UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Aplicación de herramientas de análisis estático de código y de cobertura de pruebas a aplicaciones Java

Autor: Marina Pinteño Bustillos

Director: Luis Fernández Sanz

TRIBUNAL:

Presidente:

Vocal 1º:

Vocal 2º:

CALIFICACIÓN: ....

FECHA: ....

A mis padres, los responsables de que esto haya sido posible.

Nunca consideres el estudio como una obligación, sino como una oportunidad para penetrar en el bello y maravilloso mundo del saber" Albert Einstein.

AGRADECIMIENTOS

A mis padres, por darme los medios para conseguir mis metas, por apoyarme en cada decisión que he tomado y por enseñarme lo que es la vida.

A mis hermanos y a mis padres de nuevo, por creer en mí y animarme en este camino.

A Luis Fernández, por confiar en mí y guiarme en este proyecto, por las lecciones que me ha dado y que no se aprenden en los libros.

A mis compañeros y amigos, por acompañarme durante todos estos años, por las horas de estudio juntos, por sacarme una sonrisa cada día y por compartir conmigo experiencias inolvidables.

ÍNDICE GENERAL

PARTE I RESUMEN 13

1. RESUMEN BREVE 14

2. ABSTRACT 15

3. RESUMEN EXTENDIDO 16

PARTE II MEMORIA 19

CAPÍTULO 1. INTRODUCCIÓN 20

1.1 PRESENTACIÓN DEL PROYECTO Y ÁMBITO........ 20
1.2.- GLOSARIO DE TÉRMINOS/ACRÓNIMOS 22
1.3.- ESTRUCTURA DEL DOCUMENTO. 22

CAPÍTULO 2. VERIFICADORES DE ESTILO 23

2.1.- INTRODUCCIÓN 23
2.2.- CHECKSTYLE 24
2.2.1.- ¿Qué es Checkstyle? 24
2.2.2.- Integración con Eclipse 26
2.2.3.- Guía de estilos 28
2.2.3.1.- Guía de estilo de SUN 31
3.2.3.2.- Guía de estilo Java de Google. 39
2.4.- EJEMPLO PRÁCTICO DE APLICACIÓN DE CHECKSTYLE........ 48

CAPÍTULO 3. CONTROL DE COBERTURA 52

3.1.- INTRODUCCIÓN 52
3.2.- JaCoCo: JAVA CODE COVERAGE 52
3.2.1.- ¿Qué es JaCoCo?........ 52
3.2.2.- Integración con Eclipse 53
3.2.3.- Contadores de cobertura........ 54
3.3.- EJEMPLO PRÁCTICO DE APLICACIÓN DE JaCoCo 55

CAPÍTULO 4. MÉTRICAS ORIENTADA A OBJETOS 58

4.1.- INTRODUCCIÓN 58
4.2.- MÉTRICAS ORIENTADAS A OBJETO. 59
4.2.1.- MÉTRICAS MOOD (*Metrics for Object Orientes Design*) - [Abreu y Melo, 1996]........ 61
4.2.2.- MÉTRICAS CK - [Chidamber y Kemerer, 1994] 62
4.2.3.- MÉTRICAS LK - [Lorenz y Kidd, 1994]. 65
4.2.4.- MÉTRICAS DE ROBERT MARTIN - [R. Martin, 1994] 67
4.2.5.- MÉTRICAS DE LI-HENRY [Li-Henry , 1993] 68
4.3.- METRICS 68
4.4.- EJEMPLO PRÁCTICO DE APLICACIÓN DE METRICS. 71

CAPÍTULO 5. CONCLUSIONES 75
5.1. CONCLUSIONES PRINCIPALES 75
5.2. TRABAJOS FUTUROS 76

PARTE III DIAGRAMAS 77

CAPÍTULO 6. DIAGRAMAS 78

PARTE IV PLIEGO DE CONDICIONES 79

CAPÍTULO 7. PLIEGO DE CONDICIONES 80
1.- HARDWARE 80
2.- SOFTWARE 80

PARTE V MANUAL DE USUARIO 81

M.1.- CHECKSTYLE 82
M.1.1- INSTALACIÓN 82
M.1.2.- CONFIGURACIÓN BÁSICA. 87
M.1.3.- CONFIGURACIÓN PERSONALIZABLE DE UN PROYECTO. 92
M.1.4.- CONFIGURACIÓN AVANZADA 97
M.1.4.1.- USO DE FILTROS EN PROYECTOS 97
M.1.4.2.- TIPOS DE CONFIGURACIÓN 99
M.1.4.3.- CONFIGURACIÓN DE UN CONJUNTO DE ARCHIVOS 102
M.1.4.4.- CONFIGURACIÓN DE PREFERENCIAS AVANZADAS 105

M.2.- JACOCO 107
M.2.1.- INTRODUCCIÓN 107
M.2.2.- INSTALACIÓN 108
M.2.3.- GUÍA DEL USUARIO 110

M.3.- METRICS 125
M.3.1.- INSTALACIÓN 125
M.3.2.- GUÍA DEL USUARIO 128
M.3.2.1.- PREFERENCIAS 129

BIBLIOGRAFíA 135

ÍNDICE DE FIGURAS

Figura 1. Anotaciones de Checkstyle 24
Figura 2. Vista de violaciones de regla de CheckStyle del proyecto 48
Figura 3. Detalle de violación de una regla 49
Figura 4. Lugar del código donde se produce la falta 49
Figura 5. Código después de corregir la violación de la regla 50
Figura 6. View de Checkstyle después de corregir la violación de la regla 50
Figura 7. Gráfico generado por Checkstyle 51
Figura 8. Resultados de JaCoCo en su vista 56
Figura 9. Resultados de JaCoCo sobre el código 56
Figura 10. Informe de JaCoCo 57
Figura 11. Métricas de Metrics 69
Figura 12. Personalización de la métrica NORM 70
Figura 13. Personalización de la métrica LCOM 71
Figura 14. Establecimiento de rango de métricas 72
Figura 15. Resultados de Metrics 73
Figura 16. Valores de métrica desplegada 73
Figura 17. Instalación Eclipse Marketplace (I) 82
Figura 18. Instalación Eclipse Marketplace (II) 82
Figura 19. Instalación Eclipse Marketplace (Confirmación) 83
Figura 20. Instalación Eclipse Marketplace (Términos de licencia) 83
Figura 21. Instalación Eclipse Marketplace (Reinicio Eclipse) 84
Figura 22. Instalación Eclipse a través del sitio de actualización (I) 84
Figura 23. Instalación Eclipse a través del sitio de actualización (II) 85
Figura 24. Instalación Eclipse a través de un archivo de descarga (I) 86
Figura 25. Instalación Eclipse a través del sitio de actualización (II) 86
Figura 26. Selección de View de Checkstyle 87
Figura 27. View de Checkstyle 88
Figura 28. Preferencias de Checkstyle 88
Figura 29. Propiedades de Ckeckstyle 89
Figura 30. Reconstruir el proyecto 89
Figura 31. Vista de violaciones de regla de CheckStyle del proyecto 90
Figura 32. Detalle de violación de una regla 90
Figura 33. Lugar del código donde se produce la falta 91
Figura 34. Gráfico generado por Checkstyle 91

Figura 35. Configuración personalizable Checkstyle ... 92
Figura 36. Nueva configuración ... 93
Figura 37. Propiedades nueva configuración ... 93
Figura 38. Nueva configuración creada ... 94
Figura 39. Editor de la nueva configuración ... 95
Figura 40. Configuración del nuevo módulo de la nueva configuración ... 96
Figura 41. Nuevo módulo añadido ... 97
Figura 42. Configuración nueva añadida ... 97
Figura 43. Filtros ... 98
Figura 44. Configuración externa ... 100
Figura 45. Configuración remota ... 101
Figura 46. Configuración relativa del proyecto ... 102
Figura 47. Configuración de un conjunto de archivos (I) ... 102
Figura 48. Configuración de un conjunto de archivos (II) ... 103
Figura 49. Creación de un nuevo conjunto de archivos ... 104
Figura 50. Configuración de preferencias avanzadas ... 105
Figura 51. Botón de recarga ... 106
Figura 52. JaCoCo. Botón lanzamiento ... 107
Figura 53. JaCoCo. Instalación por Marketplace(I) ... 108
Figura 54. JaCoCo. Instalación por Marketplace(II) ... 108
Figura 55. JaCoCo. Instalación desde el sitio de actualizaciones (I) ... 109
Figura 56. JaCoCo. Instalación desde el sitio de actualizaciones (II) ... 109
Figura 57. JaCoCo. Instalación descargando manual (I) ... 110
Figura 58. JaCoCo. Instalación descargando manual (II) ... 110
Figura 59. JaCoCo. Verificación de instalación ... 110
Figura 60. JaCoCo. Guía del usuario (I) ... 111
Figura 61. JaCoCo. Guía del usuario (II) ... 111
Figura 62. JaCoCo. Guía del Usuario (Lanzamiento) ... 112
Figura 63. JaCoCo. Guía del Usuario (Selección de Vista de cobertura) ... 112
Figura 64. JaCoCo. Guía del Usuario (Vista de cobertura) ... 113
Figura 65. JaCoCo. Guía del Usuario (Menú desplegable) ... 113
Figura 66. JaCoCo. Guía del Usuario (Lanzamiento de la última sesión de cobertura) ... 113
Figura 67. JaCoCo. Guía del Usuario (Volcado de ejecución de los datos) ... 113
Figura 68. JaCoCo. Guía del Usuario (Eliminar la sesión activa) ... 114
Figura 69. JaCoCo. Guía del Usuario (Eliminar todas las sesiones de cobertura) ... 114
Figura 70. JaCoCo. Guía del Usuario (Combinar sesiones: varias sesiones en una sola.) ... 114
Figura 71. JaCoCo. Guía del Usuario (Activar la sesión.) ... 114

Figura 72. JaCoCo. Guía del Usuario (Colapsar todo) 114
Figura 73. JaCoCo. Guía del Usuario (Enlazar con la selección actual) 114
Figura 74. JaCoCo. Guía del Usuario (Menú desplegable) 114
Figura 75. JaCoCo. Guía del Usuario (Opciones del menú desplegable) 115
Figura 76. JaCoCo. Guía del Usuario (Anotaciones en el código fuente) 116
Figura 77. JaCoCo. Guía del Usuario (Anotaciones en el código fuente - Edición de colores) 117
Figura 78. JaCoCo. Guía del Usuario (Propiedades de cobertura) 117
Figura 79. JaCoCo. Guía del Usuario (Decoración de la cobertura I) 118
Figura 80. JaCoCo. Guía del Usuario (Decoración de la cobertura II) 118
Figura 81. JaCoCo. Guía del Usuario (Importar sesiones) 120
Figura 82. JaCoCo. Guía del Usuario (Exportar sesiones) 120
Figura 83. JaCoCo. Guía del Usuario (Personalización atajos de teclado) 122
Figura 84. JaCoCo. Guía del Usuario (Preferencias de cobertura de código) 122
Figura 85. Metrics. Instalación desde Marketplace (I) 125
Figura 86. Metrics. Instalación desde Marketplace (II) 125
Figura 87. Metrics. Instalación desde Marketplace (III) 126
Figura 88. Metrics. Instalación desde Marketplace (IV) 126
Figura 89. Metrics. Instalación desde el sitio de actualizaciones (I) 127
Figura 90. Metrics. Instalación desde sitio de actualizaciones (II) 127
Figura 91. Metrics. Guía del usuario (Mostrar Vista) 128
Figura 92. Metrics. Guía del usuario (Vista) 128
Figura 93. Metrics. Guía del usuario (Resultado de métricas en su vista) 129
Figura 94. Metrics. Preferencias 130
Figura 95. Metrics. Preferencias (Color) 131
Figura 96.Metrics. Preferencias(Personalización de la métrica LCOM) 131
Figura 97. Metrics. Preferencias (Personalización de la métrica NORM) 132
Figura 98. Metrics. Preferencias (Rango de valores) 133
Figura 99. Metrics. Fuera del rango 133
Figura 100. Metrics. Preferencias (Botón para exportar) 134

ÍNDICE DE TABLAS

Tabla 1. Reglas de Checkstyle presentes en cada convención de estilo (Sun y Google) 27
Tabla 2. Comparación de la guía de estilo de Sun con la guía de estilo de Google 30
Tabla 3. Productos y tecnologías en los que JaCoCo está integrado 53
Tabla 4. Filtros de Checkstyle 99
Tabla 5. JaCoCo. Guía del Usuario (Atajos de inicio).. 121
Tabla 6. JaCoCo. Guía del Usuario (Accesos directos en la vista de cobertura) 121

PARTE I
RESUMEN

1. RESUMEN BREVE

El aseguramiento de calidad de software cuenta con una gran variedad de técnicas y métodos para fomentar la detección precoz de defectos como el análisis estático de código para detectar y prevenir defectos en estilo de programación, estructuras poco recomendables y otros problemas.

Por otra parte, para que las pruebas de software puedan ser controladas objetivamente, es necesario contar con herramientas que controlen la cobertura de las mismas.

Este TFG tiene como objetivo configurar herramientas de soporte para Java como Checkstyle, JaCoCo y Metrics, aplicarlas sobre muestras de código y extracción de conclusiones para su aplicación en proyectos.

Palabras Clave: pruebas de software, análisis estático, Java, herramientas cobertura.

2. ABSTRACT

Software quality assurance has a variety of techniques and methods to encourage early detection of defects like static code analysis to detect and prevent defects in programming style, inadvisable structures and other problems.

Moreover, it is necessary to have tools that they control the coverage of tests for controlling them objectively.

The aims of this TFG are to setup support tools for Java as Checkstyle, Jacoco and Metrics, apply on code samples and drawing conclusions for implementation in projects.

Keywords: software testing, static analysis, Java, coverage tools.

3. RESUMEN EXTENDIDO

Este proyecto se desarrolla con el objetivo analizar el uso de diferentes herramientas gratuitas de análisis estático de código y de control de cobertura de pruebas en aplicaciones Java. Existe gran variedad de técnicas y métodos para promover la detección temprana de defectos como es el análisis estático de código que sirve para detectar y prevenir defectos en estilo de programación, estructuras poco recomendables y otros problemas.

El análisis estático de software es un análisis que se realiza al software sin tener que ejecutar el programa para obtener información sobre el mismo, de cómo está escrito,… para poder mejorarlo.

Como complemento a este análisis se realizan pruebas software para comprobar si el programa hace realmente lo que tiene que hacer. A menudo las pruebas que se diseñan no prueban todo el código dejando partes del mismo sin probar lo que conlleva un riesgo. Para ello es necesario realizar un control de cobertura del código.

Además del análisis estático de código y del control de cobertura del mismo, es necesario realizar unas medidas en el proceso de desarrollo para cuantificar lo realizado y estimar la medición de la calidad del producto. Las métricas evalúan en qué grado el software posee las distintas características que definen la calidad de un producto software.

Las métricas tradicionales no se adecúan bien a la programación orientada a objetos por lo que hay que definir métricas que se ajusten a las características específicas del software orientado a objetos. Estas métricas se centran en la herencia, complejidad de clases, polimorfismo y encapsulamiento. Además hay que tener en cuenta la cohesión y el acoplamiento, características no sólo aplicables al enfoque orientado a objetos.

Las métricas más importantes son:

- Métricas MOOD (10) que son métricas a nivel de sistema:
    - MHF - *Method Hiding Factor*.
    - MIF- *Method Inheritance Factor*
    - AHF - *Attribute Hiding Factor*
    - AIF - *Attribute Inheritance Factor*
    - PF - Polymorphism Factor
    - CF- Coupling Factor
- Métricas CK (11):
    - WMC - *Weighed Methods per Class* (Métodos ponderados por clase).
    - DIT- *Depth of Inheritance tree* (Profundidad del árbol de herencia).
    - NOC - *Number of children* (Número de hijos).

- CBO - *Coupling between object classes* (Acoplamiento entre clases).
- RFC - *Response for a class* (Respuesta de una clase).
- LCOM - *Lack of Cohesion in Methods* (Falta de cohesión de los métodos).

- Métricas LK (12):
  - Métricas de tamaño: PIM, NIM, NCM, NVV, LOC.
  - Métricas de herencia: NMO, NMI, NMA, SIX.
  - Métricas de características internas de las clases: APPM, NOM.
- Métricas de Robert Martin (14):
  - Ca: *Afferent Couplings*.
  - Ce: *Efferent Couplings*.
  - I: *Instability* = Ce/(Ca+Ce).
  - Métrica que mide la abstracción de un paquete: NAC y NTC.
- Métricas de Li-Henry (15). Se centran en la mantenibilidad del software.
  - MPC, DAC, SIZE1 Y SIZE2.

En este trabajo se propone automatizar e integrar unas herramientas libres que permita realizar los análisis y medidas mencionadas. Para ello se utilizará Eclipse como entorno de desarrollo ya que es gratuito, de código abierto y permite la instalación de plug-in fácilmente. Con la integración de las herramientas en Eclipse se facilitará la automatización del proceso.

Las herramientas que se analizan y se integran en Eclipse son:

- CheckStyle que comprueba si el código fuente sigue un estándar de estilo;
- JaCoCo para la cobertura de las pruebas y
- Metrics para la evaluación de distintas métricas.

Para comprobar su funcionamiento y sacar conclusiones de su uso se utiliza un proyecto Java procedente de GitHub llamado "El juego de la vida".

Checkstyle es una herramienta de análisis estático de código que se utiliza para comprobar que el código analizado cumple con una serie de reglas de estilo para garantizar una cierta calidad en la codificación. Es altamente configurable y se puede crear las propias reglas de estilos. En esta herramienta viene incluida la guía de estilo de Google, la guía de estilo de Sun y la guía de estilo de Sun adaptada al formateo de Eclipse.

Esta herramienta no corrige, sólo informa y maneja una serie de reglas (Anotaciones, comentarios Javadoc, bloques, diseño de clases, cabeceras, imports, métricas, misceláneo, modificadores, convenciones de nombres, violaciones de tamaño, espacios en blanco, problemas en la codificación, duplicados, otros, filtros) que se pueden configurar por nivel de rigor: *inherit, ignore, info, warning y error*.

JaCoCo es una librería libre para controlar la cobertura de código para el lenguaje de programación Java. Existen numerosas tecnologías para la cobertura de código pero son para herramientas específicas, en cambio, JaCocCo permite que se analice la cobertura de código en entornos basados en Java VM. Esta herramienta utiliza diferentes contadores para calcular métricas de cobertura: Instrucciones, ramas, complejidad ciclomática, líneas, métodos y clases.

Metrics es un plug-in de Eclipse que calcula diferentes métricas para código Java que permite que se conozca continuamente el estado del código. Para cada métrica se puede establecer un rango de valores y cuando la métrica se salga del rango, la herramienta lo notificará. Junto con el análisis, se puede obtener un informe de los valores de las mediciones de carácter visual y exportable a XML.

Las diferentes métricas que cubre Metrics se basan en (16): NOC, NSC, NOI, DIT, NORM, NOM, NOF, TLOC, MLOC, NORM, VG, WMC, LCOM. Además de estas métricas, esta herramienta considera las siguientes métricas de (14): Ca, Ce, I, A.

Posteriormente, con las herramientas instaladas se aplican sobre el proyecto Java observando y estudiando los resultados obtenidos.

Se aporta una serie de manuales de usuarios en los que se indica paso a paso cómo se instalan y cómo funcionan de forma detallada. Se indica todas las posibles opciones y configuraciones que tiene cada herramienta.

PARTE II

MEMORIA

CAPÍTULO 1. INTRODUCCIÓN

1.1 PRESENTACIÓN DEL PROYECTO Y ÁMBITO

Una de las características típicas del desarrollo de software basado en el ciclo de vida es la realización de controles periódicos, normalmente coincidiendo con los hitos del proyecto o la terminación de documentos. Estos controles pretenden una evaluación de la calidad de los productos generados (especificación de requisitos, documentos de diseño, etc.) para poder detectar posibles defectos cuanto antes. Esto es recomendable debido a diversas razones (1):

- Cuanto más avanzado esté el proyecto más costoso será corregir defectos.
- El software es tan complejo que hace que las pruebas del software no aseguren que no haya ningún defecto en él.
- Elevados costes en los controles y pruebas hacen necesario que se realicen tareas más eficientes para reducir esos costes. Se puede realizar los controles aplicando el principio de Pareto: encontrar el 20% del software donde está el 80% de los problemas. Otra forma de reducir costes es automatizando dichos controles por medio de herramientas ya que el esfuerzo del personal cualificado es más costoso.

El análisis estático de software es un tipo de análisis de software que se realiza sin ejecutar el programa (el análisis realizado sobre los programas en ejecución se conoce como análisis dinámico de software, básicamente las pruebas de software). En la mayoría de los casos, el análisis se realiza en alguna versión del código fuente y en otros casos se realiza en el código objeto. Este análisis sirve para obtener información sobre el mismo y así mejorarlo manteniendo la semántica original. El término se aplica generalmente a los análisis realizados por una herramienta automática, el análisis realizado por un humano es llamado comprensión de programas (o entendimiento de programas) como también revisión de código. El análisis estático del código proporciona una serie de sugerencias con el objetivo de mejorar la calidad del código fuente del software.

Como no se sabe si el software hace lo que se espera, es conveniente complementarlo con pruebas y profiling para poder mejorar la calidad del software. Las pruebas constituyen un método más para poder verificar y validar el software. Se puede definir la verificación como "el proceso de evaluación de un sistema o de uno de sus componentes para determinar si los productos de una fase dada satisfacen las condiciones impuestas al comienzo de dicha fase" (2). Por ejemplo, verificar el código de un módulo

significa comprobar si cumple lo marcado en la especificación de diseño donde se describe. Por otra parte, la validación es "el proceso de evaluación de un sistema o de uno de sus componentes durante o al final del proceso de desarrollo para determinar si satisface los requisitos especificados" (2).

