto como el análisis es lanzado por el cliente, lo primero que hace SonarQube es importar el código fuente y, en caso de que existan, los archivos que contengan los tests. Este proceso es realizado por el plugin apropiado para cada lenguaje. Como hemos dicho en secciones anteriores, Sonar provee soporte, por defecto, para Java pero se puede incluir cualquier lenguaje con los plug-in correctos.

El siguiente paso, es analizar el código importando anteriormente. Los analizadores pueden ser agrupados en dos categorías: sensores y decoradores. En primer lugar se ejecutan los sensores, que se encargan de crear nuevas medidas, aplicar sus métricas para cada recurso y almacenarlas en la base de datos. Cuando todos los sensores han finalizado su trabajo, los decoradores se ejecutan. Su trabajo consiste en agregar métricas de alto nivel a partir de las métricas computadas por los sensores.

Por último, cuando todos los analizadores han terminado su trabajo, se crea una nueva *snapshot* en la base de datos y todas las nuevas métricas computadas se conectan con las que ya habían, creando así las tendencias, para saber si se ha mejorado o se ha empeorado desde el análisis anterior.

Ahora sólo queda ver la información utilizando el navegador web favorito de cada uno.

2.3 Creación de un plugin

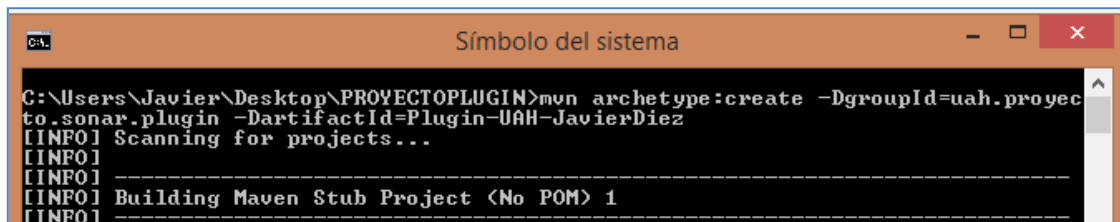
La última sección de este trabajo mostrará cómo crear un plugin para Sonar paso a paso. Para este objetivo, utilizaremos una herramienta adicional: Maven. Maven es una herramienta para la gestión y construcción de proyectos Java, que utiliza un modelo de configuración basado en XML. Maven nos ayudará a crear la estructura de carpetas, automatizando el proceso, a compilar y crear los ejecutables, y por último, nos servirá para lanzar los análisis de los proyectos en SonarQube.

Los pasos que vamos a seguir para la creación del plugin son los siguientes:

1. Configurar el esqueleto del proyecto y las variables necesarias.
2. Describir como configurar y añadir nuevas métricas.
3. Analizar las diferencias entre los dos tipos de analizadores de SonarQube: Sensores y Decoradores.
4. Describir cómo utilizar los sensores y los decoradores
5. Crear un widget para mostrar los resultados en el dashboard de SonarQube.

2.3.1 Creando el proyecto Maven

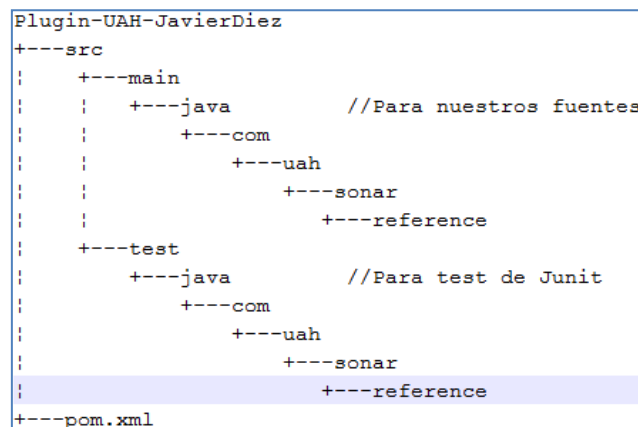
En primer lugar, vamos a crear la estructura de directorios. En la imagen 41 se muestra el comando utilizado para el esqueleto del plugin.



```
C:\Users\Javier\Desktop\PROYECTOPLUGIN>mvn archetype:create -DgroupId=uah.proyecto.sonar.plugin -DartifactId=Plugin-UAH-JavierDiez
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
```

Figura 41 Creación de directorios usando Maven

Con este comando, obtenemos la siguiente estructura de directorios:



```
Plugin-UAH-JavierDiez
+---src
|   +---main
|   |   +---java          //Para nuestros fuentes
|   |   +---com
|   |   +---uah
|   |   +---sonar
|   |   +---reference
|   +---test
|       +---java          //Para test de Junit
|       +---com
|       +---uah
|       +---sonar
|       +---reference
+---pom.xml
```

Figura 42 Estructura de directorios inicial

El siguiente paso, es crear los directorios que faltan y borrar los que no nos son útiles para el plugin. La estructura final aparece en la figura 43

```
plugin-UAH-JavierDiez
+---src
|   +---main
|       |   +---java           //Para nuestros fuentes
|           |   +---com
|               |   +---uah
|                   +---sonar
|                       +---reference //Aquí van las clases para describir el plugin y sus métricas
|                           +---batch //Aquí metemos los sensores y decoradores
|                               |
|                               |
|                               |
|                               +---ui      //Todo lo relacionado con el plugin
+---resources
    +---widget          //Ficheros Ruby para la presentación de la información
+---pom.xml
```

Figura 43 Estructura de directorios fina

Cabe destacar, que la estructura de directorios mostrada en la figura 43, es la estructura de nuestro plugin pero en otros plugin puede ser diferente. Por ejemplo, es muy común que bajo el directorio de *resources* haya otra carpeta con ficheros de configuración, pero en nuestro caso, no la hemos añadido ya que no se va a utilizar.