La cobertura de código es una medida (porcentual) en las pruebas de software que mide el grado en que el código fuente de un programa ha sido probado. Sirve para determinar la calidad de las pruebas que se lleven a cabo y para determinar las partes críticas del código que no han sido comprobadas y las partes que ya fueron probadas. En primer lugar es necesario escribir pruebas y en segundo lugar contar con alguna herramienta que permita medir la cobertura. En la actualidad los lenguajes más populares cuentan con herramientas para medir la cobertura. Entre ellos, Java cuenta con un catálogo muy amplio de opciones, tanto como herramientas individuales como con opciones de entornos de desarrollo integrados.

Para obtener producto software de calidad hay que medir el proceso de desarrollo, cuantificar lo realizado, estimar la medición de la calidad de un producto software es una ventaja estratégica para las empresas debido a que proporciona conocimiento de los procesos productivos y permite conocer qué factores influyen más y permite modificarlos para que las tareas menos eficientes mejoren obteniendo así software de mayor calidad, haciendo a las organizaciones más eficientes y permitiendo una ventaja con respecto a sus competidores.

El entorno de desarrollo que se utilizará es Eclipse. Es un entorno gratuito, de código abierto y que permite de forma sencilla por medio de plug-in la integración de herramientas también gratuitas para comprobar si cumple con el convenio de estilo, cobertura de pruebas y para evaluar una serie de métricas. Con la integración de las herramientas en Eclipse se facilitará la automatización del proceso.

El lenguaje Java cuenta con muchísimas herramientas, librerías y tecnologías para la evaluación de la calidad del software. Se ha escogido herramientas libres y que se integren en Eclipse para facilitar la automatización de las mismas. Las herramientas en las que este proyecto se centrará para su análisis y estudio son:

- CheckStyle que comprueba si el código fuente sigue un estándar de estilo;
- JaCoCo para la cobertura de las pruebas y
- Metrics para la evaluación de distintas métricas.

Por lo tanto, en este proyecto se analizará el uso de herramientas de análisis estático de código, de evaluación de métricas y de cobertura de pruebas en aplicaciones Java que ayudarán a reducir el coste de realización de controles para el aseguramiento y mejora de la

calidad del producto. Además de analizar su funcionalidad se configurará sus opciones y se aplicará a muestras de código Java extrayendo conclusiones para mejorar la calidad del mismo.

1.2.- GLOSARIO DE TÉRMINOS/ACRÓNIMOS

Automatizar las herramientas: incluirlas en Eclipse para que el desarrollador vaya desarrollando cumpliendo con ciertos estándares.

OO: Orientado a Objetos.

1.3.- ESTRUCTURA DEL DOCUMENTO.

En este apartado se explica la estructura de este documento. En él se expone lo necesario para que el trabajo que se ha llevado a cabo se entienda, así como unas conclusiones que se han extraído al interpretar los resultados.

En el capítulo dos se explica qué son los verificadores de estilo y de porqué son importantes, se introduce a la herramienta Checkstyle exponiendo en qué consiste, las reglas con las que trabaja, su integración en el entorno de desarrollo Eclipse, se expone la guía de estilo de dos estándares (Sun y Google) y se les compara y, por último, en este capítulo se realiza una aplicación de la herramienta a un proyecto Java.

En el tercer capítulo de la memoria se habla sobre el control de cobertura dando una explicación de qué son, luego se introduce la herramienta JaCoCo explicando qué es, cómo funciona, contadores de cobertura con los que trabaja y, para acabar, una aplicación de esta herramienta a un proyecto Java.

El cuarto capítulo trata sobre las métricas. En primer lugar se hace una visión general de las métricas, luego se centra en las métricas orientadas a objetos como son las métricas MOOD, de Chidamber y Kemerer, de Lorenz y Kidd, de Robert Martin y de Li-Henry. También se explica la herramienta Metrics, su funcionamiento y las métricas que incluye y se aplica sobre el proyecto Java para ver los resultados de su aplicación.

Finalmente se incluyen las conclusiones, pliego de condiciones, un manual de usuario para la instalación y utilización de cada herramienta y la bibliografía consultada para la realización del trabajo.

CAPÍTULO 2. VERIFICADORES DE ESTILO

2.1.- INTRODUCCIÓN

Es importante que el código cumpla unas reglas de estilo de programación para que facilite el mantenimiento y la portabilidad del producto software y, por lo tanto, tenga una mayor calidad. Los verificadores de estilo en el código tienen un papel importante en este aspecto. Se piensa que cuando tú estudias te enseñan a tener un buen estilo al programar pero la realidad es otra, a veces es por la presión de tener que entregar cierto proyecto en un tiempo determinado, otras veces por la ignorancia de no saber programar con un buen estilo. Lo cierto es que como el estilo de programación no afecta a la funcionalidad del software no se le presta la atención que se le debería prestar. Se puede pensar que mientras que funcione el software qué más da cómo esté escrito y qué estilo use. Pero lo cierto es que no hay que dejarlo de lado.

Como bien se sabe el mantenimiento es una de las etapas del desarrollo software que más coste tiene, entre un 70% y un 80%. El mantenimiento puede ser de programas de hace años en el cual casi siempre el que hace el mantenimiento no es el propio autor del código sino que es otra persona. Un software que no ha tenido en cuenta una buena codificación será más difícil de comprender y entonces tendrá una baja calidad y, por lo tanto, su mantenimiento será menos eficiente que mantener un código bien escrito basado en una serie de estándares que se establezcan. Las convenciones de código mejoran la lectura del software haciendo que el código se entienda mucho mejor, más rápidamente y más a fondo. Todo esto repercute en un menor coste y esfuerzo a la hora del mantenimiento.

Por ello es importante seguir una forma de estilo y, evidentemente, no hay que aprenderse cada una de las reglas. La solución es aplicar herramientas de análisis estático de código y automatizarla con el entorno de desarrollo de forma que según se vaya codificando la herramienta se encarga de indicarte de forma automática si cumples con las reglas preestablecidas o no. Existe una amplia variedad de herramientas para la verificación de estilo. La herramienta escogida ha sido Checkstyle principalmente por ser una herramienta libre y por su facilidad de uso.

2.2.- CHECKSTYLE

2.2.1.- ¿Qué es Checkstyle?

Checkstyle (3) es una herramienta de análisis estático de código que se utiliza para comprobar que el código analizado cumple con una serie de reglas de estilo para garantizar una cierta calidad en la codificación. Esta herramienta ayuda a los programadores a escribir código Java cumpliendo con unos estándares establecidos, facilitando la automatización del proceso de verificación del código.

Destacar que Checkstyle sólo informa inmediatamente, a medida que se está programando, qué trozos de código no cumple con las reglas, no lo corrige.

Las ventajas que proporciona esta herramienta son:

- Facilita el mantenimiento del código.
- Asegura la calidad.
- Facilidad de uso: amigable con el usuario ya que maneja mensajes de las reglas en castellano, ventaja para algunos programadores. También se puede implementar las propias reglas obteniendo una mejor calidad en el código. Tener una mejor calidad en el código va a facilitar la codificación y el mantenimiento del mismo.

Checkstyle maneja una serie de reglas. El conjunto de reglas es muy completo y pueden configurarse con un nivel de rigor: inherit, ignore, info, warning y error. Las reglas se clasifican en los siguientes grupos:

1) <u>Anotaciones</u>: controla el estilo con el uso de anotaciones.

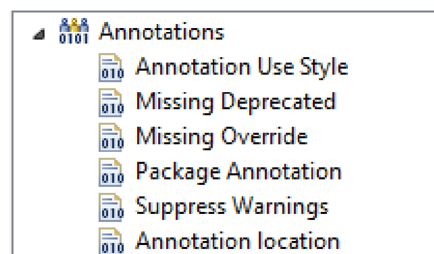


Figura 1. Anotaciones de Checkstyle

2) <u>Comentarios Javadoc</u>: Comprueban que haya comentarios Javadoc y si están bien. Por ejemplo comprueba que existan comentarios Javadoc en la definición de clases e interfaces, de métodos o constructores, etc.

3) Bloques. Reglas para los bloques de código y sus llaves, comprueba si hay bloques vacíos y bloques anidados, la colocación de la apertura y cierre de llaves, etc.

4) Diseño de clases: se comprueba el diseño de clase como por ejemplo, si hay una correcta visibilidad de los elementos de las clases.

5) Cabeceras: comprueban que las cabeceras de los ficheros se basen en unas expresiones regulares.

6) Imports: se comprueba las sentencias *import* para que no usen " * ", *imports* sin usar, que no sean redundantes, etc.

7) Métricas: restringe el número de veces que se usan operadores lógicos dentro de una misma instrucción, dependencia de clases, etc.

8) Misceláneo: comprueba el estilo de la definición de los tipos *array*, sangría, etc.

9) Modificadores: es comprueba si se sigue el orden establecido para los modificadores y evita modificadores innecesarios.

10) Convenciones de nombres: comprueba si los distintos identificadores cumplen con expresiones regulares concretas, es decir, puedes definir una expresión regular para el nombre de todo.

11) Violaciones de tamaño: define un máximo para el tamaño de tus clases, métodos, líneas y número de parámetros de un método. Sobre todo maneja la longitud de la línea de código. Por defecto, no son estrictas en el tamaño de una clase o de un archivo fuente pero sí son estrictas en el tamaño de línea.

12) Espacios en blanco: esta regla nos indica si hay espacios o saltos de líneas o tabulaciones de más en el código. Esta regla parece insignificante pero si tenemos un código de 10000 líneas de código, y hay muchos espacios en blanco, tabulaciones de más, etc. Al momento de codificar este código será más eficiente que si no tenemos espacios en blanco innecesarios. Checkstyle te previene de estos espacios en blanco innecesarios.

13) Problemas en la codificación: comprueba las malas prácticas como asignaciones internas y posibles fuentes de bugs como, por ejemplo, definir un método *equals* que no es *equals(Object*). También comprobaciones como detectar sentencias vacías, existencia de literales no definidos como constantes, sentencias *switch* que no incluyen default, etc.

14) <u>Duplicados</u>: te permite definir un mínimo de líneas para buscar código duplicado. Comprobación de código duplicado incluyendo, para la comparación del código, comentarios Javadoc.

15) <u>Otros</u>: internos a la herramienta y por defecto activados.

16) <u>Filtros</u>: para eventos de auditoría del propio Checkstyle. No hace falta mirarlos.

2.2.2.- Integración con Eclipse

Checkstyle puede integrarse como plug-in en Eclipse. Al integrarse en Eclipse se somete el código a una constante inspección notificándose inmediatamente. Esto consigue una retroalimentación directa a los programadores. Esta integración da como resultado un entorno de trabajo visual, fácil de usar e intuitivo haciendo que la configuración de las distintas funcionalidades de esta herramienta se haga de manera rápida y eficaz.

Incluye soporte para el convenio de estilo de codificación propuesto por SUN y para el propuesto por Google para Java. También es altamente configurable y estas reglas pueden ser redefinidas e incluso el usuario puede crear reglas nuevas desde cero. Esto es una gran ventaja ya que se puede adaptar al estilo de codificación interno del entorno y la configuración creada se puede utilizar en múltiples proyectos.

La configuración de reglas se puede hacer utilizando el editor de configuración Checkstyle del plug-in o incluso utilizar un archivo de configuración Checkstyle existente desde una ubicación externa.

Checkstyle cuenta con dos estándares incluidos: guía de estilo oficial de SUN y guía de estilo Java de Google. A continuación se muestra una tabla comparativa de éstos dos estándares con respecto a qué módulos de las reglas de Checkstyle usan.

REGLAS	GUÍA DE ESTILO DE SUN	GUÍA DE ESTILO DE GOOGLE
ANOTACIONES	Ninguno	Annotation location
COMENTARIOS JAVADOC	En package, method, type, variable y style.	Non empty @-clause description, Javadoc tag continuation indentation, javadoc paragraph, @-clause order, method javadoc, single line javadoc
CONVENCIONES DE NOMBRES	Constant, Local Final Variable, Local variable, Member, Method, Package, Parameter, Static Variable Name	Package, type, member, parameter, local variable, class types parameter, method type parameter, abbreviation as word in name, method

CABECERAS	Ninguno	Ninguno
IMPORTS	Evitar *, imports ilegales, imports redundantes, imports no usados	Evitar "*", custom import oprder
VIOLACIONES DE TAMAÑOS	Máxima longitud de archivo, de línea y de método. Máximo número de parámetros	Máxima longitud de línea
ESPACIOS EN BLANCO	Empty For Iterator Pad, Generic Whitespace, Method Parameter Pad, No whitespace after and before, Operator wrap, Paren Pad, Tyoecast paren pad, whitespace after and before	File tab character, empty line separator, separator wrap, generic whitespace, method parameter pad, operator wrap
MODIFICADORES	Modifier Order and refundant modifier	Modifier order
BLOQUES	Avoid nested blocks, empty block, left curly brace placement, need braces, rigth curly brace placement	Empty block, need braces, left and rigth curly brace placement
PROBLEMAS EN LA CODIFICACIÓN	Avoid inline conditionals, empty statement, equals ans hashCode, hidden field, illegal instantiation, inner assignment, magic number, missing switch default, simplify boolean return and expression	Illegal tokens text, one statement per line, multiple variable declaration, missing switch default, fallthrough, no finalizer, overload methods declaration order, variable declaration usage distance
DISEÑO DE CLASES	Design for extension, final class, hide utility class constructor, interface is type, visibility modifier	One top level class
MÉTRICAS	Ninguna	Ninguna
MISCELÁNEAS	Translation, array type style, final parameter, upper ell	Indentatin, upper ell, array type style, avoid escaped unicode characters, outer type file name
OTROS	Checker, treeWalker	Checker, treeWalker, noLineWrap, summaryJavadocCheck
FILTERS	Ninguno	Ninguno

Tabla 1. Reglas de Checkstyle presentes en cada convención de estilo (Sun y Google)

La instalación del plug-in y de la cómo configurar un proyecto está en el apartado M.1.1 Instalación de Checkstyle.

2.2.3.- Guía de estilos

Como se ha indicado anteriormente, Checkstyle cuenta con la guía de estilo oficial de SUN y guía de estilo Java de Google como estándares. A continuación se muestra una tabla comparativa de éstos dos estándares.

<table>
<tr><th></th><th></th><th>GUÍA DE ESTILO DE SUN</th><th>GUÍA DE ESTILO DE GOOGLE</th></tr>
<tr><td rowspan="2">FICHEROS
Serán de extensiones .java o .class</td><td>NOMBRE</td><td>GNUmakefile para ficheros "make".
README.</td><td>Codificados en UFT-8
Son casi-sensitive</td></tr>
<tr><td>ORGANIZACIÓN</td><td colspan="2">Evitar ficheros de más de 2000 líneas de código.
Consta de las siguientes partes:
- Comentarios iniciales: nombre de la clase, versión, fecha y copyright.
- <code>Package</code> e <code>import</code>.
- Declaraciones de clases:
  - Comentarios de documentación
  - Sentencia <code>class</code> e <code>interface</code>.
  - Modificadores por orden: <code>public, protected, private, abstract, static, final, transient, volatile, synchronized, native, strictfp...</code>
  - Constructores.
  - Métodos.</td></tr>
<tr><td rowspan="4">FORMATO</td><td>SANGRÍA</td><td>4 espacios como unidad de sangría.
Un tab = 8 espacios
Usar tabuladores o espacios, no ambos.</td><td>2 espacios cada vez que se abre un bloque. Será de 4 espacios o más:
- Hay ajuste de línea. La línea de después de la primera tiene esa sangría.
- Múltiples líneas puede variar a más de 4 espacios.</td></tr>
<tr><td>LLAVES</td><td colspan="2">La apertura de llaves en la misma línea de la declaración, el cierre de llaves en nueva línea alineada con la declaración.
Si no hay nada entre las llaves se pueden juntas. ("{}")
Usarlas aunque sólo haya una sentencia.</td></tr>
<tr><td>FORMATO DE LÍNEA</td><td>Máximo 80 caracteres.
Romper líneas detrás de comas o delante operador binario.
Alinear la expresión cortada con el principio de su expresión.</td><td>Máximo entre 80 - 100 carateres.
No tienen límite de caracteres en:
- URL de javadoc
- Sentencias de package e imports
- Líneas de comandos
Romper líneas que se pasen del límite:
- antes de un operador binario.
- después del símbolo de asignación
- nombre de constructor o método unido a la apertura de paréntesis.
- después de una coma.</td></tr>
<tr><td>DECLARACIONES</td><td colspan="2">Una declaración por línea.
Inicializar cuanto antes.</td></tr>
</table>

<table>
<tr>
<td></td>
<td>ESPACIOS EN BLANCO</td>
<td>LÍNEA EN BLANCO:
- Entre las secciones de un fichero
- Entre las definiciones de clases e interfaces.
- Antes de un comentario
- Entre dos métodos
- Después de la cabecera de un método.
- Después de un bloque de declaraciones de variables locales.
- Entre secciones lógicas de un método.
ESPACIOS EN BLANCO:
- Entre palabra reservada y un paréntesis.
- Después de cada coma.
- Alrededor de un operador binario.
- Las partes de una sentencia for.
- Después de un Cast.</td>
<td>LÍNEAS EN BLANCO:
- Entre los miembros consecutivos de una clase. Opcionalmente entre campos para agruparlos lógicamente.
- Opcionalmente antes del primer miembro o después del último miembro de una clase.
ESPACIOS EN BLANCO:
- Entre palabra reservada y un paréntesis.
- Entre el cierre de llaves y una palabra reservada.
- Antes de una apertura de llaves.
- En ambos lados de un operador binario o ternario.
- Después del cierre de un paréntesis de un cast.
- En ambos lados de una doble barra que comienza un comentario al final de una línea.
- Entre los tipos y una variable de una declaración.
- Opcionalmente dentro de las llaves al iniciar un array.</td>
</tr>
<tr>
<td rowspan="2">COMENTARIOS</td>
<td>COMENTARIOS DE IMPLEMENTACIÓN</td>
<td colspan="2">Evitar comentarios obvios, sólo poner los útiles.
No encerrarlos en cuadrados de asteriscos u otros caracteres.
Elegir buen nombre a variables evita comentarios.
Misma sangría que el bloque a que hace referencia.
<code>/*</code>
<code>*Comentario de bloque</code>
<code>*/</code>
<code>//comentario sencillo</code>
<code>/*comentario sencillo*/</code></td>
</tr>
<tr>
<td>COMENTARIOS DE JAVADOC.</td>
<td>/**
* Comentario de clase
*/</td>
<td>/**
* Comentario de clase
*/
O si es una línea:
/** una línea javadoc */

Línea en blanco entre párrafos.
Orden de las At-clauses: @param, @return, @throws, @deprecated. Y nunca descripción vacía.
Fragmento resumen al principio de cada clase.
Se usa como mínimo en las public class y en cada public o protected miembro de una clase.</td>
</tr>
</table>

<table>
<tr><td rowspan="5">CONVENCIONES DE NOMBRES</td><td>GENERAL</td><td>Usar nombres que aporten información.
Palabras unidas se pone la primera letra de cada palabra en mayúsculas</td><td>Usar sólo letras ASCII y números o guion bajo.
Packages en minúsculas.</td></tr>
<tr><td>CLASES E INTERFACES</td><td colspan="2">Sustantivos con primera letra en mayúsculas: UpperCamelCase
ClaseEjemplo</td></tr>
<tr><td>MÉTODOS</td><td colspan="2">Verbos con las letras en minúsculas excepto si es compuesta por más palabras: LowerCamelCase
metodoEjempo</td></tr>
<tr><td>VARIABLES Y PARÁMETROS</td><td colspan="2">Nombres cortos con significado: LowerCamelCase
Evitar nombres de un sólo carácter excepto en el bucle for.
miVariable</td></tr>
<tr><td>CONSTANTES</td><td colspan="2">Siempre en mayúsculas y si son más de una palabra separadas con _
MI_CONSTANTE = 4;</td></tr>
<tr><td colspan="2">HÁBITOS DE PROGRAMACIÓN</td><td>Poner bloques entre llaves aunque sólo tenga una sentencia.
No hacer ninguna variable de instancia o clase pública sin una buena razón.
Usar paréntesis para evitar confusiones.
Evitar uso de literales.
En un SWITCH siempre tiene que aparecer la sentencia default aunque esté vacía.
Valores de retorno. Usar
<code>return (cond?sent1: sent2);</code>
En vez de:

<code>If (exprBool) {</code>
<code>   return true;</code>
<code>} else {</code>
<code>   return false;</code>
<code>}</code></td><td>Poner bloques entre llaves aunque sólo tenga una sentencia.
Usar paréntesis para evitar confusiones.

En un SWITCH siempre tiene que aparecer la sentencia default aunque esté vacía.

@Override para anular el método de una superclase. Si el método al que quiere anular es @Deprecated, se puede omitir override.

Excepción Caught: sino hace nada poner un comentario de la razón.</td></tr>
<tr><td colspan="2">ANOTACIONES</td><td>-------</td><td>Cada anotación en una línea nueva y sin sangría pero en una nueva línea el bloque al que hace referencia.
Si el bloque no contiene parámetros pueden aparecer en una misma línea.
Las anotaciones que hacen referencia a un mismo campo pueden ir seguidos en la misma línea.</td></tr>
</table>

Tabla 2. Comparación de la guía de estilo de Sun con la guía de estilo de Google

2.2.3.1.- Guía de estilo de SUN

Esta guía (4) pertenece a Oracle y está archivado. No se sigue manteniendo y su última actualización fue en 1999. A pesar de esto es una convención importante y a tener en cuenta. Las recomendaciones que proporciona este documento se recogen de forma resumida a continuación.