Vamos a añadir los ficheros java y ruby necesarios. La figura 44 muestra donde y qué ficheros se añaden.

```

plugin-UAH-JavierDiez
+---src
|   +---main
|   |   +---java           //Para nuestros fuentes
|   |   |   +---com
|   |   |   |   +---uah
|   |   |   |   |   +---sonar
|   |   |   |   |   |   +---reference //Aquí van las clases para describir el plugin y sus métricas
|   |   |   |   |   |   |   +---ProyectoMetrics.java
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   +---ProyectoPlugin.java
|   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   +---batch //Aquí metemos los sensores y decoradores
|   |   |   |   |   |   |   |   |   |   +---ProyectoDecorator.java
|   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   +---ProyectoSensor.java
|   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   +---ui //Todo lo relacionado con el plugin
|   |   |   |   |   |   |   |   |   |   |   |   |   +---ProyectoRubyWidget.java
|   |   +---resources
|   |   |   +---widget           //Ficheros Ruby para la presentación de la información
|   |   |   |   +---proyecto_widget.html.erb
|
+---pom.xml

```

Figura 44 Ficheros y directorios

Con todo esto ya tendríamos la estructura principal del proyecto. Las clases que hemos añadido son:

- *ProyectoMetrics* y *ProyectoPlugin*. Describen la configuración del plugin y sus métricas
- Las dos clases de analizadores: *ProyectoDecorator* y *ProyectoSensor*
- Dos clases que describen el widget: *ProyectoRubyWidget* y *proyecto_widget.html.erb*

Para finalizar, vamos a hablar sobre el archivo de configuración POM.xml. Éste archivo, lo utiliza Maven para configurar y construir nuestro plugin. Es un archivo que contiene toda la información de configuración necesaria como por ejemplo, las dependencias, versiones del software utilizado y soportado (versiones de JUnit, del JDK, de SonarQube) y otros parámetros.

Vamos a comentar los parámetros más importantes de este archivo:

- **groupId:** Es el identificador de grupo de Maven. Debe ser el mismo para todos los plugin que se suban a la comunidad de SonarQube.
- **artifactID:** El identificador de Maven usado para crear el directorio de carpetas. Este será el nombre que tenga el ejecutable al construir el plugin.
- **version:** La versión del plugin. Cuando se está desarrollando suele tener el formato *1.0-SNAPSHOT*. Al final finalizar el desarrollo, o cuando se va a poner una versión en producción se quita sufijo *SNAPSHOT*. De esta manera podemos diferenciar cuando una versión esta todavía siendo desarrollada o cuando está lista para su uso.
- **package:** Define el tipo de empaquetado que tendrá el proyecto. En nuestro caso será *sonar-plugin*. Este parámetro le indica a Maven la lista de acciones que tiene que llevar a cabo en cada etapa de construcción del ejecutable.
- **description:** Se usa para describir el propósito del plugin.
- **name:** Se utiliza para las cabeceras de las licencias. Además, se muestra en el centro de información de SonarQube.
- **pluginkey:** Parámetro muy importante. Se utiliza para construir el nombre de los paquetes, las clases autogeneradas y las carpetas donde se guardan los archivos.

Hay otros muchos parámetros y posibles configuraciones. Se recomienda visitar la última documentación disponible en la página de SonarQube para una lista completa de los parámetros.

2.3.2 Definiendo la configuración disponible del widget

Lo primero que tenemos que definir son los parámetros de configuración del plugin. Nuestro plugin mostrará información general del proyecto analizado, y varias métricas de tamaño del código. No necesitamos ningún parámetro de configuración adicional. Nuestra clase de entrada al plugin se muestra en la figura 45.

```
package com.uah.sonar.reference;

import com.uah.sonar.reference.batch.ProyectoSensor;
import com.uah.sonar.reference.batch.ProyectoDecorator;
import com.uah.sonar.reference.ui.ProyectoRubyWidget;
import org.sonar.api.Properties;
import org.sonar.api.Property;
import org.sonar.api.SonarPlugin;

import java.util.Arrays;
import java.util.List;

/**
 * Clase de entrada para todas las extensiones
 */
@Properties({
    @Property(
        key = ProyectoPlugin.Descripcion,
        name = "Plugin Descripcion",
        description = "Este plugin reporta información general del proyecto analizado, así como una serie de métricas de tamaño del código.",
        defaultValue = "Información reportada:Nombre Proyecto, Fecha análisis, Ruta Proyecto - Métricas: LoC,NCoC,Sentencias"
    )
})

public final class ProyectoPlugin extends SonarPlugin {

    public static final String Descripcion = "descripcion.funcionalidad.plugin";

    // Declaramos todas las extensiones
    public List getExtensions() {
        return Arrays.asList(
            // Definiciones de las métricas que vamos a almacenar
            ProyectoMetrics.class,

            // Batch: Definimos clase Sensor y Decorator
            ProyectoSensor.class, ProyectoDecorator.class,

            // UI: Definimos las clases que compondrán el interfaz gráfico, esto es, el widget que se visualizará
            ProyectoRubyWidget.class);
    }
}
```

Figura 45 Clase de entrada al plugin

Vamos a analizar el código anterior. En primer lugar, veamos las dos anotaciones usadas para describir propiedades: *@Properties* y *@Property*. La primera de ellas se usa para definir un array de propiedades, mientras que la segunda se utiliza para describir cada propiedad individualmente. Los atributos requeridos son *key* y *name*.

Cada propiedad debe tener un identificador único. Es buena idea seguir un patrón de nombres cuando tenemos varias propiedades. En nuestro plugin sólo hemos introducido la descripción de la funcionalidad del plugin.

Los atributos *description* y *name* son utilizados por la interfaz gráfica de SonarQube. Con el atributo *name* damos un nombre identificativo a la propiedad y si fuera necesario, podemos utilizar el atributo *description* para dar una descripción más completa. El atributo *defaultValue* se puede utilizar para dar un valor por defecto.

Por defecto todas las propiedades son de tipo *string*. Se puede cambiar el tipo de una propiedad con el atributo *type*, que esperará un valor de la lista *PropertyType Enum*.

Por último, en cuanto a atributos se refiere, podemos escoger desde qué partes de la aplicación serán visibles las propiedad con los atributos *global*, *project* y *module*. Por defecto, las propiedad están configuradas a nivel global, pero podemos aplicar la configuración que consideremos necesaria.

Vamos a finalizar el análisis del código fuente analizando la clase que extiende *SonarPlugin*. En cada plugin sólo debe de haber una clase que extienda la clase base *SonarPlugin* para que SonarQube lo reconozca y cargue el plugin. Esta clase debe implementar el método *getExtension*. La función del método es devolver todas las clases que participan en el plugin. Esto es así, porque los plugin en SonarQube son un conjunto de clases que extienden otras clases o implementan interfaces definidos por el API de SonarQube. Suelen denominarse **puntos de extensión**. Por tanto, el método *getExtension* permite a SonarQube registrar todos los puntos de extensión que nosotros hemos definido y utilizarlos cuando sea conveniente (durante el análisis, al mostrar el widget, etc.).

La configuración del widget ya está completada, por tanto, pasemos a definir el conjunto de métricas que vayamos a calcular y a guardar en la base de datos.

2.3.3 Describiendo las métricas

Es momento de definir qué vamos a calcular y a almacenar en la base de datos de SonarQube. La figura 46 muestra la clase *ProyectoMetrics*.

```
public final class ProyectoMetrics implements Metrics {

    public static final Metric NOMBRE_PROYECTO = new Metric.Builder("nombre_proyecto", "Nombre_Proyecto", Metric.ValueType.STRING)
        .setDescription("Nombre del proyecto analizado")
        .setQualitative(false)
        .setDomain(CoreMetrics.DOMAIN_GENERAL)
        .create();

    public static final Metric FECHA_PROYECTO = new Metric.Builder("fecha_proyecto", "Fecha_Proyecto", Metric.ValueType.STRING)
        .setDescription("Fecha de análisis")
        .setQualitative(false)
        .setDomain(CoreMetrics.DOMAIN_GENERAL)
        .create();

    public static final Metric RUTA = new Metric.Builder("ruta", "Ruta", Metric.ValueType.STRING)
        .setDescription("Ruta del proyecto analizado en el directorio local")
        .setQualitative(false)
        .setDomain(CoreMetrics.DOMAIN_GENERAL)
        .create();

    public static final Metric DENSIDAD_COMENTARIOS = new Metric.Builder("d_comentarios", "Densidad de Comentarios", Metric.ValueType.PERCENT)
        .setDescription("Densidad de comentarios")
        .setDirection(Metric.DIRECTION_BETTER)
        .setQualitative(true)
        .setDomain(CoreMetrics.DOMAIN_GENERAL)
        .setWorstValue(0.0)
        .setBestValue(100.0)
        .create();

}
```

Figura 46 Clase de definición de métricas

Si echamos un vistazo a la definición de la clase en la documentación vemos que tenemos que implementar la clase abstracta *Metrics*, el cual es otro punto de extensión de SonarQube. A continuación sólo nos queda definir cada nueva métrica como una variable estática final. Tenemos que pasar tres parámetros obligatorios a cada métrica; un identificador único, que debe ser también único respecto al *pluginkey*, un nombre corto, y el tipo de la métrica.

Cada métrica tiene también una serie de atributos, cuya lista se encuentra en la documentación de SonarQube. Vamos a analizar tres atributos que nos parecen importantes:

- **Domain:** Especifica a qué grupo será asignada la métrica. Se puede crear uno propio o asignar la métrica a uno existente como *Complexity*, *Documentation* o *Rules*. Las métricas pertenecientes a un mismo grupo se pueden visualizar en la interfaz gráfica en la misma lista desplegable. La figura 47 muestra nuestras métricas visualizadas en SonarQube.
- **Direction:** Una de las características de SonarQube es el uso de las *tendencias*. Indica si una métrica está empeorando o mejorando. Si asignamos este atributo a *TRUE* estamos indicando que un incremento del valor de esta métrica es una buena tendencia y si le damos valor *FALSE* indicamos lo contrario. Asignado el valor de *TRUE/FALSE* al atributo *qualitative*, indicamos a SonarQube pintar de rojo o verde el icono de tendencia en el widget.
- **BestValue:** Indicamos el mejor valor que puede alcanzar una métrica. El atributo *WorstValue* sirve para lo contrario. Cuando una métrica alcanza el mejor valor posible se oculta en la vista del widget.

Lo último que necesitamos en la clase *ProyectoMetrics*, es llamar al método *getMetrics* devolviendo una lista con todas las métricas creadas.

El siguiente paso consiste en crear una clase Sensor para calcular nuestras métricas.