1) **Nombre de los ficheros**. En esta sección se indica cuáles son las extensiones y los nombres de los ficheros.

   a) Las extensiones de los ficheros tienen que ser: `.java` para ficheros fuente en Java y `.class` para ficheros de bytecode Java.7

   b) En cuanto a los nombres, los más usados son: GNUmakefile: para ficheros “make”. Se usa gnumake para construir nuestro software y README: fichero que resume los contenidos de un directorio en particular.

2) **Organización de los ficheros.** Los ficheros de más de 2000 líneas son incómodos y deben ser evitados. Los ficheros fuente Java constan de las siguientes partes:

   a) Comentarios iniciales en el que se indique el nombre de la clase, información de la versión, fecha y copyright.

   b) Sentencia `package` primero y luego los `imports` como primeras líneas no-comentarios de los ficheros.

   c) Declaraciones de clases e interfaces. Éstas se dividen a su vez en:

      i) Comentarios de documentación de a clase o interface `(/**…*/)`.

      ii) Sentencia `class` o `interface`.

      iii) Comentario de implementación de la clase o interface si fuera necesario. Este comentario debe contener cualquier información aplicable a toda clase o interface que no era apropiada para estar en los comentarios de la documentación de la clase o interface.

      iv) Variables de clase. El orden es el siguiente: `public - protected - private - abstract - static - final - transient - volatile synchronized - native - strictfp`

      v) Constructores.

      vi) Métodos: agrupándolos por funcionalidad más que por visión.

3) Sangría.

a) Emplear 4 espacios como unidad de sangría.

b) Los tabuladores deben ser exactamente cada 8 espacios. Usar tabuladores o espacios pero no ambos.

c) Longitud de línea:

i) Líneas de máximo 80 caracteres.

ii) Romper las líneas detrás de una coma o delante de un operador binario, si hay operaciones anidadas, cuanto mayor nivel mejor.

```
algunMetodo(longExpresion1, longExpresion2, longExpresion3, longExpresion4,
        longExpresion5);

var = algunMetodo1(longitudExpresion1,
                 algunMetodo2(longExpresion2,
                          long"expresion3));

longNombre1 = longNombre2 * (longNombre3 + longNombre4 - longNombre5)
             + 4 * longNombre6; //PREFER

longNombre1 = longNombre2 * (longNombre3 + longNombre4
                           - longNombre5) + 4 * longNombre6; //EVITAR

//NO USAR ESTA SANGRÍA
if ((condicion1 && condicion2)
    || (condicion3 && condicion4)
    || (condicion5 && condicion6)) { //BAD WRAPS
    hacerAlgoSobreEso();
}

//USAR ESTA SANGRÍA
if ((condicion1 && condicion2)
        || (condicion3 && condicion4)
        || (condicion5 && condicion6)) {
    hacerAlgoSobreEso();
}

//O USAR ESTA SANGRÍA
if ((condicion1 && condicion2) || (condicion3 && condicion4)
        || (condicion5 && condicion6)) {
    hacerAlgoSobreEso();
}
```

iii) La expresión cortada que se alinee al principio de su expresión.

4) **Comentarios.** Hay dos tipos de comentarios: comentarios de implementación y comentarios de documentación (Comentarios Javadoc).

a) Comentarios de implementación son para comentar nuestro código o para describir la especificación del código para ser leídas por desarrolladores que pueden no tener el código fuente a mano.

i) Comentarios sólo relevantes para la lectura y entendimiento del programa. Evitar comentarios obvios.

ii) No encerrarlos en grandes cuadrados dibujados con asteriscos u otros caracteres.

iii) Una forma de evitar comentarios es eligiendo bien los nombres de atributos, variables y métodos.

```
/*
* Comentario de bloque.
*/
/* Comentario sencillo */

Instrucción;    //Comentario
```

b) Comentario de documentación (Javadoc). Estos comentarios describen clases Java, interfaces, constructores, métodos y atributos. Debe aparecer justo antes de la declaración y no dentro.

```
/**
* Comentario Javadoc, la clase Ejemplo ofrece ...
*/
public class Ejemplo { ...
```

5) **Declaraciones.**

a) Cantidad por línea: una declaración por línea ya que facilita los comentarios.

```
int nivel; // nivel de sangria
int tamano; // tamano de la tabla
```

Eso es preferible a:

```
int nivel, tamaño;
```

b) Inicialización: intentar inicializar las variables locales cuando se declaran.

c) Colocación: ponerlas justo al inicio del bloque donde estén. Un bloque es la parte del código encerrada entre llaves ("{ " y "}"). Excepto el índice de los bucles for que se pueden declarar e inicializar en la sentencia `for`.

d) No declarar la misma variable en un bloque interno.

e) Declaraciones de clases e interfaces:

i) Ningún espacio en blanco entre el nombre del método y el paréntesis que abre su lista de parámetros.

ii) Los métodos se separan con una línea en blanco.

f) Uso de llaves: poner la llave abierta en la misma línea que la declaración y la llave cerrada en una nueva línea, alineada con la declaración. Si no hay nada entre las llaves se puede poner el cierre de llaves justo después de la apertura ("{}").

6) Sentencias.

a) **Sentencias simples**: cada línea una sola sentencia.

b) **Sentencias return**: no debe usarse con paréntesis a menos que hagan el valor de retorno más obvio.

c) **Sentencias if, if-else, if else-if else**:

```
if (condicion) {
    sentencias;
}
if (condicion) {
    sentencias;
} else {
    sentencias;
}
if (condicion) {
    sentencias;
} else if {
    sentencias;
} else {
    sentencias;
}
```

d) **Sentencias for**:

```
for (int i = 0; i < tamaño; ++i) {
    sentencias;
```

```
}
```

La variable de control se declara dentro del bucle:

```
for (int i = 0; i < tamaño; ++i) {
    sentencias;
}
```

e) **Sentencias While**:

```
while (condición) {
    sentencias;
}
```

f) **Sentencias Do-While**:

```
do {
    sentencias;
} while (condición);
```

g) **Sentencias Switch**:

Siempre tiene que haber un caso default aunque no tenga nada.

```
switch (condicion) {
case ABC:
sentencias;
/* este caso se propaga */ Cuando uno de los casos no tenga break
case DEF:
      sentencias;
      break;
case XYZ:
      sentencias;
      break;
default:
      sentencias;
      break;
}
```

h) **Bloques try-catch**: Una sentencia `try-catch` debe tener la siguiente forma:

```
try {
    sentencias;
} catch(ExceptionClass e) {
     sentencias;
}
```

Una sentencia `try-catch` puede ir seguida de un `finally`, cuya ejecución se ejecutará independientemente de que el bloque `try` se haya completado con éxito o no.

```
try {
    sentencias;
} catch(ExceptionClass e) {
     sentencias;
} finally {
     sentencias
}
```

7) Espacios en blanco.

a) Línea en blanco:

i) Entre las secciones de un fichero fuente.

ii) Entre las definiciones de clases e interfaces.

iii) Entre dos métodos.

iv) Entre las variables locales de un método y su primera sentencia.

v) Antes de un comentario de bloque o de un comentario de una línea.

vi) Entre las secciones lógicas de un método para facilitar la lectura.

b) Espacios en blanco.

i) Entre una palabra reservada y un paréntesis.

ii) Después de cada coma.

iii) Alrededor de un operador binario.

iv) Las partes de una sentencia for deben separarse con espacios en blanco.

v) Después de un “`Cast`”.

8) Convenciones de nombres.

a) Usar nombres con una palabra o frase corta que aporten información. No abreviaturas.

b) Clases e Interfaces: siempre sustantivos con la primera letra en mayúsculas. Si es palabra compuesta la letra mayúscula de cada palabra. Ejemplo: `ClaseEjemplo`.

c) Métodos: siempre verbos con las letras en minúsculas. Si es palabra compuesta la letra mayúscula de cada palabra. Ejemplo: `metodoEjemplo`.

d) Variables: siempre nombres cortos pero con significado. Ejemplo: `miVariable`.

e) Constantes: siempre en mayúsculas y separadas con _ si son más de dos palabras. `MI_CONSTANTE=4;`

9) Hábitos de programación.

a) No hacer ninguna variable de instancia o clase pública sin una buena razón.

b) Evitar usar un objeto ara acceder a una variable o método de clase (`static`).

c) Evitar el uso de literales (excepto -1,0 y 1).

d) Inicializar una variable en la declaración.

e) Usar paréntesis para evitar confusiones.

f) Valores de retorno.

i) Usar `return (condición? Sentencia1: Sentencia2);` En vez de

```
if (expresionBooleana) {
    return true;
} else {
    return false;
}
```

i) Usar `return expresiónBooleana;` En vez de:

```
if (condicion) {
    return x;
}
return y;
```

10) Ejemplo de código.

```
/*
* @(#)Bla.java           1.82 99/03/18
*
* Copyright (c) 1994-1999 Sun Microsystems, Inc. All rights reserved.
```

```
*
*/

package java.bla;

import java.bla.blabla.BlaBla;

/**
*      La descripción de la clase viene aqui.
*
* @version          datos de la versión (numero y fecha)
* @author           Nombre Apellido
*/
public class Bla extends OtraClase {
             /* Un comentario de implemantación de la clase viene aquí.*/

      /**
           El comentario de documentación de claseVar1v    */
      public static int claseVar1;

      /**
      * El comentario de documentación de classVar2
      * ocupa más de una línea
      */
      private static Object claseVar2;

      /** Comentario de documentación de instanciaVar1 */
      public Object instanciaVar1;

      /** Comentario de documentación de instanciaVar2 */
      protected int instanciaVar2;

      /** Comentario de documentación de instanciaVar3 */
      private Object[] instanciaVar3;

      /**
      * ...Comentario de documentación del constructor Bla...
      */
      public Bla() {
           // ...aquí viene la implementación...
      }
      /**
      * ...Comentario de documentación del método hacerAlgo...
      */
      public void hacerAlgo() {
          // ...aquí viene la implementación...
      }

      /**
```

```
        * ...Comentario de documentación de hacerOtraCosa...
        * @param unParametro descripción
        */
        public void hacerOtraCosa(Object unParametro) {
        // ...aquí viene la implementación...
        }
}
```

3.2.3.2.- Guía de estilo Java de Google.

Esta guía (5) está basada en la anterior y está actualizada para incluir las novedades del lenguaje. Además incluye una sección de buenas prácticas.

1) Ficheros.

a) Nombre: son case-sensitive y su extensión es `.java`.

b) Archivos codificados en UTF-8.

2) Organización de los ficheros.

Los ficheros fuente Java constan de cuatro partes. Cada parte está separada por una línea en blanco. Dentro de cada grupo no tiene que aparecer línea en blanco.

a) Comentarios iniciales que aparezca la información de la licencia y copyright si tiene.

b) Sentencia `package`. No se aplica el límite de 80-100 caracteres por sentencia.

c) Sentencias `import`. No se aplica el límite de 80-100 caracteres por sentencia. Estas sentencias se separan en diferentes grupos (separados por una línea en blanco):

   i) Static imports.

   ii) Com.`google` imports.

   iii) Imports de terceras partes en orden de ordenación ASCII como `android`, `com`, `org`, `junit`, `sun`

   iv) `Java` imports.

   v) `Javax` imports.

d) Declaraciones de clases. Sus métodos deben llevar un orden lógico. No dividir las sobrecargas, por ejemplo si hay varios constructores o métodos con el mismo nombre que aparezcan de forma secuencial.

3) Formato.

a) <u>Llaves</u>: son usados con las sentencias `if, else, for, do y while` incluso cuando el cuerpo está vacío o contiene una sentencia simple. Ver ejemplo.

   i) Antes de la apertura de las llaves no salto de línea.

   ii) Después de la apertura de llaves o antes del cierre de llaves sí salto de línea.

   iii) Hay salto de línea después del cierre de llaves si es de un método, constructor o clase llamada. Si ese cierre va seguido de else o de una coma no hay salto de línea.

   iv) Para las clases enumeradas hay una serie de excepciones que se ven en el 3)h).

   v) Bloques vacíos pueden cerrarse las llaves inmediatamente ({}).

b) <u>Sangría</u>: dos espacios cada vez que se abra un nuevo bloque, cuando se cierre, en el siguiente nuevo bloque se eliminan los dos espacios. Como por ejemplo:

```
return new MyClass() {
  @Override public void method() {
      if (condition()) {
        try {
          something();
        } catch (ProblemException e) {
         recover();
        }
      }
   }
};
```

c) <u>Una sentencia por línea</u>.

d) <u>Líneas de 80 o de 100 caracteres</u>. Excepciones:

   i) Líneas donde el límite es imposible como URL en Javadoc.

   ii) Sentencia de package e imports.

   iii) Líneas de comandos que pueden ser copiados y pegados en una Shell.

e) <u>Ajuste de línea</u>. Consiste en dividir una línea que se pasa del máximo de caracteres establecidos. Con las excepciones citadas anteriormente.

i) Principalmente se debe romper la línea en un nivel superior. Además:

(1) Cuando la línea se rompe en un operador de no asignación, se rompe antes del símbolo. También se aplica a los operadores (. & |)

(2) Cuando la línea se rompe en un operador de asignación, se rompe después del símbolo.

(3) Un nombre de método o constructor permanece unido a la apertura de paréntesis que le sigue.

(4) Una coma permanece unido al carácter que le precede.

ii) La sangría será de 4 espacios o más cuando:

(1) Hay un ajuste de línea, cada línea después de la primera tiene una sangría de 4 espacios o más desde la línea original.

(2) Cuando hay múltiples líneas, la sangría puede variar más de 4 espacios. En general, dos líneas usan la misma sangría si comienzan con elementos sintácticamente paralelos.

f) <u>Espacios en blanco</u>.

i) *Líneas en blanco*:

(1) Entre los miembros consecutivos de una clase (constructores, campos, métodos….). Es opcional poner líneas en blanco entre campos pero se puede poner para agruparlos de forma lógica.

(2) Dentro del cuerpo de los métodos como necesidad de crear agrupamiento lógico.

(3) Opcionalmente antes del primer miembro o después del último miembro de una clase.

ii) *Espacios en blanco*:

(1) La separación de una palabra reservada (`if, for o catch`) de una apertura de paréntesis (( ) que le sigue.

(2) La separación de una palabra reservada (`else o catch`) del cierre de una llave (}) que le precede.

(3) Antes de una apertura de llaves con dos excepciones:

(a) `SomeAnnotation({a,b})`.

(b) `String [][] x = {{“foo”}}`.

(4) En ambos lados de un operador binario o ternario. También en los dos puntos (:), en amperrsand y en | de un catch.

(5) Después de ,:; o del cierre de un paréntesis de un `cast`.

(6) En ambos lados de una doble barra que comienza un comentario al final de una línea.

(7) Entre los tipos y una variable de una declaración: `List<String> list`

(8) Es opcional dentro de las llaves al iniciar un array. Son válidos:

```
new int[] {5,6}
new int[] {5,6}
```

iii) *Alineación horizontal*: no es requerido.

```
private int x; //this is fine
private Color color; //this too

private int x;     //permitted, but future edits
private Color color;    //may leave it unaligned
```

g) <u>Agrupación de paréntesis</u>. Son opcionales pero recomendables. Se puede prescindir de ellos cuando el código no da pie a confusión.

h) <u>Constructores específicos</u>.

i) *Clases Enum*: después de cada coma que sigue a la constante, el salto de línea es opcional.

```
private enum Suit {CLUBS, HEARTS, SPADES, DIAMONDS}
```

ii) *Declaraciones e inicialización de variables*: se declara una variable por línea. Cuando sea necesario e inicialización tan pronto como sea posible. las variables locales se declaran cerca del punto que se utilizan en primer lugar (dentro de lo razonable), para minimizar su alcance.

iii) *Arrays*. Pueden declararse como un bloque, como por ejemplo:

```
new int[] {
  0, 1, 2, 3
}

new int[] {
  0,
  1,
  2,
  3
}
```

```
new int[] {
  0, 1,
  2, 3
}

new int[]
{0,     1,     2,     3}
```

Los corchetes forman parte del tipo, no de la variable:

```
String [] args es válido, String args[] no es válido
```

iv) *Sentencias Switch*. Su sangría es como cualquier otro bloque. Fall-through comentado. La sentencia default tiene que aparecer aunque no contenga código.

```
switch (input) {
  case 1:
  case :
    prepareOneOrTwo();
    // fall through
  case3:
    handleOneOrTwo ();
  default:
    handleLargeNumber(input);
}
```

v) *Anotaciones*. Cada anotación en una línea y en una línea nueva y sin sangría el bloque al que hace referencia.

```
@Nullable
public String getNameIfPresent() { ... }
```

La excepción es que si no tiene parámetros pueden aparecer en la misma línea:

```
@Override public int hashCode() { ... }
```

Y las anotaciones que aparecen para un mismo campo pueden ir en la misma línea:

```
@Partial @Mock DataLoader loader;
```

vi) *Comentarios*. Misma sangría que el bloque al que hace alusión el comentario. Pueden ser /*…*/ o //. El primer estilo es para una línea o para múltiples líneas donde a partir de la primera línea se empieza con * alineados con la línea anterior. Los comentarios // son para una línea sólo.

```
/*
 * Esto
 * está bien.
 */

// Y esto
// también

/* También puedes
 * incluso hacer esto.  */
```

Nota: no usar comentarios encerrados en cajas de asteriscos u otros caracteres.

vii) Modificadores. Cuando están presentes, aparecen en el orden recomendado por la especificación del lenguaje Java:

(1) Modificadores. El orden es el siguiente:

```
public protected private abstract static final transient volatile
synchronized native strictfp
```

(2) Literales: Sufijo de long integer en mayúsculas, nunca en minúsculas. Por ejemplo, `3000000000L`, en vez de `3000000000l`

4) Nombrado.

a) <u>Normas comunes a todos los identificadores.</u>

i) Usar sólo letras ASCII y números o guion bajo.

b) <u>Normas para identificadores</u>.

i) *Nombre de los paquetes en minúsculas*. Ejemplo: `com.example.deep_space`

ii) *Nombre las clases e Inferfaces*. Escritos en UpperCamelCase.

(1) Normalmente son nombres o frases nominales. Las interfaces también pueden ser adjetivos.

(2) Las clases de test empiezan o acaban con la palabra `Test`.

iii) *Nombre de los métodos*:

(1) Escritos en LowerCamelCase.

(2) Normalmente son verbos o frases verbales (Por ejemplo, sendMessage).

(3) En los métodos de test JUnit pueden separarse los nombres con guion bajo.

iv) *Nombre de constantes*:

(1) Normalmente son nombres.

```
// Constantes
static final int Number = 5;
static final InmutableList<String> NAMES = InmutableList.of("Ed, "Ann");
static final Joiner COMMA_JOINER = Joines.on(','); //Porque Joiner es
inmutable
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }
```

(2) Todas las letras en mayúsculas y las palabras separadas por guion bajo.

v) *Nombre de parámetros*:

(1) Normalmente son nombres. Los nombres con un solo carácter deben ser evitados.

(2) Escritos en lowerCamelCase.

vi) *Nombre de las variables locales*. Los nombres de un solo carácter deben ser evitados. Exceptuando las variables temporales y bucles.

vii) Nombre del tipo de las variables:

(1) Pueden nombrarse de uno de los dos estilos que hay:

(2) Letra mayúscula seguida por un número simple (Por ejemplo, `T2`).

(3) Un nombre de la clase seguido por una letra mayúscula. (por ejemplo, `RequestT`).

(4) Los nombres de un solo carácter deben ser evitados. Exceptuando las variables temporales y bucles.

c) Camel case. Existen frases en inglés o acrónimos como "IPv6" o "iOS" que hay que convertir a camel case. Los pasos a seguir son:

i) Convertir la frase en ASCII y eliminar los apóstrofos. Por ejemplo, Müller's Algorithm puede ser Muellers algorithm.

ii) Dividir el resultado en palabras. Si la palabra está en camel case como por ejemplo "AdWords" se pone como "ad words".

iii) Ahora se ponen todas las letras en minúsculas.

(1) En mayúsculas la primera letra de cada palabra.

(2) Si es lower camel case en mayúsculas la primera letra de cada palabra excepto la primera palabra.

iv) Unir las palabras.

v) Ejemplos de palabras con camelCase correctas e incorrectas:

(1) XML HTTP request -> XmlHttpRequest (Correcto) -> XMLHTTPRequest (Incorrecto)

(2) new customer ID -> newCustomerId (correcto) -> newCustomerID (incorrecto)

5) Prácticas de Programación.

a) `@Override`. Se usa cuando un método de clase anula un método de superclase o un método de clase que implementa un método de interfaz. Si el método de la superclase es `@Deprecated`, se puede omitir `@Override`.

b) Excepción Caught: raramente es correcto no hacer nada en respuesta a la captura de una excepción. En el caso que no haya ninguna acción, la razón se pone con un comentario. En test puede omitirse el comentario.

c) Miembros estáticos. Cuando una referencia a un miembro de la clase estática debe ser calificado, se clasificó con el nombre de esa clase, no con una referencia o expresión de tipo de esa clase.

```
Foo aFoo = ...;
Foo.aStaticMethod(); // bien
aFoo.aStaticMethod(); //mal
somethingThatYieldsAFoo().aStaticMethod(); //muy mal
```

d) Finalizadores: no se usan.

6) Javadoc.

a) <u>Formato</u>

i) *Formas generales*.

Un bloque de Javadoc básico se muestra en la siguiente imagen:

```
/**
 * Aquí se escriben múltiples líneas de Javadoc
 *
 */
public int method(String p1) { ... }
```

O si es una línea:

```
/** Aquí se escriben una corta línea de Javadoc. */
```

ii) *Párrafos*. Una línea en blanco (línea con un asterisco) aparece entre párrafos y antes defrupo de at-clauses.

iii) *At-Clauses*: aparecen en este orden: `@param`, `@return`, `@throws`, `@deprecated` y nunca aparecen con una descripción vacía. Cuando una at-clause no cabe en una línea, la línea a continuación tiene una sangría de 4 o más espacios desde la posición de `@`.

b) <u>El fragmento resumen</u>. El Javadoc para cada clase y miembro comienza con un fragmento de resumen breve. Este fragmento es muy importante: es la única parte del texto que aparece en ciertos contextos, como los índices de clase y método.

c) Donde se usa `Javadoc`. Como mínimo está presente en las `public class` y en cada `public` o `protected` miembro de una clase. Cada vez que hay un comentario explicando el comportamiento de una clase o método es mejor usar Javadoc. Hay algunas excepciones.

i) Excepción: Métodos autoexplicativo. Javadoc es opcional para los métodos simples y obvios como “`getFoo`.”

ii) Excepción: `Overrides`. Javadoc no siempre está en un método que sobre escribe un método `super`.