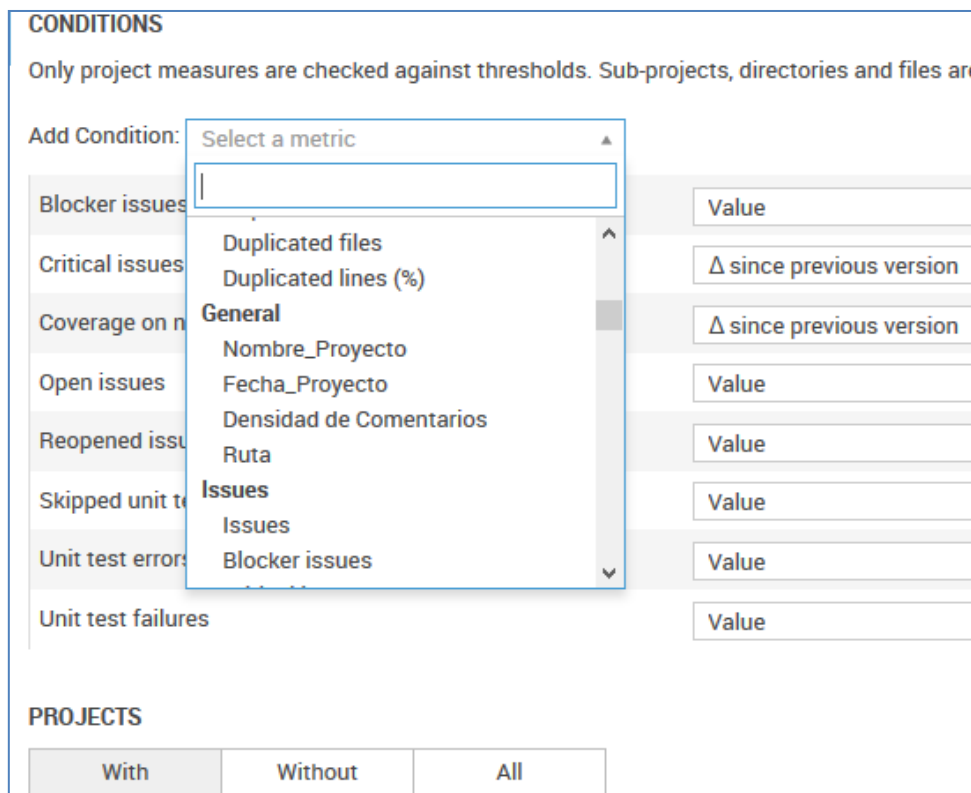


Figura 47 Nuestras métricas pertenecen al mismo grupo

2.3.4 Implementando los analizadores: Clase Sensor

Como hemos comentando anteriormente, existen dos clases de analizadores. Los sensores y los decoradores. La diferencia lógica es que los sensores sólo se ejecutan una vez, mientras que los decoradores se ejecutan una vez por cada recurso existente en el proyecto a analizar. Los **recursos** pueden ser proyectos, directorios, archivos, programas y bloques. En la tabla 41 vemos un resumen con las diferencias entre Sensor y Decorador.

Característica	Sensor	Decorador
Análisis	Se ejecutan una vez a nivel de proyecto	Se ejecutan una vez por cada recurso del proyecto
Orden de ejecución	Se pueden ejecutar en orden aleatorio	Se ejecutan sólo cuando todos los Sensores se hayan ejecutado. Se puede especificar el orden de ejecución entre los decoradores
Métricas / Uso	Se usan para añadir nuevas medidas a SonarQube	Usadas para agregar métricas de alto nivel a partir de las medidas recogidas por el Sensor

Tabla 41 Diferencias entre decorador y sensor

Ahora que ya sabemos la diferencia entre los dos tipos de analizadores hay que decidir cual nos conviene usar. Vamos a utilizar un analizador de cada tipo para ver cómo funcionan y qué funcionalidades nos aportan. En esta sección vamos a analizar el Sensor. En este caso, vamos a analizar la clase por partes para una mayor claridad. La figura 48 muestra la primera parte de la clase.

```
private ProjectClasspath pro;  
private Settings settings;  
  
public ProyectoSensor(ProjectClasspath pro,Settings settings) {  
    this.pro = pro;  
    this.settings = settings;  
}
```

Figura 48 Los componentes del constructor serán inyectados por SonarQube

Lo primero que hacemos es declarar el constructor y todos los componentes que necesitamos. Como vamos a guardar información general del proyecto como el nombre, la ruta y la fecha del análisis, utilizamos los componentes *ProjectClasspath* y *settings*. Estos componentes serán inyectados por el **contenedor IoC** de SonarQube.

Es interesante pararse a analizar qué significa el contenedor IoC de SonarQube. *loc*, *Inversion of control*, es un método de programación en el que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales, en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones. Tradicionalmente, el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones. En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

Siguiendo la definición de la clase, tenemos que definir dos métodos: *shouldExecuteOnProject* y *analyse*. El primero de ellos los vemos en la imagen 78.

```
public boolean shouldExecuteOnProject(Project project) {  
    return true;  
}
```

Figura 49 Método para filtrar en qué tipos de proyectos se ejecuta nuestro sensor

Como vemos, es un método realmente simple. Especificamos para qué tipo de proyectos queremos que se ejecute el sensor. En nuestro caso queremos que se ejecute para todos los proyectos dado que son métricas muy generales, pero podríamos definir que sólo se ejecutara para proyectos escritos en un determinado lenguaje, que existan una serie de parámetros y otros muchos filtros posibles. En nuestra clase Decorador, aplicamos el filtro para que sólo se ejecuten en programas escritos en Java como veremos en la siguiente sección.

En la figura 50, vemos el otro método obligatorio.

```

public void analyse(Project project, SensorContext sensorContext) {
    LOG.info("Empezamos a guardar las métricas");
    LOG.info("Guardamos la métrica NOMBRE_PROYECTO");
    String nombreP = "";
    nombreP = project.getName();

    Measure measure = new Measure(ProyectoMetrics.NOMBRE_PROYECTO, nombreP);
    sensorContext.saveMeasure(measure);

    LOG.info("Guardamos la métrica FECHA_PROYECTO");
    Date date = new Date();
    date = project.getAnalysisDate();

    measure = new Measure(ProyectoMetrics.FECHA_PROYECTO, date.toString());
    sensorContext.saveMeasure(measure);

    LOG.info("Guardamos la métrica RUTA");
    List<File> pr = new ArrayList<File>();
    pr = pro.getElements();
    String f = "";
    f = (pr.get(0)).getPath();

    measure = new Measure(ProyectoMetrics.RUTA, f);
    sensorContext.saveMeasure(measure);

    LOG.info("Todas las métricas han sido guardadas correctamente.");
    LOG.info("Pasamos a decorarlas");
}

```

Figura 50 Método donde se analizan y almacenan las métricas

En el método *analyse* es donde ocurre toda la magia. En este método recibimos la información del código, calculamos métricas y las guardamos en la base de datos.

En nuestra clase, utilizamos dos clases o puntos de extensión diferentes para conseguir la información que necesitamos y almacenarla en la base de datos. Las dos clases son *Project* y *ProjectClassPath*. Para almacenar las métricas utilizamos el método *saveMeasure*. Este método toma dos parámetros: el identificador único de la métrica y el valor que queremos almacenar. El resto de acciones se procesan automáticamente por el API de SonarQube.

2.3.5 Implementando los analizadores: Clase Decorador

Vamos a analizar ahora el segundo tipo de analizador de SonarQube. La clase debe implementar el interfaz *decorator* y los métodos *shouldExecuteOnProject* y *decorate*. El primero de los métodos es exactamente igual y tiene la misma función que en la clase Sensor. La imagen 51 muestra el método. En este caso, sólo se ejecutará para proyectos Java.

```
public boolean shouldExecuteOnProject(Project project) {  
    // Sólo se ejecutará en proyectos JAVA  
    return StringUtils.equals(project.getLanguageKey(), Java.KEY);  
}
```

Figura 51 Filtrado de ejecución de los decoradores

El segundo de los métodos, *decorate*, tiene la misma que el método *analyse* en la clase Sensor. Aquí es donde calculamos las nuevas métricas. La figura 52 muestra el código del método.

```
public void decorate(Resource resource, DecoratorContext context) {  
  
    Measure comentarios = context.getMeasure(CoreMetrics.COMMENT_LINES);  
    Measure ncloc = context.getMeasure(CoreMetrics.NCLOC);  
  
    if (ResourceUtils.isRootProject(resource)) {  
        if ((MeasureUtils.hasValue(comentarios) && comentarios.getValue() > 0.0)  
            && (MeasureUtils.hasValue(ncloc) && ncloc.getValue() > 0.0)) {  
  
            double x = comentarios.getValue();  
            double y = ncloc.getValue();  
            double z = (x / (x + y)) * 100;  
  
            double n = Math rint(z);  
  
            context.saveMeasure(ProyectoMetrics.DENSIDAD_COMENTARIOS, n);  
        } else {  
  
            context.saveMeasure(ProyectoMetrics.DENSIDAD_COMENTARIOS, 0.0);  
        }  
    }  
}
```

Figura 52 Método decorate

La principal diferencia entre los métodos *analyse* y *decorate*, es que el primero de ellos recibe el proyecto propiamente dicho, mientras que el segundo recibe un recurso. Por tanto hay que realizar un filtrado previo, para definir en qué recursos queremos que se ejecute el método. Como ejemplo, y para que quede totalmente claro, si nuestro proyecto tiene 150 recursos, este método se ejecutará 150 veces. Uno por cada recurso pasado como parámetro. En nuestro caso, sólo vamos a ejecutar el código del método si el recurso es el proyecto principal. Además, comprobamos que las métricas, previamente calculadas, del número total de líneas y del número total de comentarios exista, y sea mayor que cero.

Llegados a este punto, hay que aclarar cómo usamos las métricas previamente calculadas. En la imagen 53, vemos como "importamos" esas métricas dentro de nuestro decorador.

```
@DependsUpon
public List<Metric> dependsOnCoreMetrics() {
    return Lists.<Metric>newArrayList(CoreMetrics.COMMENT_LINES,CoreMetrics.NCLOC);
}
```

Figura 53 Método para utilizar otras métricas ya calculadas

Utilizamos la anotación *@DependsUpon* antes del método donde queremos importar las métricas previamente calculadas por otros plugin o puntos de extensión. El nombre que le demos al método es indiferente.

Las nuevas métricas se guardan de la misma forma que en la clase *Sensor* pero hay que tener en cuenta que el contexto cambia para cada recurso. En este caso, hemos calculado la densidad de comentarios del código.

Ya hemos descrito la mayor parte de los componentes que forman el plugin, sólo nos queda crear un widget para mostrar los resultados en el dashboard de SonarQube.

2.3.6 Creando el widget

El widget del plugin se compone de dos clases: una clase java que extiende otra clase básica y necesita ser incluida en los puntos de extensión, y una clase Ruby que representa el widget. El código de la clase Java se muestra en la figura 54.