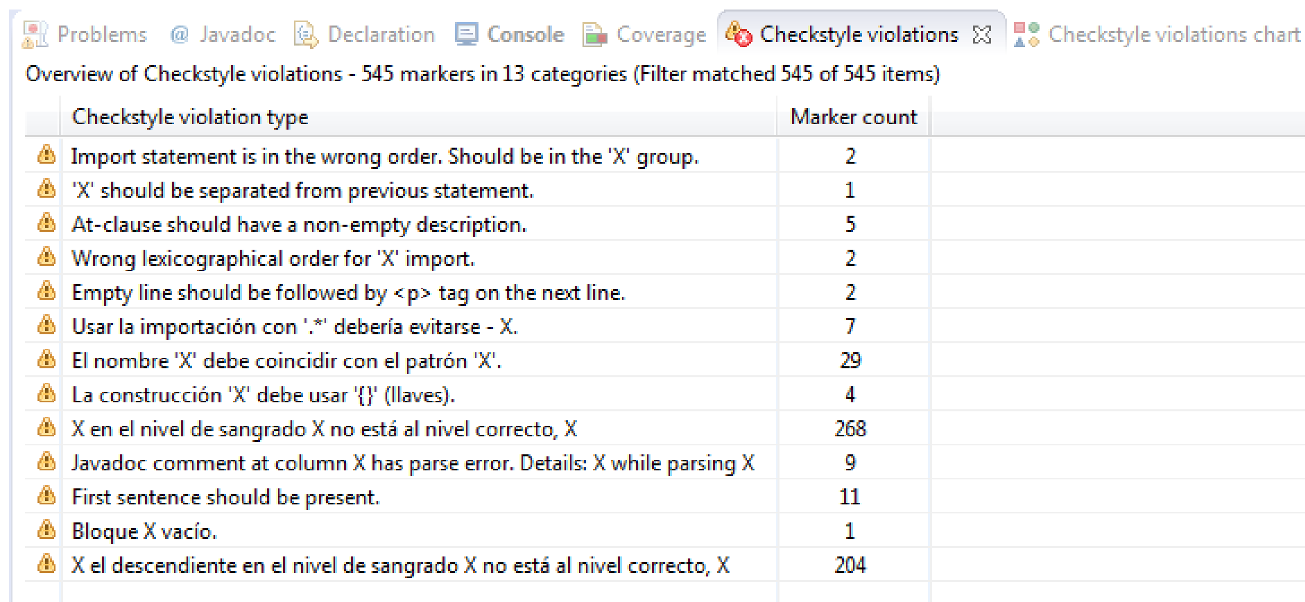
2.4.- EJEMPLO PRÁCTICO DE APLICACIÓN DE CHECKSTYLE

En este apartado se aplica la herramienta Checkstyle y se comprueba su funcionamiento en un proyecto Java. El proyecto Java se ha descargado de GitHub (https://github.com/mac-comp124-f13/game-of-life) y se llama el juego de la vida.

Para tener el programa controlado por Checkstyle se siguen los pasos que se indican en el Manual de usuario de esta herramienta.

Para este ejemplo práctico se ha usado el estándar de Google.

Una vez reconstruido el proyecto podemos ver las diferentes violaciones de las reglas que se han obtenido en la pestaña de "Checkstyle violations" si las hay. En el caso de este proyecto nos encontramos con las siguientes violaciones:



Problems @ Javadoc Declaration Console Coverage Checkstyle violations Checkstyle violations chart

Overview of Checkstyle violations - 545 markers in 13 categories (Filter matched 545 of 545 items)

Checkstyle violation type	Marker count
Import statement is in the wrong order. Should be in the 'X' group.	2
'X' should be separated from previous statement.	1
At-clause should have a non-empty description.	5
Wrong lexicographical order for 'X' import.	2
Empty line should be followed by <p> tag on the next line.	2
Usar la importación con '.*' debería evitarse - X.	7
El nombre 'X' debe coincidir con el patrón 'X'.	29
La construcción 'X' debe usar '{}' (llaves).	4
X en el nivel de sangrado X no está al nivel correcto, X	268
Javadoc comment at column X has parse error. Details: X while parsing X	9
First sentence should be present.	11
Bloque X vacío.	1
X el descendiente en el nivel de sangrado X no está al nivel correcto, X	204

Figura 2. Vista de violaciones de regla de CheckStyle del proyecto

Como se puede observar la mayoría de las violaciones son por la sangría, no está en el nivel correcto. Si pinchamos en el cada uno de los tipos de despliega el tipo y muestra un listado detallado de los detalles de las diferentes localizaciones en las que se ha detectado esta violación.

Problems | @ Javadoc | Declaration | Console | Coverage | Checkstyle violations | Checkstyle violations chart

Details of Checkstyle violation "X en el nivel de sangrado X no está al nivel correcto, X" - 268 occurrences

Resource	In Folder	Line	Message
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	95	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	15	member def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	25	method def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	29	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	35	method def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	37	member def type en el nivel de sangrado 8 no está al...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	39	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	45	method def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	48	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	54	method def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	60	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	66	method def modifier en el nivel de sangrado 4 no es...

Figura 3. Detalle de violación de una regla

Y pulsando sobre cada una de ellas se accede a la construcción que produce la violación.

GameBoard.java | Conway.java | MainWindow.java

```
3  /**
4   * RuleSet implementing Conway's Game of Life.
5   *
6   * @author Michael Ekstrand <ekstrand@cs.umn.edu>
7   */
8  public class Conway implements RuleSet {
9
10     public String getName() {
11         return "Conway's Rules";
12     }
13
14     /**
15      * Applies the rules of Conway's Game of Life.
```

Problems | @ Javadoc | Declaration | Console | Coverage | Checkstyle violations | Checkstyle violations chart

Details of Checkstyle violation "X en el nivel de sangrado X no está al nivel correcto, X" - 268 occurrences

Resource	In Folder	Line	Message
Conway.java	/game-of-life-master/src/edu/macalester/comp124...	10	method def modifier en el nivel de sangrado 4 no es...

Figura 4. Lugar del código donde se produce la falta

Ya sabemos dónde exactamente se viola la regla y donde. Haciendo cumplir la regla, se reconstruye el proyecto y se ve que no hay advertencia:

```
2
3⊖ /**
4   * RuleSet implementing Conway's Game of Life.
5   *
6   * @author Michael Ekstrand
7   */
8  public class Conway implements RuleSet {
9
10⊖    public String getName() {
11        return "Conway's Rules";
12    }
13
14⊖      /**
15       * Applies the rules of Conway's Game of Life.
```

Figura 5. Código después de corregir la violación de la regla

Y se refleja que ya no aparece esa violación.

Problems | @ Javadoc | Declaration | Console | Coverage | Checkstyle violations | Checkstyle violati

Overview of Checkstyle violations - 541 markers in 13 categories (Filter matched 541 of 541 items)

Checkstyle violation type	Marker count
Import statement is in the wrong order. Should be in the 'X' group.	2
'X' should be separated from previous statement.	1
At-clause should have a non-empty description.	5
Wrong lexicographical order for 'X' import.	2
Empty line should be followed by <p> tag on the next line.	2
Usar la importación con '.*' debería evitarse - X.	7
El nombre 'X' debe coincidir con el patrón 'X'.	29
La construcción 'X' debe usar '{}' (llaves).	4
X en el nivel de sangrado X no está al nivel correcto, X	266
Javadoc comment at column X has parse error. Details: X while parsing X	8
First sentence should be present.	11
Bloque X vacío.	1
X el descendiente en el nivel de sangrado X no está al nivel correcto, X	203

Figura 6. View de Checkstyle después de corregir la violación de la regla

También se puede ver estas violaciones en forma de gráfico que se puede guardar en formato PNG:

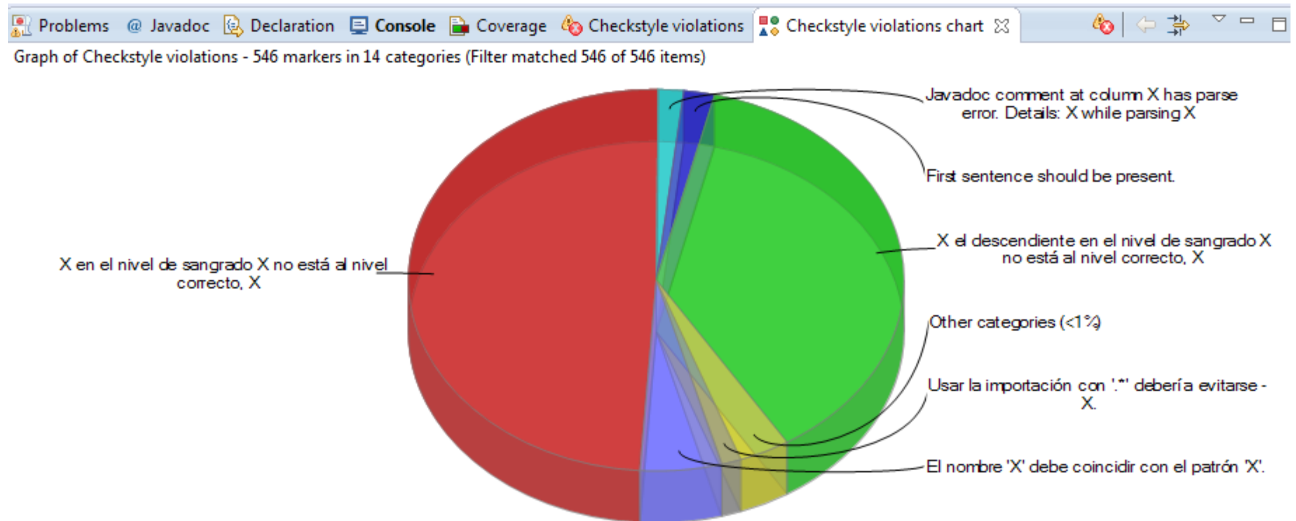


Figura 7. Gráfico generado por Checkstyle

¿Qué se consigue con el uso de Checkstyle?

Se consigue que se lleve a cabo unas reglas y estándares que se proponga en la empresa. Los desarrolladores tienen que seguir esas reglas las cuales no tiene que aprenderse ya que el IDE le indicará si no cumple las reglas.

Normalmente las personas que desarrollan el software son distintas a las que luego tendrán que mantenerlo, y que cada uno codifica de una manera distinta, por ello si se sigue unos estándares, el mantenimiento del código se hace más eficiente. Un código escrito por varias personas puede ser un caos para su entendimiento sino se ha llevado un estilo de codificación posterior provocando un mayor gasto, una menor eficiencia a la hora de mantenerlo y resultaría un software de mala calidad.

El estilo de programación adoptada por un proyecto de desarrollo de software puede ayudar a cumplir con las buenas prácticas de programación que mejoran la calidad del código, la legibilidad, la reutilización y reducir el costo del desarrollo.

CAPÍTULO 3. CONTROL DE COBERTURA

3.1.- INTRODUCCIÓN

Para asegurar la calidad del software hay que tener una buena estrategia de pruebas. Además es conveniente realizar un análisis de la cobertura de pruebas. La cobertura consiste en la cantidad de código recorrido por una serie de casos de pruebas, es decir, comprobar qué partes del código han sido sometidas a pruebas y cuáles no.

Se puede decidir si completar el 100% de la cobertura de sentencias o el 80% (asumiendo riegos) o completar el 100% en componentes propensos a defectos y reducir el porcentaje al 70%. El proceso consiste en ejecutar casos, comprobar la cobertura y añadir nuevos casos según se requiera para completar el criterio de pruebas.

Para medir la cobertura hacemos uso de la herramienta JaCoCo que permite controlar si las pruebas han pasado por todos los caminos de decisión. Esta información se obtiene gráficamente como valores estadísticos. Existen muchas formas métricas para medir la cantidad de código que se ha probado. JaCoCo realiza un análisis de cobertura de instrucciones, ramas, clases, métodos, líneas y complejidad ciclomática y también genera informes sobre los análisis realizados.

3.2.- JaCoCo: JAVA CODE COVERAGE

3.2.1.- ¿Qué es JaCoCo?

JaCoCo es una librería libre para controlar la cobertura de código para el lenguaje de programación Java, creado por el equipo de EclEmma y se ha basado en todo lo aprendido de usar e integrar las librerías existentes durante muchos años.

Es cierto que existen varias tecnologías de cobertura de código abierto para Java pero ninguna de ellas está realmente diseñado para la integración ya que la mayoría son especialmente aptas para una herramienta en particular. En cambio, JaCoCo proporciona una tecnología estándar para analizar la cobertura de código en entornos basados en Java VM proporcionando una librería ligera, flexible y bien documentada para la integración con diversas herramientas de construcción y desarrollo.

Actualmente JaCoCo está integrado en los siguientes productos y tecnologías[1]:

Integración de JaCoCo/EclEMma project		
Tecnología	Documentación	Comentarios
Java API	http://www.eclemma.org/jacoco/trunk/doc/api/index.html	
Command Line	http://www.eclemma.org/jacoco/trunk/doc/agent.html	
Apache Ant	http://www.eclemma.org/jacoco/trunk/doc/agent.html	
Apache Maven	http://www.eclemma.org/jacoco/trunk/doc/maven.html	
Eclipse	Proyecto EclEMma(http://www.eclemma.org/)	Desde versión 2.0
Integración de terceras partes		
Productos	Documentación	Comentarios
Arquillian	http://arquillian.org/modules/jacoco-extension/	Framework de prueba de Java EE
Gradle	http://gradle.org/docs/current/userguide/jacoco_plugin.html	Build System with JaCoCo plug-in
IntelliJ IDEA	https://www.jetbrains.com/idea/help/code-coverage.html	Desde versión 11.1
Jenkins	https://wiki.jenkins-ci.org/display/JENKINS/JaCoCo+Plugin	Proyecto GSoC de Ognjen Bubalo
NetBeans	http://wiki.netbeans.org/MavenCodeCoverage	Desde versión 7.2
sbt	https://bitbucket.org/jmhofer/jacoco4sbt	Simple Build Tool
TeamCity	http://docs.codehaus.org/display/SONAR/Code+Coverage+by+Unit+Tests+for+Java+Project	Servidor de integración continua desde la versión 8.1
Urban Code	https://developer.ibm.com/urbancode/plugin/jacoco-3519516/	Plataforma de entrega continua por IBM

Tabla 3. Productos y tecnologías en los que JaCoCo está integrado

3.2.2.- Integración con Eclipse

Al igual que Checkstyle, JaCoCo puede integrarse como plug-in en Eclipse. Al integrarse en Eclipse se facilita la automatización del proceso para controlar la calidad del producto y se consigue una retroalimentación directa a los programadores. Esta integración

[1] http://www.eclemma.org/jacoco/trunk/doc/integrations.html

da como resultado un entorno de trabajo visual, fácil de usar e intuitivo haciendo que la configuración de las distintas funcionalidades de esta herramienta se haga de manera rápida y eficaz.

Su instalación y funcionamiento se pueden ver en el apartado M.2.Manual de Usuario de JaCoCo.

3.2.3.- Contadores de cobertura

JaCoCo utiliza diferentes contadores para calcular métricas de cobertura por medio de información contenida en los archivos de clase Java que básicamente son las instrucciones de código de bytes de Java e información de depuración opcionalmente incrustados en archivos de clase. Este enfoque permite el análisis de aplicaciones incluso cuando el código fuente no está disponible. De todas formas hay limitaciones a este enfoque. Los archivos de clase tienen que ser compilado con información de depuración para calcular la cobertura de nivel de línea. No todas las construcciones del lenguaje Java pueden ser compiladas directamente a código byte correspondiente. En tales casos, el compilador de Java crea los llamados código sintéticos que a veces da lugar a resultados inesperados de cobertura de código.

C1: Instrucciones.

La unidad más pequeña son las simples instrucciones de código de byte. La cobertura de instrucciones da información sobre la cantidad de código que ha sido ejecutada o no.

Este criterio se cumple cuando se recorren todas las instrucciones al menos una vez.

C2: Ramas (Branches).

JaCoCo calcula la cobertura en las ramificaciones de todos los estamentos de if y switch. Esta métrica cuenta el total de ramificaciones que hay en un método y determina cuantas se han ejecutado y cuantas no. Por lo tanto, este criterio se cumple cuando cada decisiOn se evalúa a true y false al menos una vez. El manejo de excepciones no se considera en esta métrica. Una vez realizado la métrica, en la parte izquierda de las líneas de código aparecen diamantes de un color u otro dependiendo de cómo han sido ejecutadas.

- No hay cobertura: no ha sido ejecutada (diamante color rojo).
- Cobertura parcial: sólo una parte ha sido ejecutada (diamante color amarillo).
- Cobertura completa: todas han sido ejecutadas (diamante color verde).

C3: Complejidad Ciclomática.

JaCoCo también calcula complejidad ciclomática para cada método no abstracto y resume la complejidad de las clases, los paquetes y grupos. Según McCabe, la complejidad del código puede obtenerse desde su flujo de control, o dicho de una manera más exhaustiva del número de rutas linealmente independientes del código. Por lo tanto, el valor de la complejidad puede servir para indicar el número de casos de pruebas unitarias necesarias para cubrir completamente una pieza de software. JaCoCo calcula la complejidad ciclómatica de un método con la ecuación siguiente:

$v(G) = B - D + 1$ (B) número de ramas y (D): el número de puntos de decisión.

C4: Líneas.

Una línea de la fuente se considera ejecutado cuando al menos una instrucción que se asigna a esta línea ha sido ejecutada. Como una línea puede tener múltiples instrucciones de código, se muestra tres estados diferentes para cada línea:

- No hay cobertura: ninguna instrucción de la línea ha sido ejecutada (color rojo).
- Cobertura parcial: sólo una parte de la línea ha sido ejecutada (color amarillo).
- Cobertura total: todas las instrucciones han sido ejecutadas (color verde).

C5: Métodos

Un método se considera como ejecutado cuando al menos una instrucción ha sido ejecutado. Como JaCoCo funciona en el nivel de código de bytes también constructores e inicializadores estáticos se cuentan como métodos.

C6: Clases.

Una clase se considera como ejecutada cuando al menos uno de sus métodos ha sido ejecutado. Como los tipos de interfaz Java pueden contener inicializadores estáticos tales interfaces también se consideran como clases ejecutables.

3.3.- EJEMPLO PRÁCTICO DE APLICACIÓN DE JaCoCo

En este apartado se aplica la herramienta JaCoCo en el mismo proyecto Java en ql que se aplicó Checkstyle para ver cómo funciona y los resultados que da.

1.- El primer paso es hacer clic con el botón derecho sobre el proyecto y seleccionar *Coverage as > JUnit Test.*

Al comprobar la cobertura de pruebas del proyecto se puede observar que con las pruebas que tiene hay clases que no han sido probadas como InvalidBoardException y otras que apenas las prueban como LifeComponent.java.

Problems | @ Javadoc | Declaration | Console | Coverage | Checkstyle violations | Checkstyle violations chart

edu.macalester.comp124.life (1) (18-jun-2015 11:31:23)

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
game-of-life-master	58,1 %	860	621	1.481
src	49,5 %	600	612	1.212
edu.macalester.comp124.life	49,5 %	600	612	1.212
MainWindow.java	52,0 %	274	253	527
LifeComponent.java	33,1 %	92	186	278
GameBoard.java	57,8 %	212	155	367
InvalidBoardException.java	0,0 %	0	16	16
Conway.java	91,7 %	22	2	24
test	96,7 %	260	9	269
edu.macalester.comp124.life	96,7 %	260	9	269
BoardTest.java	95,0 %	171	9	180
BoardTest	95,0 %	171	9	180
ConwayTest.java	100,0 %	89	0	89
ConwayTest	100,0 %	89	0	89

Figura 8. Resultados de JaCoCo en su vista

Gracias al control de cobertura de pruebas podemos saber las partes de nuestro código que han sido probadas y cuáles no. Con esto se observa que hay una estrategia de pruebas baja ya que en total es un 58'1% de cobertura por lo que sería aconsejable incluir algunos casos de prueba más para llegar al menos un 80% de código probado y asegurar así una mayor calidad del software.

Dentro del código se puede observar por medio de colores en las líneas de código cuáles han sido probadas totalmente (color verde), cuáles no han sido probadas nada (color rojo) y cuáles han sido probadas en parte (color amarillo).

```
263         */
264     public void stateChanged(ChangeEvent e) {
265         Object source = e.getSource();
266         if (source == tbRun) {
267             onRunToggled();
268         } else {
269             ButtonModel b = ruleSetButtons.getSelection();
270             if (b.getActionCommand().equals("conway")) {
271                 board.setRuleSet(new Conway());
272             } else {
273                 // Uncomment for HighLife
274                 // board.setRuleSet(new HighLife());
275             }
276         }
277     }
```

Figura 9. Resultados de JaCoCo sobre el código

Además, se puede obtener un informe del análisis de cobertura.

edu.macalester.comp124.life (1) (18-jun-2015 11:31:23)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
game-of-life-master		58%		26%	52	99	159	341	23	61	1	10
Total	621 of 1.481	58%	56 of 76	26%	52	99	159	341	23	61	1	10

edu.macalester.comp124.life (1) (18-jun-2015 11:31:23) Created with JaCoCo 0.7.2.20140912

Figura 10. Informe de JaCoCo

Esta herramienta facilita la realización de pruebas al poner en conocimiento a los desarrolladores o testers qué parte del código se ha probado o no, si el código que no se ha probado no hace falta probarlo o se ha pasado y es necesario diseñar pruebas para ello, si una parte del código propensa a errores está completamente cubierta,… en definitiva, esta herramienta hace que el software que se está desarrollando tenga una mayor calidad.