La clase implementa *RubyRailsWidget* que tiene el método *getTemplatePath* que hace referencia a la clase Ruby. El método *getTitle* se utiliza en la interfaz gráfica de SonarQube para mostrar el título en el dashboard, mientras que el método *getId* se utiliza para asignar un identificador único al widget.

```

@UserRole(UserRole.USER)
@Description("Widget sobre información general del proyecto y métricas de tamaño del código")
@WidgetCategory("Proyecto")
@WidgetProperties({
    @WidgetProperty(key = "Parametro1",
        description = "Este parámetro es obligatorio",
        optional = false
    ),
    @WidgetProperty(key = "ParametroMaximo",
        description = "Valor maximo",
        type = WidgetPropertyType.INTEGER,
        defaultValue = "8"
    ),
    @WidgetProperty(key = "ParametroOpcional",
        description = "Este parametro es opcional"
    )
})
public class ProyectoRubyWidget extends AbstractRubyTemplate implements RubyRailsWidget {

    public String getId() {
        return "proyecto";
    }

    public String getTitle() {
        return "Proyecto";
    }

    @Override
    protected String getTemplatePath() {
        return "/widget/proyecto_widget.html.erb";
    }
}

```

Figura 54 Clase Java que configura el widget

Por último, queda mencionar las anotaciones que hay antes de la definición de la clase. La anotación *@Description* se usa para añadir una descripción del widget en el dashboard de SonarQube, la anotación *@Category* para indicar bajo qué grupo de widget se encontrará nuestro widget. Si no añadimos esta anotación, nuestro widget sólo estará visible bajo la anotación *none* o *uncategorized list*. Finalmente, las anotaciones *@WidgetProperty* se utilizan para que el usuario final pueda definir una serie de parámetros. Todo esto se visualiza en SonarQube tal y como se muestra en la figura 55.

El método *trend_icon*, se utiliza para pintar la flecha de tendencia de la métrica. Como vemos, resulta interesante y rápido utilizar ciertos métodos que ya han sido definidos anteriormente por los desarrolladores de SonarQube.

En nuestro widget, las métricas que hemos analizado y mostramos por pantalla las podemos dividir en métricas sobre información general del proyecto, y métricas de tamaño del código. Las métricas de información general quedan representadas en el dashboard como muestra la figura 57.

DATOS GENERALES

Nombre Proyecto	Simple Java Maven Project
Fecha Análisis	Sun Apr 05 17:34:59 CEST 2015
Ruta Proyecto	C:\Users\Javier\Desktop\PROYECTO\SonarSource-sonar-examples-a3d772d\projects\languages\java\maven\java-maven-simple\target\classes

Figura 57 Métricas generales del proyecto

Por otro lado, hemos representado tres métricas directas de tamaño del código (Número de líneas totales del código, número de líneas sin comentarios del código y número de sentencias), y una métrica indirecta de tamaño del código (densidad de comentarios). Además, hemos añadido una descripción de qué es lo que mide realmente cada métrica, ya que aunque se trata de métricas muy generales, es muy frecuente que la definición de estas métricas se confunda y no sepa realmente que estamos midiendo. La figura 58 muestra la representación en el dashboard.

MÉTRICAS

Información	Métrica	Valor	Tendencia
Número de líneas totales del código fuente(espacios,variables,nombres de funciones y sentencias)	LoC	28	↗
Número de líneas sin comentar del código fuente(variables,nombres de funciones y sentencias)	NCLoC	18	↗
Número sentencias(no incluye los nombres de las funciones)	ST	9	↗
Número de comentarios del código fuente	CL	5	↗
Densidad de comentarios($CL/[NCLoC+CL]$)	DSI	22.0%	

Figura 58 Métricas de tamaño del código

La otra parte que queríamos analizar detalladamente se encuentra en la figura 59.

```

```

Figura 59 Gráfico generado mediante google chart

Dado que el proyecto se basa en el análisis de herramientas de dashboard, y debido a la importancia que le hemos dado a la presentación de la información, hemos querido presentar nuestras métricas de una forma visualmente atractiva para el usuario y además, demostramos como se pueden utilizar otras herramientas externas o extensiones en nuestro widget.

Para tal propósito hemos utilizado una herramienta de google llamada *Google Chart*, que nos permite mostrar una serie de gráficos utilizando los parámetros necesarios. En nuestro widget hemos elegido un gráfico circular y un diagrama de Venn. La figura 60 muestra el resultado final de los dos elementos.

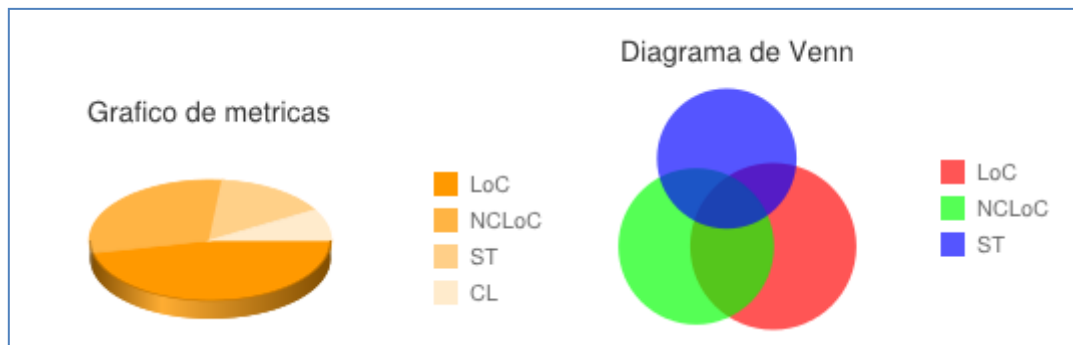


Figura 60 Gráfico circular y diagrama de Venn

El gráfico circular muestra todas las métricas de tamaño de código (directos e indirectas), mientras que el diagrama de Venn utiliza como datos sólo las métricas directas de tamaño del código. Las zonas superpuestas de las circunferencias, indican los elementos comunes en los conjuntos superpuestos, es decir, la intersección de los conjuntos, o lo que es lo mismo, el conjunto de líneas de código que son comunes en las métricas.

La representación completa del widget se puede ver en la figura 61.

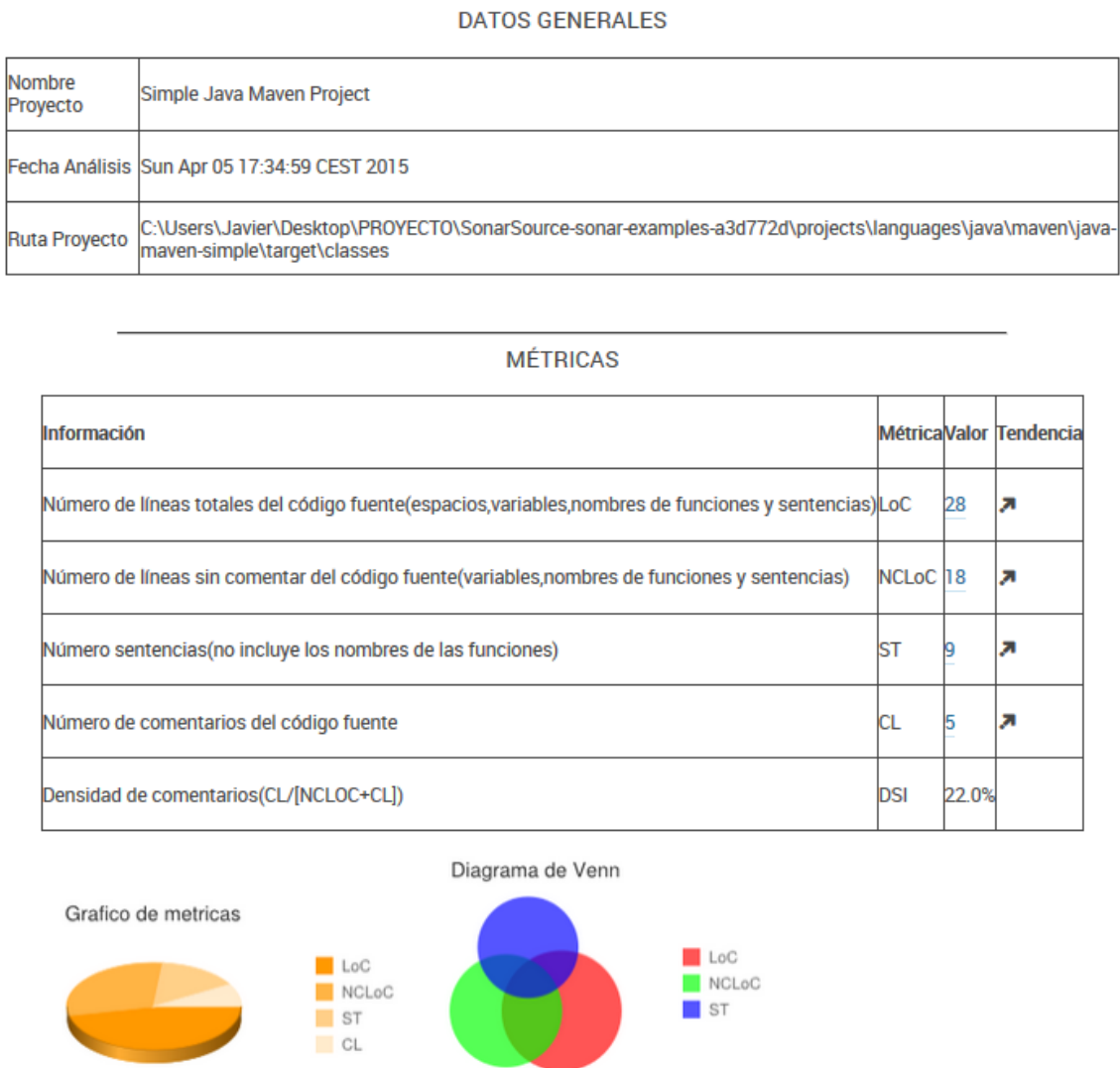


Figura 61 Representación del widget en el dashboard de SonarQube

Esto completa nuestra definición y análisis de creación de widget para SonarQube.

4 Conclusiones

Tras finalizar el estudio sobre las herramientas de calidad de software, se han formado una serie de ideas y conclusiones personales que se describen a continuación. El análisis de estas herramientas ha permitido conocer numerosas opciones a utilizar en los proyectos software, así como distintos enfoques y puntos de vista para llevar el desarrollo del software a un nivel de calidad superior.

En la primera parte del proyecto, se realizó una búsqueda de distintas herramientas que tuvieran como objetivo la calidad del software, la gestión de proyectos e incidencias o una mezcla de varias de estas características. Todo ello, con el fin de mejorar la calidad de los proyectos, de los procesos o metodologías que se llevan a cabo en los entornos reales de trabajo.

El resultado de la búsqueda fue claro: Hay numerosas opciones en el mercado, tanto de uso libre como privativo, de mayor o menor complejidad, con unas u otras características que permiten que alguna de estas herramientas sea válida para todo equipo de trabajo.

No hay razón aparente para no utilizar alguna de estas herramientas que ayuden a los equipos de trabajo, pero la experiencia personal del autor en el ámbito laboral indica lo contrario. Hoy en día, hay todavía muchos equipos de trabajo que no utilizan ninguna de estas herramientas. Es más, no utilizan ni herramientas, ni metodologías, ni procesos que ayuden a la calidad final de los resultados o a optimizar los métodos de trabajo.

Ya sea en equipos desarrollo de software o en equipos de pruebas, todavía queda mucho camino por recorrer en el mundo de la calidad de software. No se le da la importancia que se le debería, y el tiempo y los recursos que se le dedican está todavía lejos de ser aceptable.

Tras finalizar el estudio de todas las herramientas de calidad del software, se llevo a cabo un análisis en profundidad de tres herramientas software. Las conclusiones de este estudio dejan a SonarQube como la herramienta más potente de las tres analizadas.

Pero más allá de si una u otra es mejor, dado que dependiendo de las necesidades y del entorno de trabajo en el que nos encontremos puede resultar más conveniente usar una herramienta que podemos considere de un perfil inferior, hay que destacar los cuadros de mando y los dashboard de las aplicaciones estudiadas.