CAPÍTULO 4.
MÉTRICAS ORIENTADA A OBJETOS

4.1.- INTRODUCCIÓN

Medición es "el proceso por el cual se asignan números o símbolos a atributos de entidades del mundo real de tal manera que describa dichos atributos de acuerdo con unas reglas claramente definidas" (6). Una *entidad* puede ser una persona, un programa desarrollado o el proceso de desarrollo. Un *atributo* (dimensión) es una característica o propiedad de una entidad como puede ser la altura de una persona o la longitud del proceso de desarrollo. La finalidad del uso de las métricas es evaluar sistemas para conseguir alta calidad, robustez y facilidad de mantenimiento.

Las métricas evalúan en qué grado el software posee las distintas características que definen la calidad de un producto software, por lo que se puede decir que las métricas tienen un papel decisivo para obtener un producto de calidad.

Las métricas tradicionales no se adecúan bien a la programación orientada a objetos por lo que hay que definir métricas que se ajusten a las características que diferencian el software orientado a objetos del software con otro enfoque. Estas métricas se centran en la herencia, complejidad de clases, polimorfismo y encapsulamiento. Los objetivos de las mismas son (7):

- Comprender la calidad del producto.
- Estimar la efectividad del proceso.
- Mejorar la calidad del trabajo realizado a nivel del proyecto.

El software orientado a objetos se distingue del software tradicional en las siguientes características en las cuales se centran las métricas:

- **Encapsulamiento**: engloba las responsabilidades de una clase, incluyendo sus atributos y operaciones. Esta característica influye en las métricas dado que cambia el objetivo de las mediciones pasando de un módulo simple a ser un paquete de atributos y operaciones.
- **Ocultación de la información**: un sistema OO bien diseñado tiene que impulsar al ocultamiento de información. Esta característica consiste en no mostrar los detalles de la implementación de las rutinas a los elementos del programa exteriores al

objeto. Aquellas métricas que proporcionen una indicación del grado en que se ha conseguido ese ocultamiento de la información indicarán la calidad del programa.

- **Herencia**: es un mecanismo que posibilita que las responsabilidades de un objeto se pasen a otros objetos, es decir. se basa en una relación jerárquica entre varias clases de tal forma que se pueden compartir atributos y operaciones en la subclase de la superclase de la que hereda ciertas propiedades y añade propiedades particulares. Favorece la reutilización Las métricas indicarán si el programa tiene un grado adecuado de jerarquía y niveles de herencia.

- **Técnicas de abstracción de objetos**: la abstracción es un mecanismo que permite al diseñador a centrarse en los detalles importantes de un componente de un programa sin prestar atención a los detalles de niveles inferiores. Las métricas OO representan la abstracción en términos de medidas de una clase como por ejemplo número de instancias por clase por aplicación. Estas métricas permiten evaluar la calidad de las abstracciones indicando si hace falta una mejora.

Otras características no sólo aplicables al enfoque orientado a objetos son:

- **Cohesión**. Es la medida en que un componente se dedica a realizar una tarea para la que fue creado, delegando las tareas complementarias a otros componentes, es decir, un objeto tiene un alto nivel de cohesión si ejecuta una tarea sencilla y requiere poca interacción con métodos que se ejecutan en otros objetos. Un objeto coherente debe hacer (idealmente) una sola cosa. La cohesión debe ser máxima.

- **Acoplamiento**. Indica en qué medida una clase utiliza atributos y/o métodos de otra. El acoplamiento entre clases debe ser mínimo.

Teniendo en cuenta estas características, no sólo deben construirse usando encapsulamiento, abstracción de datos y ocultamiento de la información, sino que deben poseer alta cohesión y bajo acoplamiento. Una proporción adecuada de estas características en u producto software hace que tenga una buena calidad.

La medición ayuda a evaluar la presencia de estas características. En el siguiente apartado se presenta una serie de métricas orientada a objetos que son las que servirán para evaluar y medir la calidad de programas y aplicaciones en lenguaje Java.

4.2.- MÉTRICAS ORIENTADAS A OBJETO.

(8) agrupa cada una de las métricas acorde con el tipo de granularidad que mide. Los tipos de granularidad son: sistema, acoplamiento, herencia, clase y método. Basándose en lo

que (8) han definido, en (9) se describen un conjunto representativo de métricas orientadas a objetos en los que se pueden establecer rangos de valores fuera de los cuales una clase o trozo de código es más propensa a errores. Así que se cuenta con:

- **Métricas a Nivel de Sistema**: "Aunque un sistema puede subdividirse en componentes, se considera que actúa como un sistema. Además, las características de un buen componente son las de un buen sistema y al revés. Las características medibles de un sistema pueden incluir el número de clases en el sistema"(8). A nivel de sistema opera el conjunto de métricas MOOD (*Metrics for Object Oriented Design*) definido por (10) e incluyen: MHF (Factor de Ocultamiento de los métodos), MIF (Factor de herencia de los métodos) ,AHF (Factor de Ocultamiento de los Atributos) , AIF (Factor de Herencia de los Atributos), PF (Medidas de proporción de polimorfismo) y CF (Medidas de proporción de Acoplamiento).

- **Métricas de Acoplamiento**: Con esta métrica se mide cuánto una clase usa métodos y atributos de otra clase. Para formar un sistema las clases tienen que interactuar y esa interacción puede indicar su complejidad. Las métricas de acoplamiento se incluye la métrica de Acoplamiento entre clases de objetos (CBO) definida por (11). Miden el acoplamiento de una clase con las otras.

- **Métricas de Herencia**: Las clases se encuentran en entramado de jerarquía de clases. Si hay un cambio en la clase padre, las demás subclases heredan los cambios. Estas relaciones pueden indicar al diseñador donde los cambios podrían mejorar el desarrollo. El entramado en sí contiene características interesantes como la profundidad y la amplitud. En este tipo de métrico se encuentran las definidas por (11): DIT (Profundidad del Árbol de Herencia) y NOC (Número de hijos); y la definida por (12): SIX (Índice de Especialización por Clase).

- **Métricas de Clases**: Las clases pueden contener métodos que son innecesarios o demasiados complejos para trabajar juntos. Puede tener datos extraños que compliquen el proceso de programación. Estas métricas identifican características dentro de las clases descubriendo clases que pueden necesitar rediseñarse. Dentro de este tipo de métrica se encuentras las definidas por (11): RFC (Respuesta de una clase), WMC (Métodos ponderados por Clase) y LCOM (Falta de Cohesión en los Métodos).

- **Métricas de Métodos**: Atributos y métodos se producen en el mayor nivel de detalle. Las características de los métodos y atributos son conocidos y las técnicas para analizarlos son las tradicionales. Métodos tienen la complejidad adicional de llamadas a otros que los contiene. En estas métricas se encuentra, por ejemplo, la

tradicional complejidad ciclomática de (13). También se encuentran: LOC (Líneas de código) y NOM (Número de mensajes enviados).

Las métricas que se han nombrado surgen de la bibliografía especializada: Métricas CK, Métricas MOOD, Métricas LK. A continuación se hace una descripción de estos grupos de métricas.

4.2.1.- MÉTRICAS MOOD (*Metrics for Object Orientes Design*) - [Abreu y Melo, 1996]

Abreu y Melo presentaron estas métricas para medir los principales mecanismos de los diseños orientados a objetos como son encapsulamiento, herencia, polimorfismo y acoplamiento para evaluar la productividad del desarrollo y la calidad del producto.

Como se ha visto en el apartado anterior, estas métricas están encuadradas dentro de métricas a nivel de sistema.

MHF - *Method Hiding Factor*.

Consiste en el cociente entre la suma de las invisibilidades de todos los métodos definidos en todas las clases y el número total de métodos en el sistema. Invisibilidad de un método consiste en el porcentaje total de clases desde las cuales el método es invisible.

Es decir, es la proporción entre el número de métodos privados y el número total de métodos.

(10) han demostrado empíricamente que cuanto mayor es MHF, la densidad de defectos y el esfuerzo necesario para corregirlos debería disminuir.

Propiedad OO que trata: Encapsulamiento.

MIF- *Method Inheritance Factor*

Medidas de proporción de métodos heredados sobre la base del total de métodos del sistema. Es un medio para expresar el nivel de reusabilidad de un sistema.

Propiedad OO que trata: Herencia.

AHF - *Attribute Hiding Factor*

Medidas de proporción de atributos ocultos sobre la base del total de atributos del sistema. Lo ideal es que sea 100% y que se debe evitar los atributos públicos para no violar los principios de encapsulación.

Propiedad OO que trata: Encapsulamiento.

AIF - *Attribute Inheritance Factor*

Medidas de proporción de atributos heredados sobre la base del total de atributos del sistema. Es un medio para expresar el nivel de reusabilidad de un sistema. Si hay mucha reutilización a través de herencia, el sistema se hace más difícil de entender y mantener.7

Propiedad OO que trata: Herencia.

PF - Polymorphism Factor

Medidas de proporción de polimorfismo. Es la relación entre el número de métodos heredados redefinidos dividido entre el máximo número de situaciones polimórficas distintas.

Propiedad OO que trata: Polimorfismo.

CF- Coupling Factor

Medidas de proporción de Acoplamiento. Indica la relación entre clases. Cuanto mayor es el acoplamiento, mayor es la complejidad del sistema.

Propiedad OO que trata: Acoplamiento.

4.2.2.- MÉTRICAS CK - [Chidamber y Kemerer, 1994]

Estas métricas son las propuestas por los autores Chidamber y Kemerer (11), es uno de los conjuntos de métricas más referenciados. Ellos fueron los primeros en establecer un conjunto de seis métricas específicas para código OO. Todas menos una se aplican a clases tratando de medir la complejidad, el acoplamiento, la cohesión, herencia y comunicación inter-clase. A continuación se detalla cada una de las métricas indicando la propiedad OO con la que se relacionan.

WMC - Weighed Methods per Class (Métodos ponderados por clase).

Calcula la suma de la complejidad ciclomática de los métodos de una clase:

$$WMC=\Sigma_{i..n}\, mc_i$$

siendo mc_i la complejidad ciclomática del método i.

- Las clases con muchos métodos es probable que sean de aplicación específica lo que limita la posibilidad de reutilización.

- WMC también es un indicador de la cantidad de tiempo y esfuerzo que se requiere para desarrollar y mantener la clase.

- Un gran número de métodos también significa un mayor impacto sobre las clases derivadas ya que éstas heredan de los métodos de la clase base.

Por lo tanto, se recomienda mantener un WMC bajo ya que un alto WMC puede dar lugar a un árbol de herencia más complejo, menos reutilizable y, por lo tanto, más propenso a más errores.

Las clases con alto valor de WMC podrían ser reestructuradas en varias clases más pequeñas. Una buena WMC es definiendo los límites como por ejemplo entre 20 o 50 métodos o especificando que un máximo de 10% de las clases pueden tener más de 24 métodos. Esto permite clases grandes pero que la mayoría sean pequeñas.

Propiedad que trata: Complejidad.

DIT- Depth of Inheritance tree (Profundidad del árbol de herencia).

El valor del DIT nos indica:

- Mayor profundidad (valor alto) indica mayor complejidad, hay más probabilidad de heredar más métodos y variables, más difícil de entender su comportamiento.

- La herencia es una herramienta para gestionar la complejidad, en realidad, no la incrementa. Como factor positivo, mayor profundidad promueven la reutilización de métodos de herencia.

Propiedad que trata: Herencia.

NOC - Number of children (Número de hijos).

Es el número de subclases inmediatamente subordinadas a una clase en la jerarquía. NOC mide la amplitud de un árbol mientras que DIT mide la profundidad. NOC y DIT están estrechamente relacionados, los niveles de herencia se pueden añadir para aumentar la profundidad y reducir la amplitud.

Un alto NOC indica:

- Alta reutilización de la clase base.

- Un muy alto valor de NOC indica que hay fallo en la abstracción de la clase padre, falta algún nivel intermedio. Puede ser necesario introducir otro nivel de herencia.

- Un menor número de defectos debido a la alta reutilización.

Una clase con un alto NOC y un alto WMC indica la complejidad en la parte superior de la jerarquía de clases. La clase está influyendo potencialmente un gran número de clases descendientes. Esto puede ser una señal de un mal diseño. Un rediseño puede ser requerida.

No todas las clases deben tener el mismo número de sub-clases. Las clases más altas en la jerarquía deben tener más subclases que las de más abajo.

Propiedad que trata: Reutiliziación.

CBO - Coupling between object classes (Acoplamiento entre clases).

Es el número de clases acopladas a una clase. Dos clases están acopladas cuando los métodos de una de ellas usan variables o métodos de una instancia de la otra clase. Múltiples accesos a la misma clase se cuenta como un solo acceso. Otros tipos de referencia, como el uso de las constantes, manejo de eventos, el uso de tipos definidos por el usuario, y ejemplificaciones de objetos se ignoran. Si una llamada a un método es polimórfico (ya sea por anulaciones o sobrecargas), todas las clases a las que la llamada puede ir, se incluyen en el recuento.

El valor del CBO nos indica:

- Un alto CBO no es deseable ya que indica que hay un alto acoplamiento lo que es perjudicial para el diseño modular y la reutilización. Cuanto más independiente es una clase más fácil es reutilizarla. Para mejorar la modularidad las parejas de clase entre los objetos deben mantenerse al mínimo.

- Cuanto mayor es CBO, peor es el encapsulamiento y más cuesta mantenerlo.

Un CBO mayor que 14 se considera que es demasiado alto (17).

Propiedad OO que trata: Acoplamiento.

RFC - Response for a class (Respuesta de una clase).

Esta métrica es el número de métodos que pueden ser ejecutados en respuesta a un mensaje recibido por un objeto de esa clase. Cuanto mayor sea RPC, mayor esfuerzo se requiere para su comprobación y más complejo es el diseño. Existen dos variaciones de esta métrica:

RCP = M + R. Sólo considera el primer nivel del árbol de llamadas: número de métodos de una clase más el número de métodos remotos llamados directamente por una clase.

- RCP' = M + R'. Considera todo el árbol de llamadas: número de métodos de una clase más el número de métodos remotos llamados recursivamente por una clase considerando el árbol entero de llamadas.

Siendo M el número de métodos de la clase; R el número de métodos remotos llamados directamente por los métodos de la clase; y R '= número de métodos remotos llamados, de forma recursiva, a través de todo el árbol de llamadas.

Como RFC 'considera todo el árbol de llamadas y no sólo un primer nivel de la misma, que proporciona una medición más exhaustiva del código ejecutado.

El valor del RPC nos indica:

- Clases con un alto RFC serán más complicadas las pruebas y el mantenimiento.
- Clases con un alto RFC son más complejos serán más complejas., mayor será la complejidad de la clase.

Propiedad OO que trata: Complejidad.

LCOM - Lack of Cohesion in Methods (Falta de cohesión de los métodos).

Esta métrica calcula el conjunto de atributos comunes en los métodos de una clase. Cuanto más atributos comunes similares haya entre métodos, mayor cohesión hay en la clase. Se dice que LCOM = 0 si ningún método accede a sus atributos. Por ejemplo, si una clase tiene 6 métodos y 4 de ellos tienen un atributo en común se dice que LCOM = 4.

Es aconsejable tener valores bajos de LCOM. El valor del LCOM nos indica:

- A mayor cohesión mayor encapsulamiento.
- Un valor grande puede indicar que la clase debe dividirse.
- Baja cohesión indica alta complejidad y alta probabilidad de error en el desarrollo.

Propiedad OO que trata: Cohesión.

4.2.3.- MÉTRICAS LK - [Lorenz y Kidd, 1994].

Estas métricas(12) fueron definidas para medir las características estáticas del diseño del software, tales como el tamaño, el uso de herencia y el número de responsabilidades de

una clase. Por ello se dividen en cuatro categorías: tamaño, herencia, valores internos y valores externos. De esta manera, contribuyen a asegurar la mantenibilidad de los productos de software.

Métricas de tamaño. Se centran en cálculo de atributos y de operaciones para una clase individual y promedian los valores para el sistema orientado a objetos en su totalidad.

- **PIM**: Número total de métodos públicos de una clase, que están disponibles como servicios para otras clases. Es decir, los métodos que están disponibles como servicios para otras clases. Esta métrica mide la cantidad de responsabilidad que tiene una clase. Es deseado obtener un número bajo de PIM.

- **NIM**: Número total de métodos de las instancias de una clase, ya sean públicos, privados o protegidos. Esta métrica mide el tamaño de una clase. Las clases grandes suelen ser más complejas y difíciles de mantener que las más pequeñas. Es conveniente tener un valor bajo de NIM.

- **NIV**: Número total de variables (privadas y protegidas) a nivel de instancia que tiene una clase. Un alto valor NIV indica que hay un gran número de variables de instancia lo cual no es aconsejable porque indica que hay demasiado acoplamiento con otras clases. (12) sugieren que las clases son más reutilizables cuando tienen menos variables de instancia.

- **NCM**: Número total de métodos globales de una clase, es decir, métodos visibles por todas las instancias de la clase. El número de métodos de la clase puede indicar la cantidad de elementos comunes que se manejan para todas las instancias. Este número generalmente debe ser relativamente pequeño en comparación con el número de métodos de instancia (NIM).

- **NVV**: Número total de variables globales de una clase, es decir, visibles por todas las instancias de la clase. El número medio de variables de clase debe ser bajo. En general debe haber menos variables de clase que variables de instancia (NIV).

- **LOC**: Número de líneas de código por método. El tamaño de un método es usado para evaluar la comprensibilidad, reusabilidad y mantenibilidad del código.

Métricas de herencia. Se centran en la forma en que se reutilizan las operaciones a lo largo y ancho de la jerarquía de clases.

- **NMO**: Número de métodos sobrecargados, es decir, es el número total de métodos que redefine una subclase de los que haya heredado de una superclase, esto se conoce como reemplazar el método. Con esta métrica se mide la calidad del uso de la herencia, por ejemplo, los métodos reemplazados en niveles profundos pueden indicar que existe problema en el diseño. Es aconsejable tener un valor bajo de NMO.

- **NMI**: Número de métodos heredados por una subclase. Se aconseja tener un número alto de NMI porque indica la fuerza de la subclase en la especialización. Esta métrica mide la calidad del uso de la herencia.

- **NMA**: Número de métodos añadidos, número total de métodos que se definen en una subclase. En las subclases se definen nuevos métodos, que extiende el comportamiento de las superclases. El número de nuevos métodos por lo general debería disminuir a medida que se mueven a través de las capas de la jerarquía. Esta métrica, al igual que las anteriores, mide la calidad del uso de la herencia

- **SIX**: Índice de especialización para cada clase. Mide el grado en que una subclase redefine el comportamiento de una superclase. Indica cuándo hay demasiados métodos redefinidos. Una subclase debe extender el comportamiento de la superclase con métodos nuevos más que redefinir comportamiento. Se propone un valor del 15% para identificar superclases que no tienen mucho en común con sus subclases.

SIX = (Número de métodos sobreescritos * nivel de anidamiento jerarquía) / número total de métodos

Métricas de características internas de las clases . Estas métricas de las características internas de las clases examinan la cohesión y asuntos relacionados con el código, y las métricas orientadas a valores externos examinan el acoplamiento y la reutilización.

- **APPM** Promedio de parámetros por método. Se determina como el cociente entre el número total de parámetros por método y el número total de métodos.

APMM = Nºtotal de parámetros por métodos/ Nºtotal de métodos.

- **NOM** mide el número de mensajes enviados en un método, segregados por el tipo de mensaje (Unarios: sin argumentos; Binarios (un argumento especial como concatenación y funciones matemáticas; y Clave: uno o más argumentos). NOM mide el tamaño del método de una manera relativamente no sesgada.

4.2.4.- MÉTRICAS DE ROBERT MARTIN - [R. Martin, 1994]

(14) Define las siguientes métricas:

- **Ca: Afferent Couplings**: número de clases de otros paquetes que dependen de las clases del propio paquete.

- **Ce: Efferent Couplings**: número de clases dentro del propio paquete que dependen de clases de otros paquetes.

- **I: Instability** = Ce/(Ca+Ce) , métrica comprendida entre [0,1], siendo 0 la máxima estabilidad y 1 máxima inestabilidad.

- Métrica que mide la **abstracción** de un paquete:

    - **NAC** número de clases abstracta en un paquete.

    - **NTC** número total de clases en un paquete.

    Esta métrica está comprendida entre [0,1], siendo 0 completamente concreto y 1 completamente abstracto.

4.2.5.- MÉTRICAS DE LI-HENRY [Li-Henry , 1993]

Estas métricas (15) se centran en la medición de la característica de mantenibilidad del software.

- **MPC** Acoplamiento por paso de mensajes: número de mensajes o métodos invocados enviados por una clase a otras clases del sistema.

- **DAC** Acoplamiento por abstracción de datos: número de atributos de una clase que se referencian desde otra clase.

- **SIZE1:** Es una variación de la tradicional LOC.

- **SIZE2**: número de atributos + métodos locales de una clase.

4.3.- METRICS

La aplicación de métricas se apoya en la herramienta Metrics para agilizar la recogida y cálculo de valores. La herramienta Metrics es un plug-in de Eclipse que calcula diferentes métricas para código "Java" que permite que se conozca continuamente el estado del código. También indicar que es *open-source*.

Podemos establecer unos rangos para cada métrica ya que son configurables desde Eclipse y esta herramienta avisará al desarrollador cuando algún valor se sale del rango establecido.

Además, se puede obtener un informe de los valores de las mediciones de carácter visual y exportable a XML.

El análisis de las dependencias se ve gráficamente por medio de un gráfico dinámico hiperbólico que se puede rotar y hace zoom.

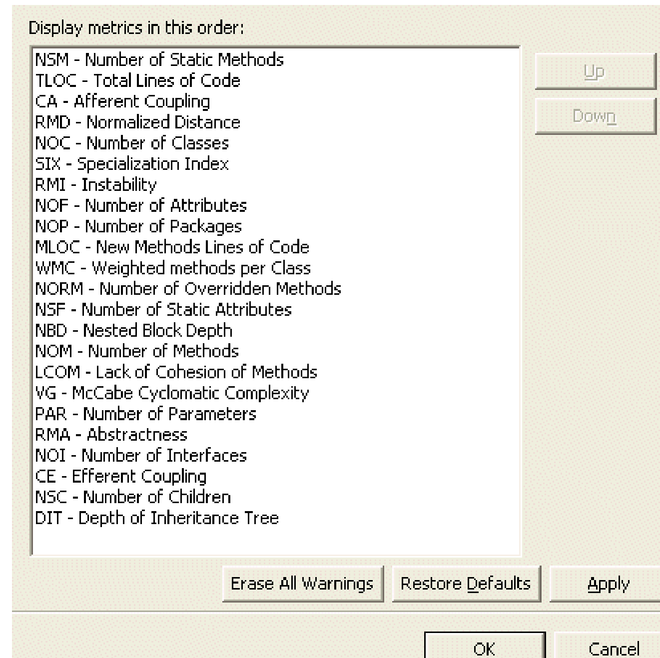


Figura 11. Métricas de Metrics

Las diferentes métricas que cubre Metrics se basan en (16):

- **Number of Classes (NOC)**: Número total de clases.
- **Number of Children (NSC):** Número total de subclases directas de una clase. Una clase que implementa una interface se cuenta como un hijo directo de esa interfaz.
- **Number of Interfaces (NOI):** Número total de interfaces.
- **Depth of Inheritance Tree (DIT)**: Profundidad de herencia del árbol.
- **Number of Overridden Methods (NORM**): Número de métodos sobrecargados. El cálculo de NORM puede personalizarse como se muestra en la imagen siguiente:

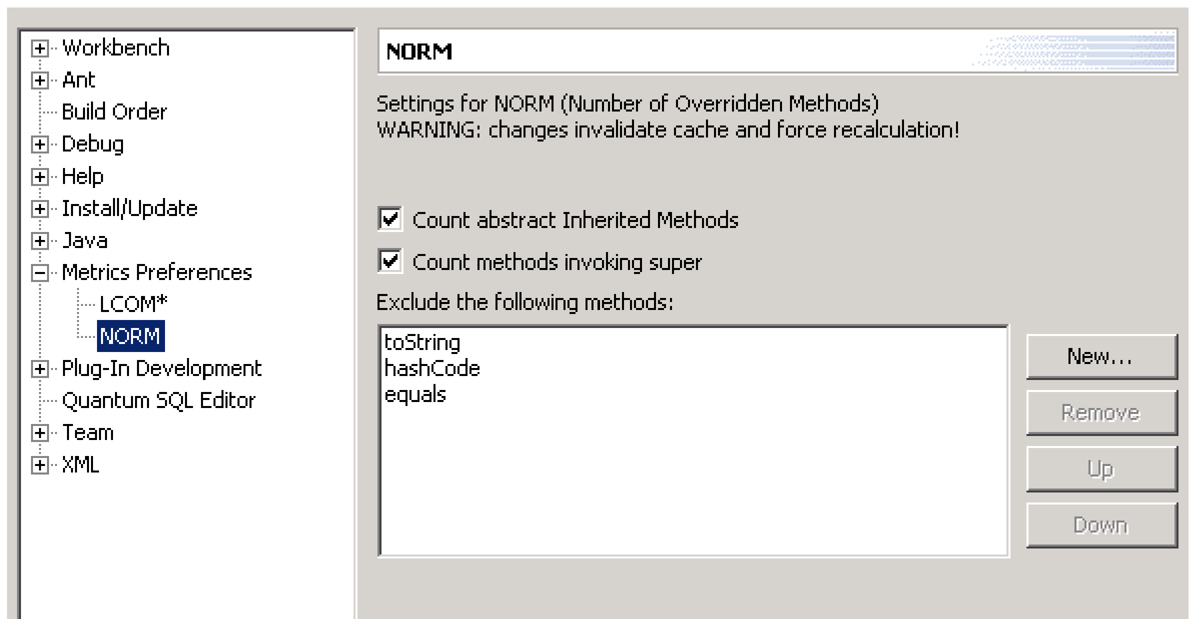


Figura 12. Personalización de la métrica NORM

Aquí se puede controlar si se debe contar los métodos abstractos, métodos que invocan a *super*. Algunos métodos como *toString*, *equals* y *hashCode* puede ser excluidos. También se pueden añadir otros.

- **Number of Methods (NOM)**: Número de métodos definidos.
- **Number of Fields**: Número de campos definidos.
- **Líneas de Código**: Se separan en:
    - **TLOC**: líneas totales del código.
    - **MLOC**: líneas de métodos del código.
- **Specialization Index**: promedio del índice de especialización y se define como: NORM * DIT / NOM. Es un indicador de nivel de clase.
- **McCabe Cyclomatic Complexity (VG).**
- **Weighted Methods per Class (WMC):** Métodos ponderados por clase.
- **Lack of Cohesion of Methods (LCOM*)**: Falta de cohesión de métodos. En la siguiente imagen se ve que se puede escoger si controlar los campos estáticos y/o métodos estáticos o no.

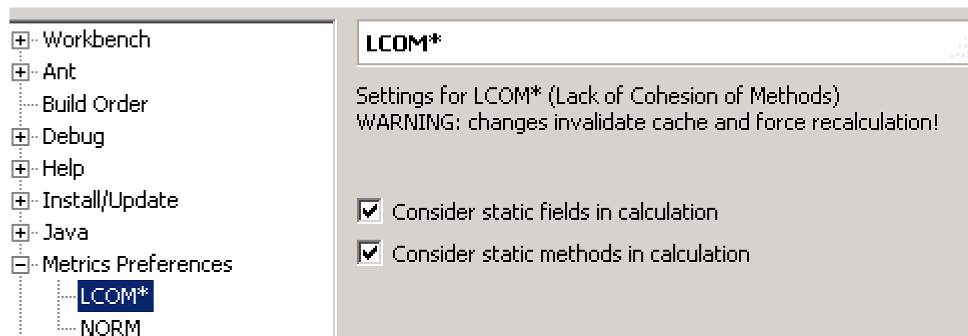


Figura 13. Personalización de la métrica LCOM

Además de estas métricas, esta herramienta considera las siguientes métricas de (14):

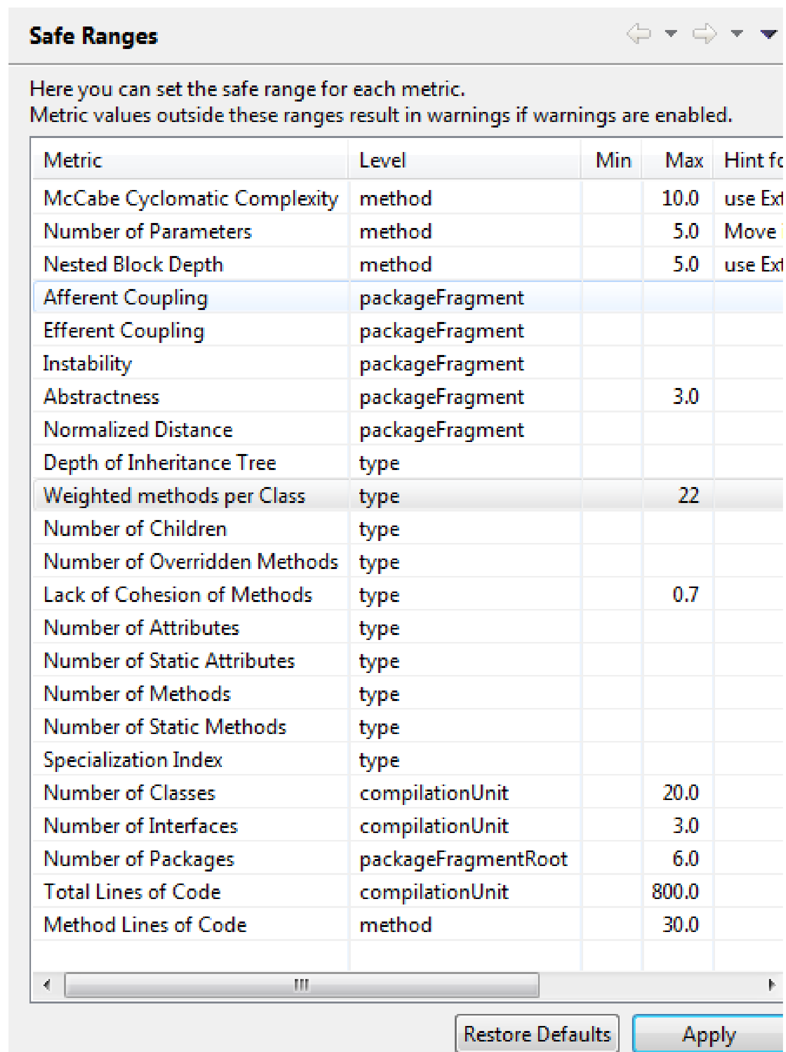
- **Afferent Coupling (Ca):** El número de clases fuera de un paquete que dependen de clases de dentro de un paquete.
- **Efferent Coupling (Ce)**: El número de clases dentro de un paquete que dependen de las clases de fuera de un paquete.
- **Instability (I)**: Ce / (Ca + Ce)
- **Abstractness (A):** Nñumero de clases abstractas (e interfaces) dividido por el número total de tipos de un paquete.
- **Normalized Distance from Main Sequence:**| A + I - 1 |, este número debe ser pequeño, cerca del cero indica buen diseño de paquetes.

En el apartado M.3. Metrics parece la instalación y el manual de usuario de esta herramienta.

4.4.- EJEMPLO PRÁCTICO DE APLICACIÓN DE METRICS

Al igual que se ha hecho con las herramientas JaCoCo y Checkstyle, se aplicará esta herramienta al proyecto.

En primer lugar se ha personalizado el rango máximo que las métricas no deben sobrepasar. Un ejemplo de posible rangos son los siguientes:



Metric	Level	Min	Max	Hint fo
McCabe Cyclomatic Complexity	method		10.0	use Ext
Number of Parameters	method		5.0	Move
Nested Block Depth	method		5.0	use Ext
Afferent Coupling	packageFragment			
Efferent Coupling	packageFragment			
Instability	packageFragment			
Abstractness	packageFragment		3.0	
Normalized Distance	packageFragment			
Depth of Inheritance Tree	type			
Weighted methods per Class	type		22	
Number of Children	type			
Number of Overridden Methods	type			
Lack of Cohesion of Methods	type		0.7	
Number of Attributes	type			
Number of Static Attributes	type			
Number of Methods	type			
Number of Static Methods	type			
Specialization Index	type			
Number of Classes	compilationUnit		20.0	
Number of Interfaces	compilationUnit		3.0	
Number of Packages	packageFragmentRoot		6.0	
Total Lines of Code	compilationUnit		800.0	
Method Lines of Code	method		30.0	

Figura 14. Establecimiento de rango de métricas

Una vez establecidos los rangos se observan los resultados.

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum
McCabe Cyclomatic Complexity (avg/max per method)		1,759	1,36	7	/game-of-life-master/src/edu/macalest
Number of Parameters (avg/max per method)		0,704	0,853	3	/game-of-life-master/src/edu/macalest
Nested Block Depth (avg/max per method)		1,574	1,011	4	/game-of-life-master/src/edu/macalest
Afferent Coupling (avg/max per packageFragment)		0	0	0	/game-of-life-master/src/edu/macalest
Efferent Coupling (avg/max per packageFragment)		1	1	2	/game-of-life-master/test/edu/macales
Instability (avg/max per packageFragment)		1	0	1	/game-of-life-master/src/edu/macalest
Abstractness (avg/max per packageFragment)		0,071	0,071	0,143	/game-of-life-master/src/edu/macalest
Normalized Distance (avg/max per packageFragment)		0,071	0,071	0,143	/game-of-life-master/src/edu/macalest
Depth of Inheritance Tree (avg/max per type)		2	1,826	6	/game-of-life-master/src/edu/macalest
Weighted methods per Class (avg/max per type)	95	10,556	9,251	25	/game-of-life-master/src/edu/macalest
Number of Children (avg/max per type)	1	0,111	0,314	1	/game-of-life-master/src/edu/macalest
Number of Overridden Methods (avg/max per type)	1	0,111	0,314	1	/game-of-life-master/src/edu/macalest
Lack of Cohesion of Methods (avg/max per type)		0,267	0,302	0,667	/game-of-life-master/src/edu/macalest
Number of Attributes (avg/max per type)	17	1,889	1,912	6	/game-of-life-master/src/edu/macalest
Number of Static Attributes (avg/max per type)	4	0,444	0,685	2	/game-of-life-master/src/edu/macalest
Number of Methods (avg/max per type)	53	5,889	3,725	14	/game-of-life-master/src/edu/macalest
Number of Static Methods (avg/max per type)	1	0,111	0,314	1	/game-of-life-master/src/edu/macalest
Specialization Index (avg/max per type)		0,056	0,157	0,5	/game-of-life-master/src/edu/macalest
Number of Classes (avg/max per packageFragment)	9	4,5	2,5	7	/game-of-life-master/src/edu/macalest
Number of Interfaces (avg/max per packageFragment)	1	0,5	0,5	1	/game-of-life-master/src/edu/macalest
Number of Packages	2				
Total Lines of Code	530				
Method Lines of Code (avg/max per method)	335	6,204	8,04	47	/game-of-life-master/src/edu/macalest

Figura 15. Resultados de Metrics

Se puede observar que dos de las métricas superaran los valores que se establecieron como máximo.

Desplegando una de las métricas se puede ver en qué clase se encuentra la superación del valor establecido.

Weighted methods per Class (avg/max per type)	95	10,556	9,251	25	/
src	81	11,571	10,266	25	/
edu.macalester.comp124.life	81	11,571	10,266	25	/
GameBoard.java	25	25	0	25	/
MainWindow.java	26	13	11	24	/
LifeComponent.java	21	21	0	21	/
InvalidBoardException.java	4	4	0	4	/
Conway.java	3	3	0	3	/
RuleSet.java	2	2	0	2	/
test	14	7	0	7	/

Figura 16. Valores de métrica desplegada

En este análisis se observa todas las métricas que esta herramienta analiza. El WMC estaba como máximo 22 por lo que al hacer uso de la herramienta nos indica que hay dos clases que superan el umbral. También otra de las métricas que se supera es MLOC ya que se indica que el número máximo de líneas en métodos es de 30 y el programa lo supera.

Hay que tener cuidado con los límites que se establecen para las métricas ya que, por ejemplo, 30 líneas de métodos en el código es una cantidad ridícula para un proyecto. Por lo tanto, se debe poner los valores de máximo correctamente.

Gracias a los resultados de las métricas se obtiene unos resultados positivos:

- Se pueden seleccionar los componentes más propensos a errores y concentrar en ellos un mayor esfuerzo de detección y evaluación.

- Sirve de feedback a los diseñadores ya que con ellos pueden saber cómo está el software y si tienen que rediseñarlo e implementarlo para obtener mejores resultados. Por ejemplo, un alto WMC indica que una clase debería ser restructurada en clases más pequeñas para disminuir su complejidad y su propensión a errores. WMC y NOC altos indican complejidad en la parte superior por lo que existe un mal diseño.

- Se obtiene un informe con información cuantitativa y cualitativa para detectar los defectos de una forma mejor.

CAPÍTULO 5. CONCLUSIONES

5.1. CONCLUSIONES PRINCIPALES

A lo largo de este Trabajo Fin de Grado se ha **analizado el uso de diferentes herramientas gratuitas de análisis estático de código y de control de cobertura de pruebas en aplicaciones Java** consiguiendo de esta forma cumplir el objetivo principal del presente trabajo.

Analizando más concretamente el trabajo realizado, se puede observar que se ha analizado diferentes herramientas:

- **Checkstyle**. Con esta herramienta he podido ver diferentes estilos de codificación y darme cuenta la importancia que supone seguir un mismo estilo de codificación en un mismo proyecto. Se debe usar porque normalmente el equipo de desarrollo se compone de más de una persona entonces se debe acordar un estándar de codificación. Esta herramienta ayuda a definir y aplicar esas normas comunes de forma fácil. Seguir un mismo estilo de codificación supone asegurar la calidad y facilitar el mantenimiento.

- **JaCoCo**. Al aplicar el control de cobertura con esta herramienta me di cuenta que puede haber casos en los que no todas las pruebas que se diseñan son suficientes para probar todo el código o por lo menos cubrir un porcentaje del mismo. Es importante cubrir las partes del código más propensos a errores. Gracias a esta herramienta nos aseguramos qué parte del código está siendo probado y cuál no, detectando posibles fallos y errores, haciendo así que el software desarrollado tenga una gran calidad.

- **Metrics**. De esta herramienta se puede destacar que es una herramienta muy completa y fácil de usar ya que muestra al usuario los resultados de evaluación de proyectos Java en el mismo entorno de trabajo. El inconveniente que le veo es que no permite al usuario añadir más métricas de evaluación. Con el uso de esta herramienta nos aseguramos que el programa esté bien diseñado, tenga buena calidad, es decir, nos aseguramos que los productos software que se obtienen satisfagan las necesidades y expectativas de los usuarios. Un

Estas tres herramientas se han automatizado en un mismo entorno de desarrollo que ha sido Eclipse, facilitando al desarrollador su labor. Al estar integrado todo en un mismo entorno se facilita un entorno de trabajo que permite a los desarrolladores detectar

situaciones que puedan afectar a la comprensión y mantenibilidad del software, como así, llevar un registro de los valores de las métricas que les permitan predecir para adecuar el diseño del programa para un software de calidad.

En resumen, en este proyecto no sólo se ha ayudado a conocer cómo se evalúa la calidad del software, sino que además, se ha aprendido nuevos conceptos, nuevas metodologías de programación y el uso de nuevas tecnologías.

5.2. TRABAJOS FUTUROS

A partir del trabajo realizado, se pueden considerar distintos caminos que se pueden seguir para continuar y mejorar el trabajo:

- Con respecto a los estilos de codificación se puede realizar un experimento que consista en un programa que tenga dos versiones diferentes.

    1. Una versión sin verificación de estilo, es decir, tal cual la han hecho.
    2. Una segunda versión en la que se le haya pasado la herramienta detectando la violación de reglas.

  Una vez se tenga las dos versiones se les pasa a las personas para qué evalúen cuál de las dos versiones es más fácil hacer cambios en el código, cuál es más fácil de mantener. Además, se les pedirá que hagan cambios en el código para comprobar cuánto les cuesta hacerlos.

- En cuanto al control de cobertura de las pruebas, se puede realizar un experimento que consista en usar herramientas por medio de un programa controlado y pedir a las personas que están haciendo el experimento a que intente hacer las pruebas. Después de eso se realiza una comparación entre lo que ha hecho la gente viendo:

    1. Si se ha cubierto todo e código del programa.
    2. Si existen pruebas que ha pasado por el mismo sitio del código.

- En relación a las métricas se podría realizar una revisión de trabajos que se han realizado con métricas para predecir si hay existe un buen diseño o no del programa. En las herramientas por defecto vienen unos valores concretos a las distintas métricas, estos valores no se saben si son fiables o no, por ello se podría hacer un análisis de qué valores máximo o mínimo se pueden otorgar a las diferentes métricas para saber si los resultados de las mismas en el programa en el que se aplica son buenos o malos.

PARTE III
DIAGRAMAS

CAPÍTULO 6. DIAGRAMAS

El programa Java usado en este proyecto este capítulo se ha obtenido de repositorio GitHub. Se puede encontrar en el siguiente enlace:

https://github.com/mac-comp124-f13/game-of-life

Además es posible consultarlo en el soporte digital que se adjunta al proyecto.

PARTE IV
PLIEGO DE CONDICIONES

CAPÍTULO 7.
PLIEGO DE CONDICIONES

En este apartado se detallan las especificaciones del equipamiento y el material informático necesario para la realización de este trabajo.

1.- HARDWARE

- Ordenador portátil MacBook.
- Microprocesador: Genuine Intel® CPU 2GHz
- Memoria RAM: 2GB
- Disco duro: 500 Gb

2.- SOFTWARE

- Sistema Operativo: Windows 7 Professional N (una partición con Windows).
- Eclipse Standard. Versión: Kepler Service Release 2
- Checkstyle Versión 6.7
- JaCoCo Version 2.3.2
- Metrics

PARTE V
MANUAL DE USUARIO

M.1.- CHECKSTYLE

M.1.1- INSTALACIÓN

Esta herramienta se puede instalar por varios métodos:

1. **A través de Eclipse Marketplace.**

   a. Dentro de Eclipse: *Help > Eclipse Marketplace*.

   b. Buscar *Checkstyle* y hacer clic en Instalar.