Estas herramientas nos ayudan a mejorar de manera significativa la toma de decisiones y por tanto, la calidad de los proyectos se incrementa notablemente. La presentación de la información de forma gráfica, ayuda a tener una visión global de los procesos que se están llevando a cabo y de cómo evolucionan con el tiempo.

Por último, el trabajo finalizó con la implementación de un plugin para SonarQube. Las posibilidades que ofrecen los plugin o extensiones son enormes, y permiten adaptar la herramienta a cualquier entorno de trabajo. Además, los plugin ofrecen la posibilidad de integrar varias herramientas en una, facilitando la gestión y la administración de los procesos de trabajo, dado que se reduce el número de herramientas que hay que mantener a diario.

5 Trabajos futuros

Una vez finalizado el proyecto, vamos a citar una serie de posibles trabajos que puede realizarse tomando como base este mismo proyecto.

Una ampliación a este trabajo, podría ser el análisis de más herramientas enfocadas a la calidad del software como Bugzilla, Mantis o Trac, y realizar una comparación más amplia entre todas las herramientas.

Otra aproximación, sería la realización de un análisis de herramientas que no presentaran la información mediante dashboard, y comparar esas herramientas con el tipo de herramientas que se han utilizado en este proyecto. Con este análisis, podríamos ver que ofrecen unas y otras herramientas, y que ventajas e inconvenientes presentan los dos tipos de herramientas.

Otro trabajo, podría ser la realización de plugins más complejos para SonarQube. Este proyecto, ofrece una base muy amplia para la realización de este tipo de plugins. Como ejemplo, proponemos la creación de un plugin que contenga todas las métricas de *Chidamber and Kememer*.

Por último, se podrían analizar plugins para otras herramientas como Jira. A partir de este enfoque, se podrían realizar numerosos trabajos como la creación de plugins, o tomar algún plugin de código abierto y extender la funcionalidad del mismo.

6 Referencias

Atlassian. Página oficial de Atlassian [En línea]. Fecha de consulta: Marzo 2015. Disponible en: <https://es.atlassian.com/>

B.A Kitchenham,. Evaluating software engineering methods and tools, part 7: planning feature analysis evaluation. SIGSOFT Software Engineering Notes 22(4), 21–24 (1997).

Bitergia. Página oficial de Bitergia [En línea]. Fecha de consulta: Marzo 2015. Disponible en: <http://bitergia.com/>

Black Duck Software, Inc. Openhub [En línea]. Fecha de consulta: Febrero 2015. Disponible en: <https://www.openhub.net/>

Fundación Wikimedia. Wikipedia [En línea]. Fecha de consulta: Marzo 2015. Disponible en: <http://es.wikipedia.org/>

G. Ann Campbell, Patroklos P. Papapetrou. SonarQube in Action. Manning Publications Co. ISBN: 9781617290954 (2013)

Google. Google charts [En línea]. Fecha de consulta: Marzo 2015. Disponible en: <https://developers.google.com/chart/>

J.L. Letouzey, C. Thierry. The « SQALE » Analysis Model. An analysis model compliant with the representation condition for assessing the Quality of Software Source Code. ISBN: 978-1-4244-7784-5 (2010)

J.L. Letouzey, M. Ilkiewicz. Managing Technical Debt with the SQALE Method. ISSN : 0740-7459 (2012)

M. Fischer, M. Pinzger, y H. Gall. Populating a Release History Database from version control and bug tracking systems. Revista Software Maintenance, páginas 23 - 32

MetricsGrimoire. Página oficial de MetricsGrimoire [En línea] Fecha de consulta: Marzo 2015. Disponible en: <https://metricsgrimoire.github.io/>

L. Patrick. JIRA 5.2 Essentials. Packt Publishing. ISBN 978-1-78217-999-3 (2013)

P. Tomas, M.J. Escalona , M. Mejias. Open source tools for measuring the Internal Quality of Java software products. A survey. Revista ELSEVIER, volumen 36. páginas 245-255 (2013)

S. Just, R. Premraj, T. Zimmermann. Towards the Next Generation of Bug Tracking Systems. Revista Visual Languages and Human-Centric Computing, páginas 82 - 85. (2008)

S. Sánchez, M. A. Sicilia, D. Rodríguez. Ingeniería del software. Un enfoque desde la guía SWEBOK. Garceta grupo editorial. ISBN: 978-84-9281-240-0 (2011)

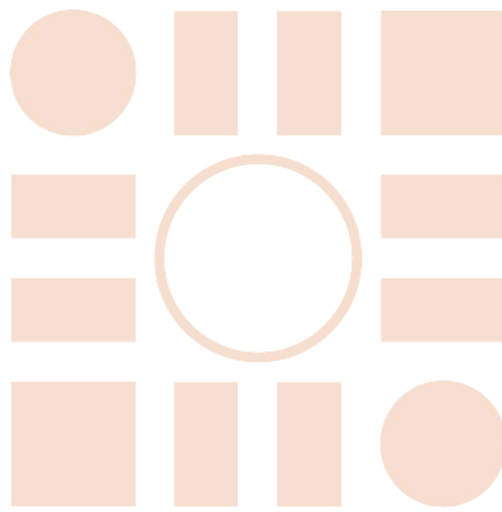
SonarSource S.A. Página oficial de SonarQube [En línea]. Fecha de consulta: Marzo 2015. Disponible en: <http://www.sonarqube.org/>

Otras referencias:

ICEBERG Workshop on software testing, metrics, data mining and decision making. University of Alcalá. 25 de marzo de 2015

Universidad de Alcalá

Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR



Universidad
de Alcalá