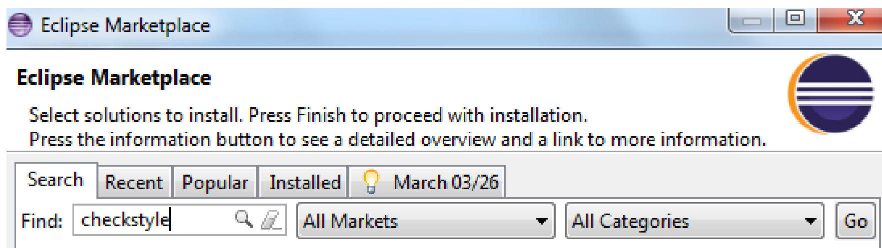


Figura 17. Instalación Eclipse Marketplace (I)

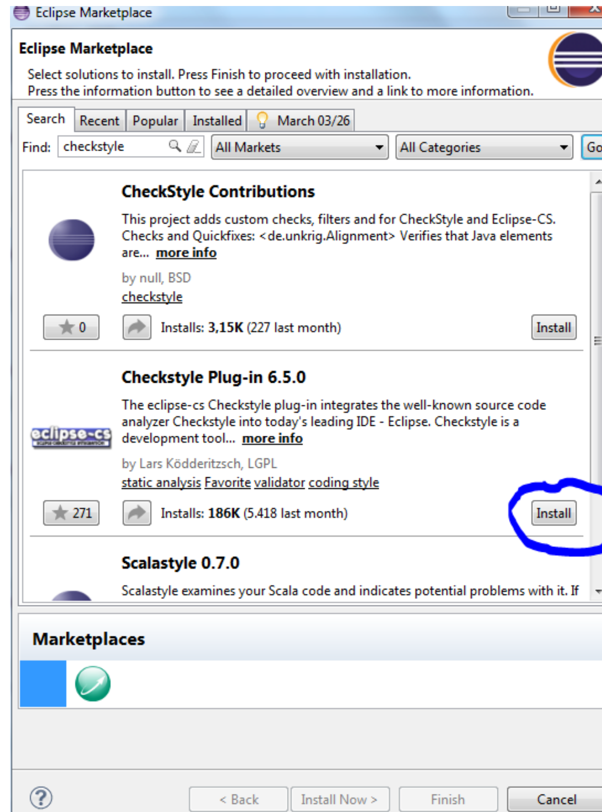


Figura 18. Instalación Eclipse Marketplace (II)

c. Confirmar la selección de características para instalar.

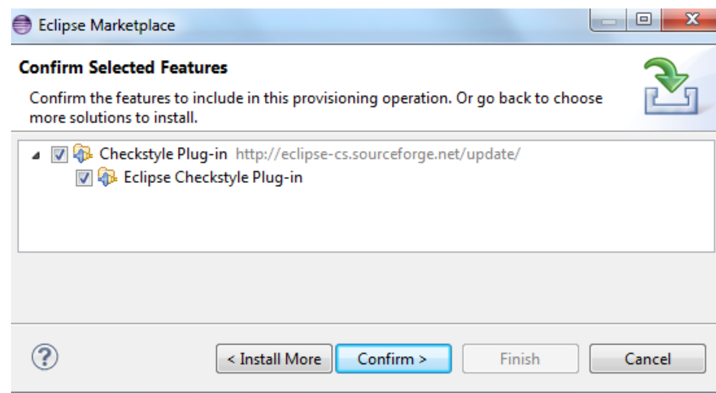


Figura 19. Instalación Eclipse Marketplace (Confirmación)

d. Leer/aceptar los términos de licencia.

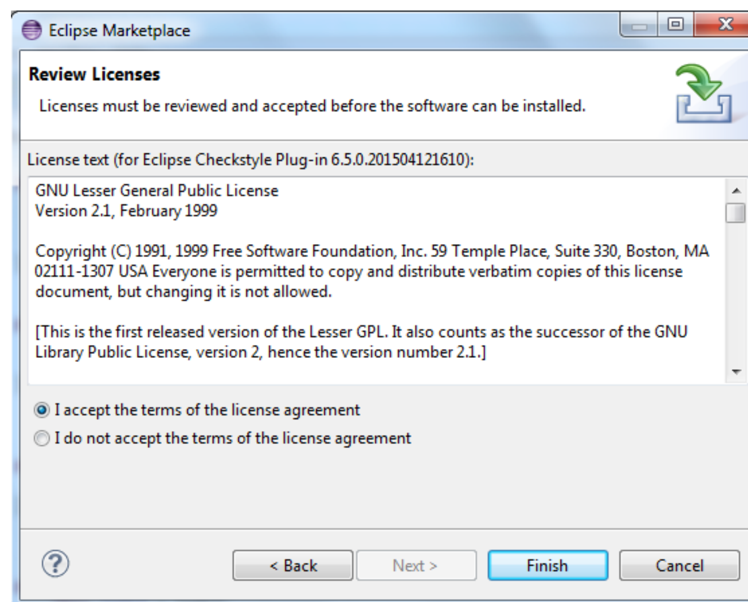


Figura 20. Instalación Eclipse Marketplace (Términos de licencia)

e. Eclipse descargará los archivos necesarios.

f. Archivo descargado, Eclipse preguntará acerca de la instalación de contenido sin firmar.

g. Reiniciar Eclipse.

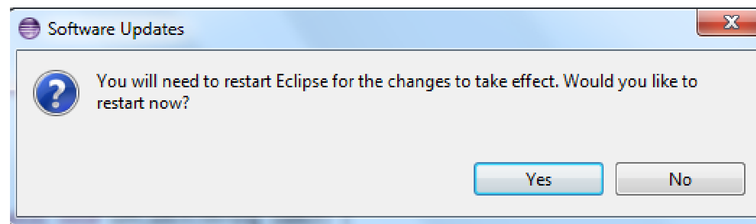


Figura 21. Instalación Eclipse Marketplace (Reinicio Eclipse)

2. A través del sitio de actualización.

a. Dentro de Eclipse: *Help → Install new software*

b. Escribir el siguiente sitio URL: `http://eclipse-cs.sf.net/update`

c. Confirmar la selección de características para instalar.

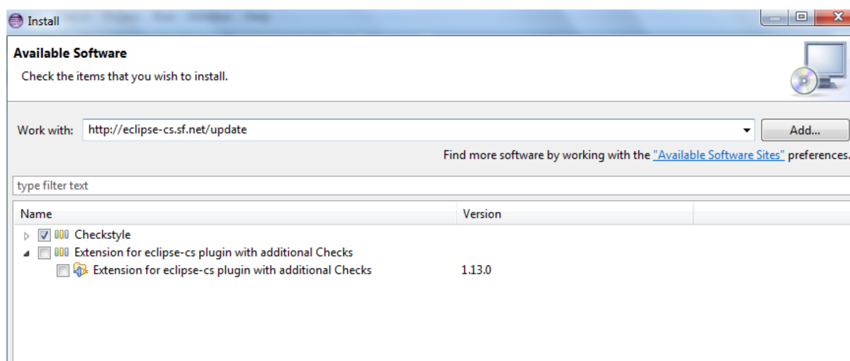


Figura 22. Instalación Eclipse a través del sitio de actualización (I)

d. Leer/aceptar los términos de licencia.

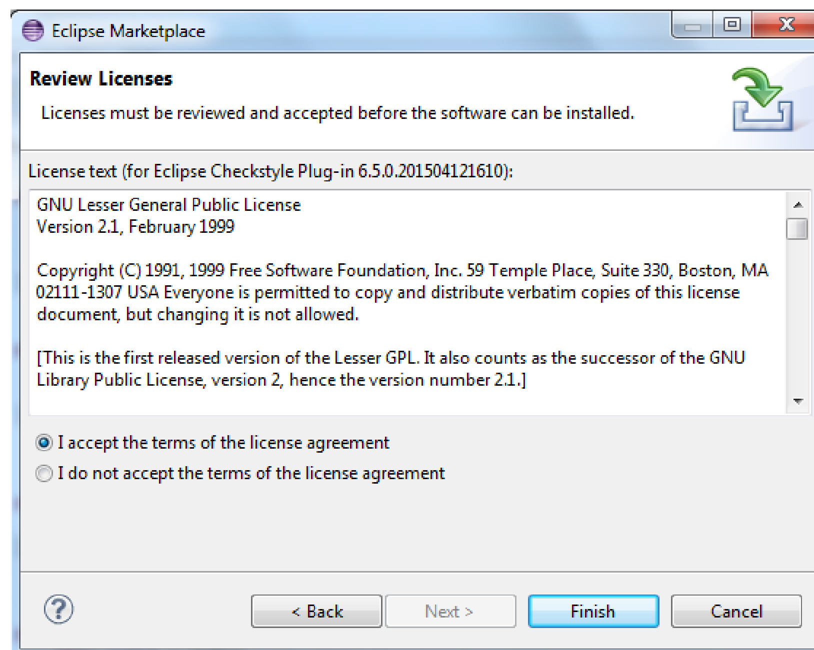


Figura 23. Instalación Eclipse a través del sitio de actualización (II)

e. Eclipse descargará los archivos necesarios.

f. Archivo descargado, Eclipse preguntará acerca de la instalación de contenido sin firmar.

g. Reiniciar Eclipse.

3. A través de un archivo descargado.

a. Descargar del sitio de actualizaciones el paquete CheckStyle Plugin

b. En Eclipse ir a: *Help* → *Install new Software*

c. Pulsar *add…*, luego *File* y seleccionar el archivo descargado.

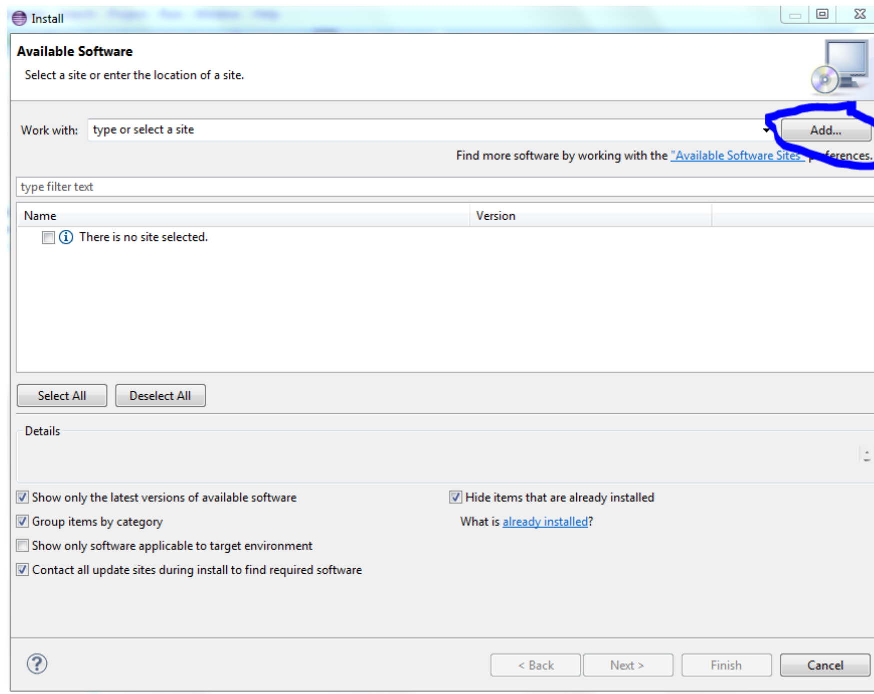


Figura 24. Instalación Eclipse a través de un archivo de descarga (I)

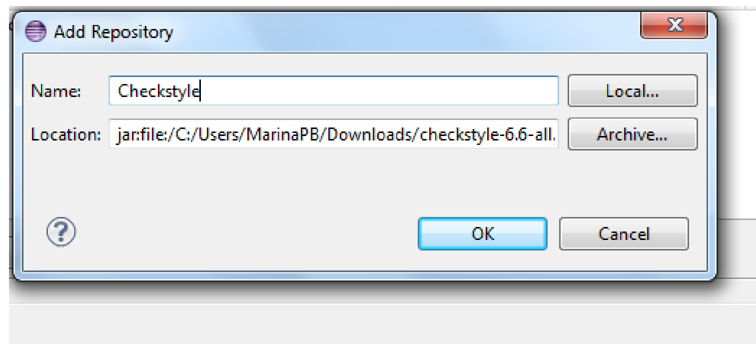


Figura 25. Instalación Eclipse a través del sitio de actualización (II)

d. Seleccionar el plug-in de CheckStyle y clic en instalar.

e. Confirmar la selección de características para instalar.

f. Leer/aceptar los términos de licencia.

g. Eclipse descargará los archivos necesarios.

h. Archivo descargado, Eclipse preguntará acerca de la instalación de contenido sin firmar.

i. Reiniciar Eclipse.

M.1.2.- CONFIGURACIÓN BÁSICA.

1.- En primer lugar seleccionamos las vistas, para ello se selecciona la opción de menú *Windows > Show View > Other* … se abrirá la siguiente ventana y se escoge las vistas que se quiera.

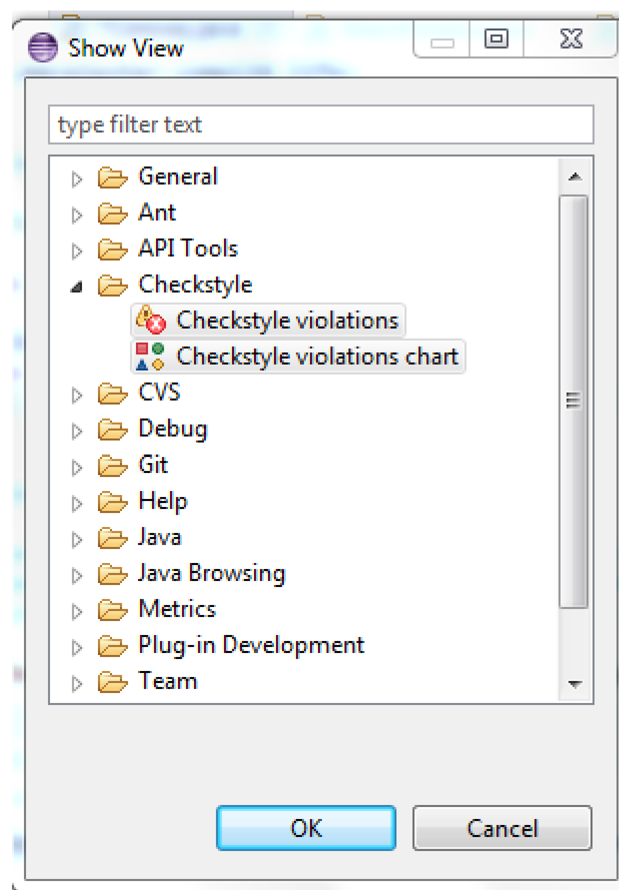


Figura 26. Selección de View de Checkstyle

Seleccionando todas las vistas, se ve como sigue:

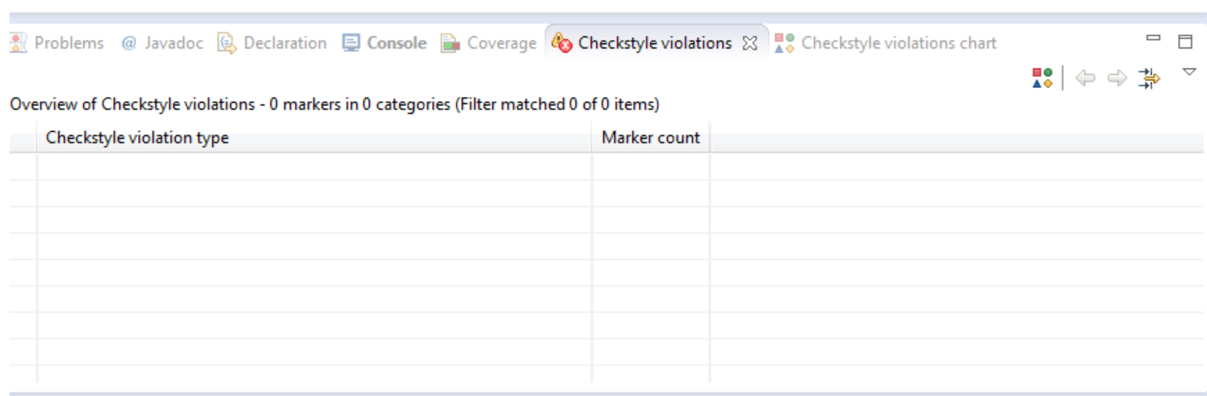


Figura 27. View de Checkstyle

2.- Una vez tenemos las vistas el siguiente paso es indicar el conjunto de reglas que se quiere usar (*Windows > Preferences…*). En Checkstyle vienen incluido el estándar de Sun, de Google y de Sun adaptado al formateo de Eclipse.

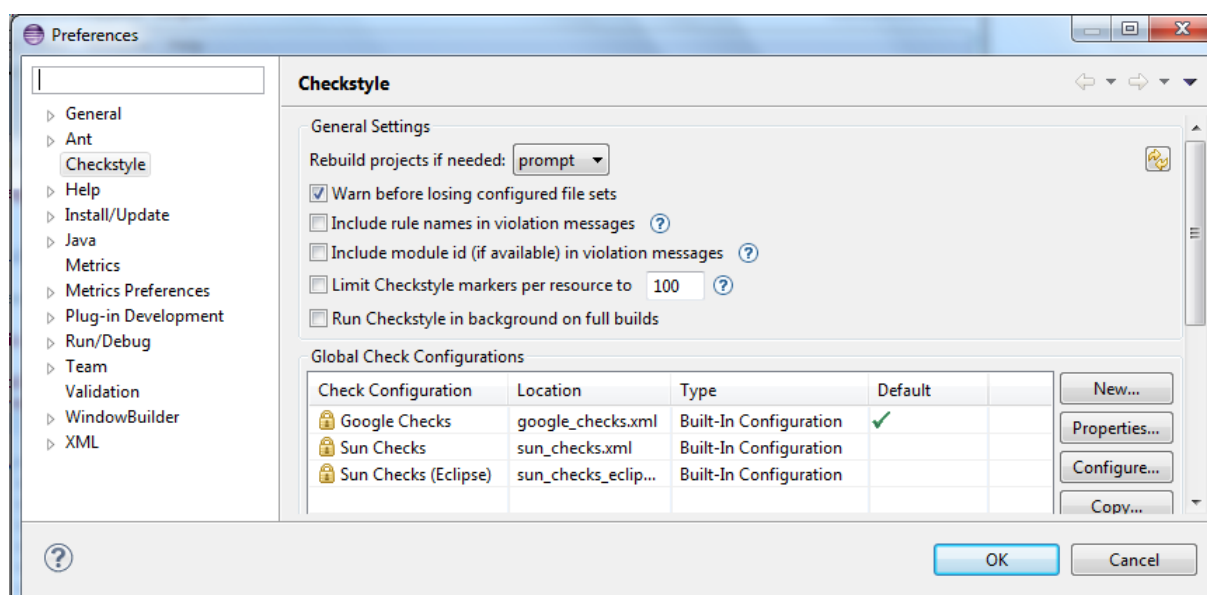


Figura 28. Preferencias de Checkstyle

Nota: si se tiene un conjunto de reglas personalizables aquí es el momento de incluirlo.

3.- El tercer paso consiste en habilitar la verificación del código por parte de CheckStyle.

Para ello le damos al botón derecho sobre el nombre del proyecto y cliqueamos en *Properties > Checkstyle* y seleccionamos la opción "*Checkstyle active for this Project*".

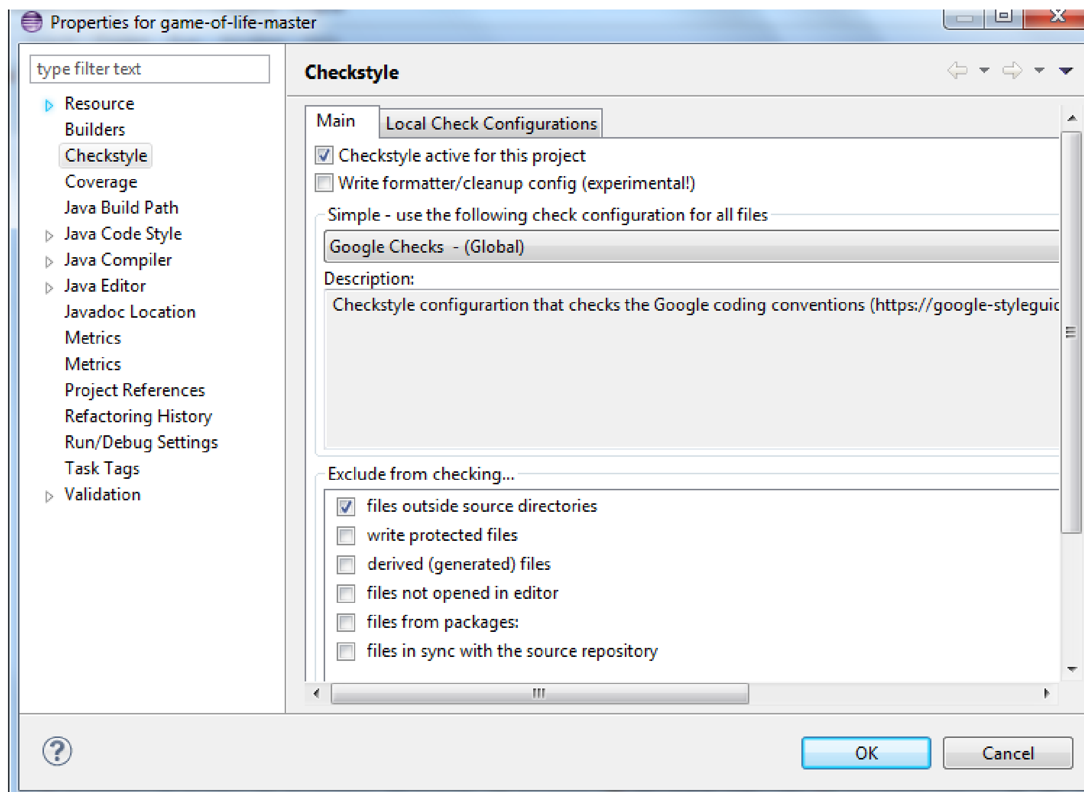


Figura 29. Propiedades de Ckeckstyle

En ese momento habrá que reconstruir el proyecto para que se lleve a cabo la verificación del código.

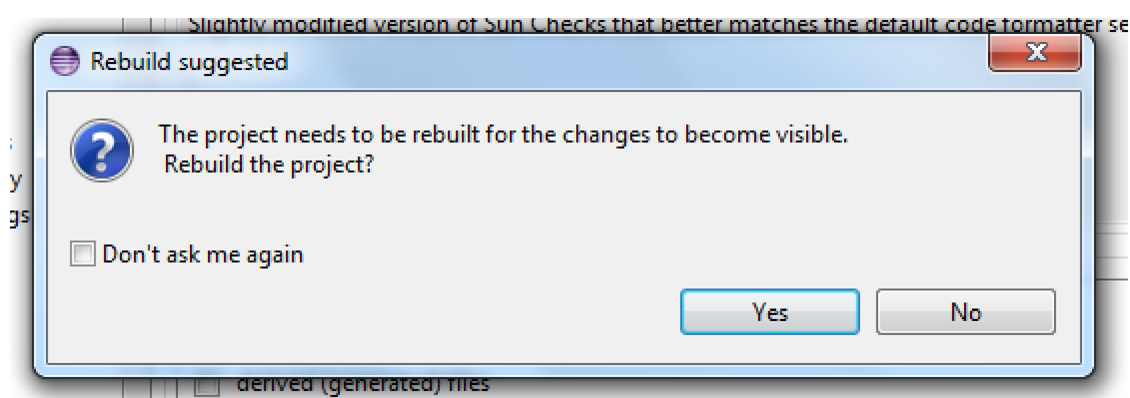


Figura 30. Reconstruir el proyecto

4.- Una vez reconstruido el proyecto podemos ver las diferentes violaciones de las reglas que se han obtenido en la pestaña de "Checkstyle violations" si las hay. En el caso de este proyecto nos encontramos con las siguientes violaciones:

Problems | Javadoc | Declaration | Console | Coverage | Checkstyle violations | Checkstyle violations chart

Overview of Checkstyle violations - 545 markers in 13 categories (Filter matched 545 of 545 items)

Checkstyle violation type	Marker count
Import statement is in the wrong order. Should be in the 'X' group.	2
'X' should be separated from previous statement.	1
At-clause should have a non-empty description.	5
Wrong lexicographical order for 'X' import.	2
Empty line should be followed by <p> tag on the next line.	2
Usar la importación con '.*' debería evitarse - X.	7
El nombre 'X' debe coincidir con el patrón 'X'.	29
La construcción 'X' debe usar '{}' (llaves).	4
X en el nivel de sangrado X no está al nivel correcto, X	268
Javadoc comment at column X has parse error. Details: X while parsing X	9
First sentence should be present.	11
Bloque X vacío.	1
X el descendiente en el nivel de sangrado X no está al nivel correcto, X	204

Figura 31. Vista de violaciones de regla de CheckStyle del proyecto

Si pinchamos en el cada uno de los tipos de despliega el tipo y muestra un listado detallado de los detalles de las diferentes localizaciones en las que se ha detectado esta violación.

Problems | Javadoc | Declaration | Console | Coverage | Checkstyle violations | Checkstyle violations chart

Details of Checkstyle violation "X en el nivel de sangrado X no está al nivel correcto, X" - 268 occurrences

Resource	In Folder	Line	Message
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	95	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	15	member def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	25	method def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	29	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	35	method def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	37	member def type en el nivel de sangrado 8 no está al...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	39	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	45	method def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	48	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	54	method def modifier en el nivel de sangrado 4 no es...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	60	method def rcurly en el nivel de sangrado 4 no está ...
ConwayTest.java	/game-of-life-master/test/edu/macalester/comp12...	66	method def modifier en el nivel de sangrado 4 no es...

Figura 32. Detalle de violación de una regla

Y pulsando sobre cada una de ellas se accede a la construcción que produce la violación.

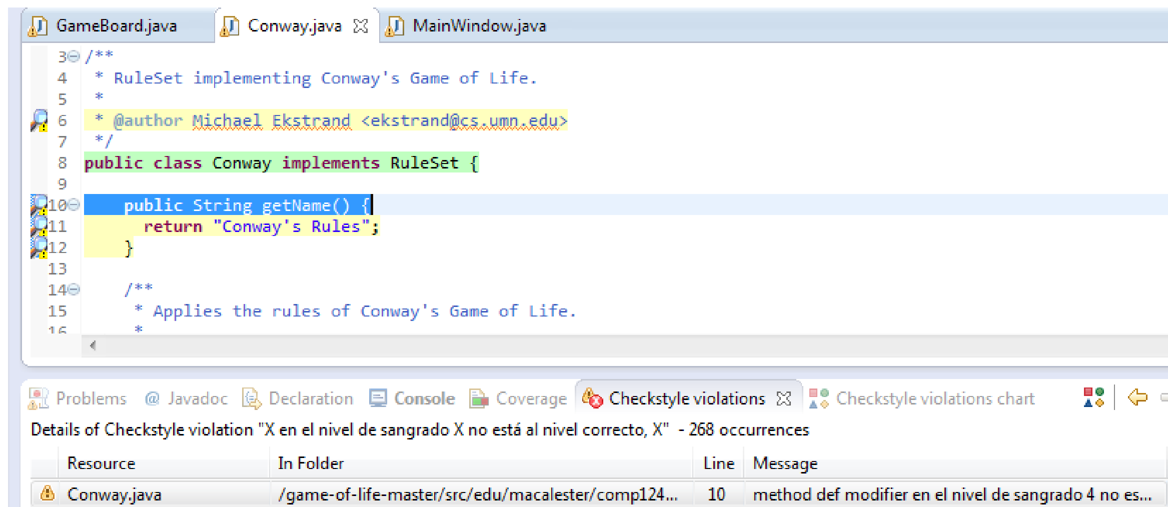


Figura 33. Lugar del código donde se produce la falta

Ya sabemos dónde exactamente se viola la regla y donde.

También se puede ver estas violaciones en forma de gráfico que se puede guardar en formato PNG:

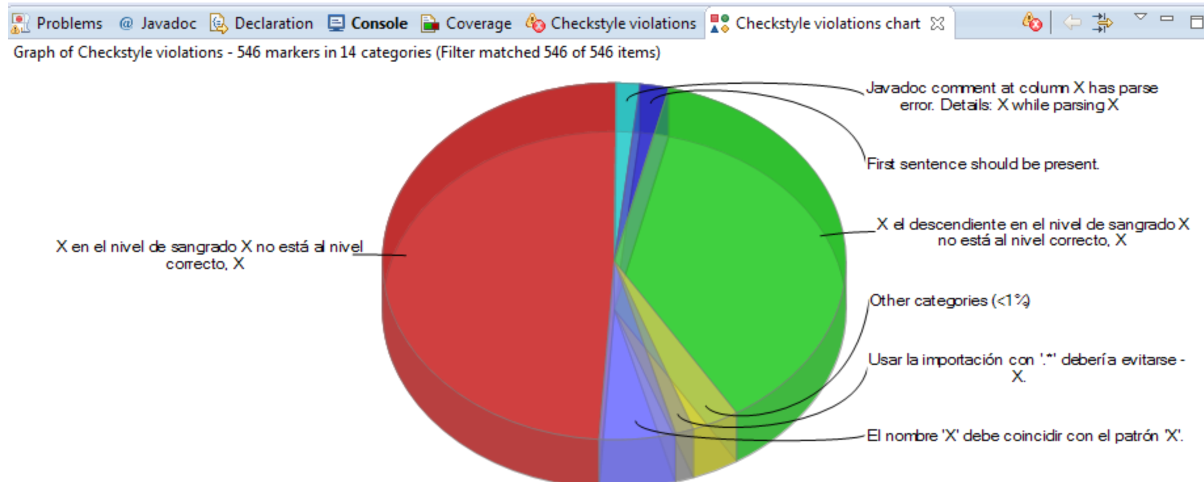


Figura 34. Gráfico generado por Checkstyle

Esta es la configuración básica, es decir, se ha utilizado uno de los estándares que ya vienen en la herramienta. Puede ser que tengamos nuestras propias reglas de estilo, por ello, el siguiente paso es crear reglas propias para que se adapten a su propio estilo de codificación.

M.1.3.- CONFIGURACIÓN PERSONALIZABLE DE UN PROYECTO.

Checkstyle permite crear reglas personalizables para que el proyecto cumpla una serie de reglas. Los pasos a seguir son los siguientes:

1.- Abrir la ventana de preferencias de Eclipse: *Window > Preferences…*

2.- Una vez en preferencia ir a la sección de Checkstyle.

La preferencias tienen que tener un aspecto como la siguiente imagen. En la parte superior hay algunos parámetros generales que se explican con más detalles en el apartado M.2. En el centro o área inferior se muestra las configuraciones que en ese momento están disponibles. Las configuraciones que sean de tipo "Built-it" no se pueden modificar ni eliminar pero sí copiar para, a partir de esas configuraciones, hacer la tuya propia.

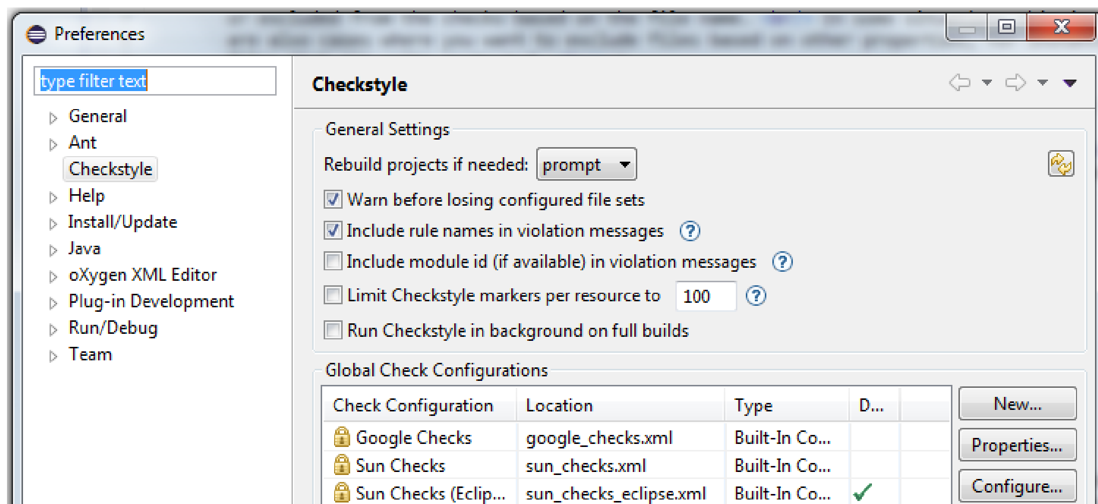


Figura 35. Configuración personalizable Checkstyle

3.- Hacer clic en *New…* para crear una nueva configuración.

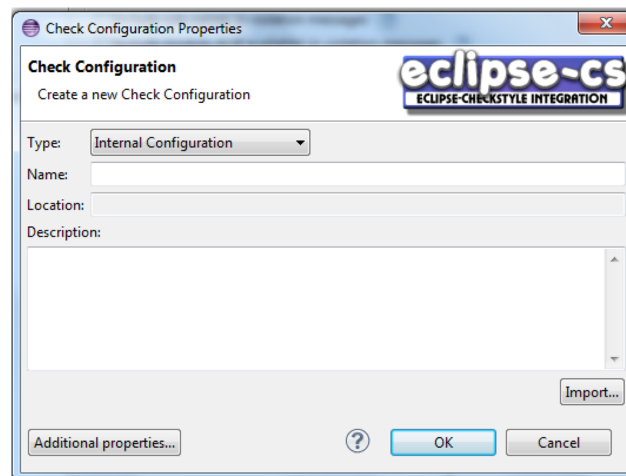


Figura 36. Nueva configuración

Hay que escoger el tipo de configuración que se quiere crear:

- Internal configuration.
- External contiguration file.
- Remote configuration.
- Project Relative configuration.

Éstas se explican en el apartado M.1.4.2. De momento sólo se considera la configuración interna. El campo de texto *Localitation* aparece atenuado porque las configuraciones internas se guardan dentro del plug-in. Hay que poner un nombre a la configuración y una descripción, esto último es opcional.

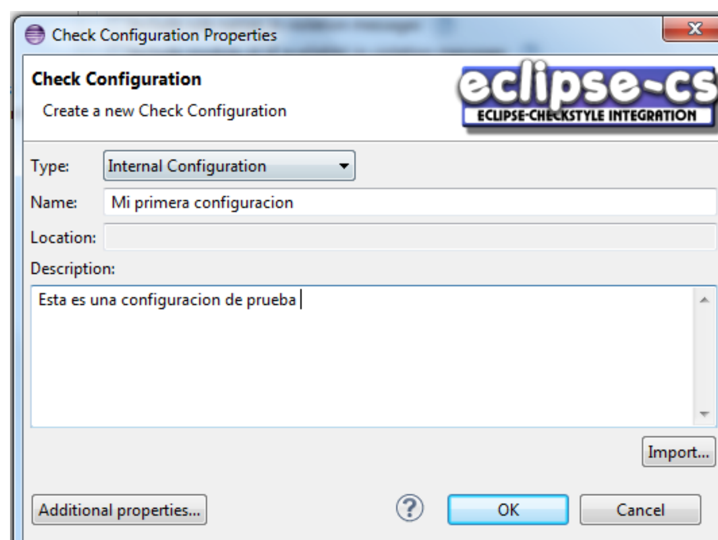


Figura 37. Propiedades nueva configuración

Cuando se termine de rellenar los campos se hace clic en OK y se vuelve a la pantalla principal de configuración.

Ahora la configuración aparecerá en la tabla de configuraciones disponibles.

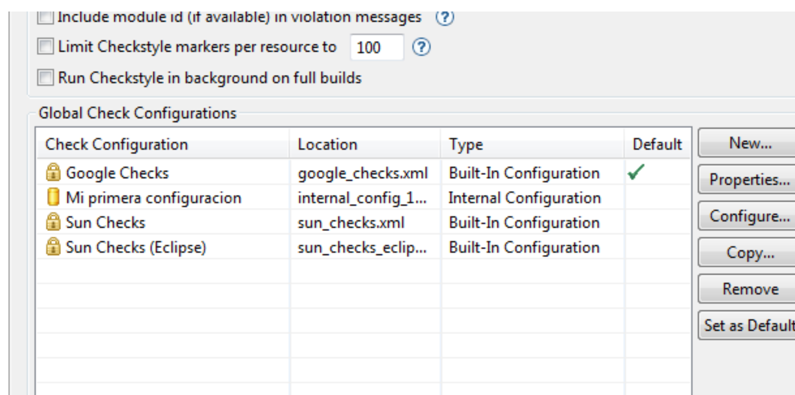


Figura 38. Nueva configuración creada

Para volver al diálogo anterior se selecciona la configuración en la tabla donde aparece y se hace clic en *Properties*… La configuración creada está vacía ya que no se ha definido aún nada.

Puede volver a este diálogo seleccionando la configuración y pulsar *Propiedades ...* Nuestra configuración es (por supuesto) bastante vacío porque todavía no hemos definido ninguna comprobación.

4.- Seleccionar la configuración creada y pulsar en Configurar… para abrir el editor de configuración.

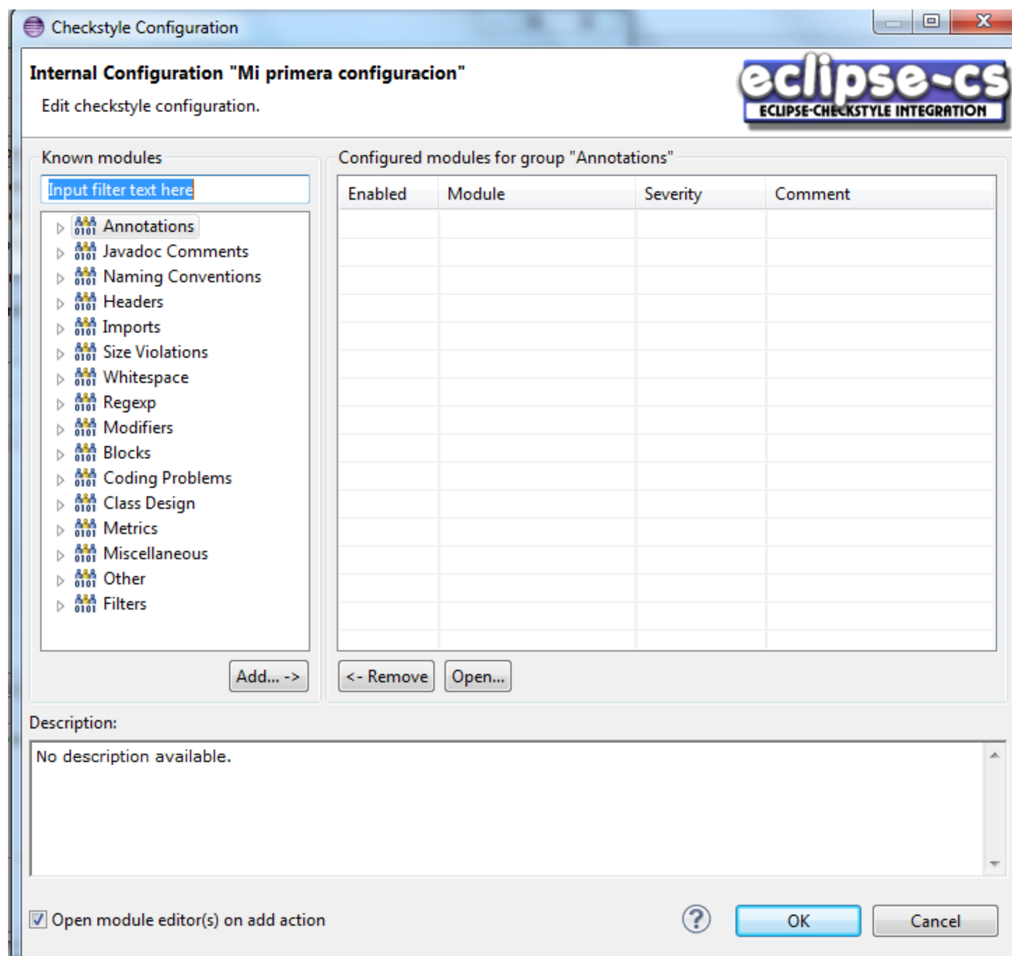


Figura 39. Editor de la nueva configuración

En la parte izquierda aparecen todos los módulos del plug-in. En la parte derecha se muestra lo que contiene cada módulo. Ahora están vacíos. En el cuadro de texto se describe el módulo que se ha seleccionado.

Si se quiere añadir un módulo hay que pulsar sobre *Add…->* Por ejemplo, se selecciona Jacadoc Comments y se despliega. Se escoge *Method Javadoc* y doble clic sobre él.

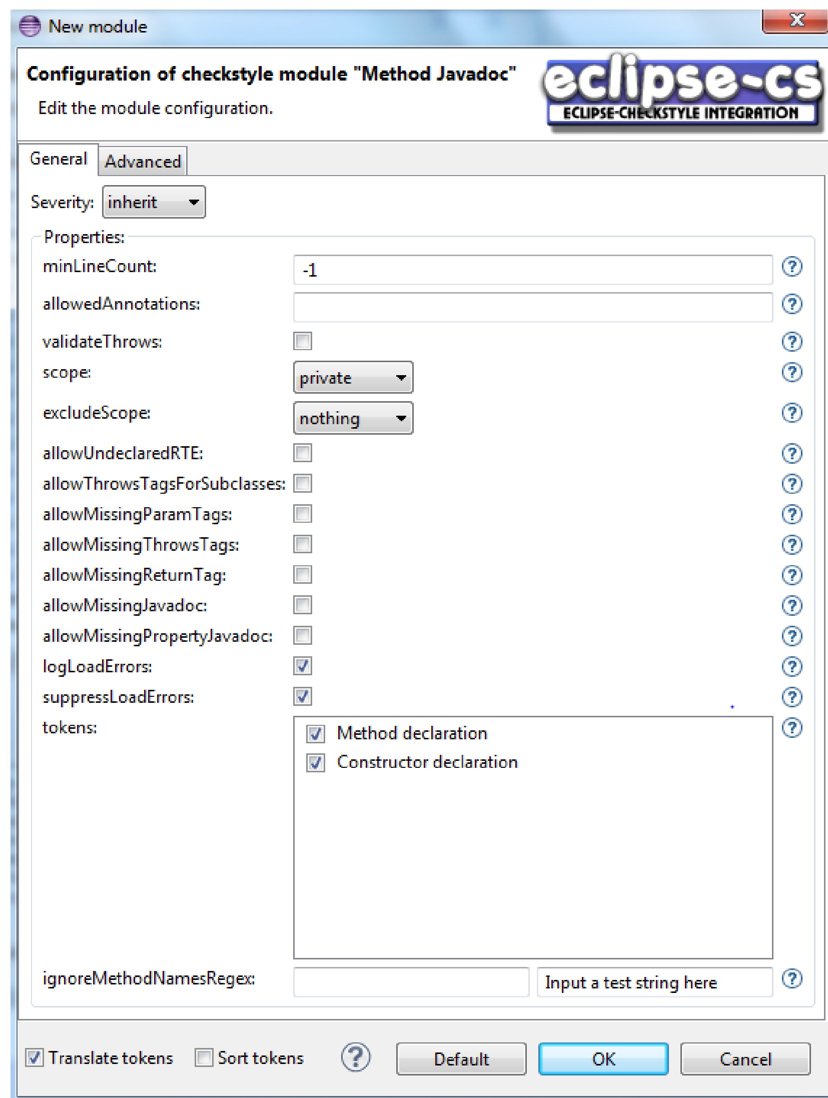


Figura 40. Configuración del nuevo módulo de la nueva configuración

Ahora se puede personalizar como se quiera. Se puede escoger el nivel de gravedad o rigor. Esto está en todos los módulos.

Se acepta y aparecerá en dentro de los módulos de Javadoc Comments.

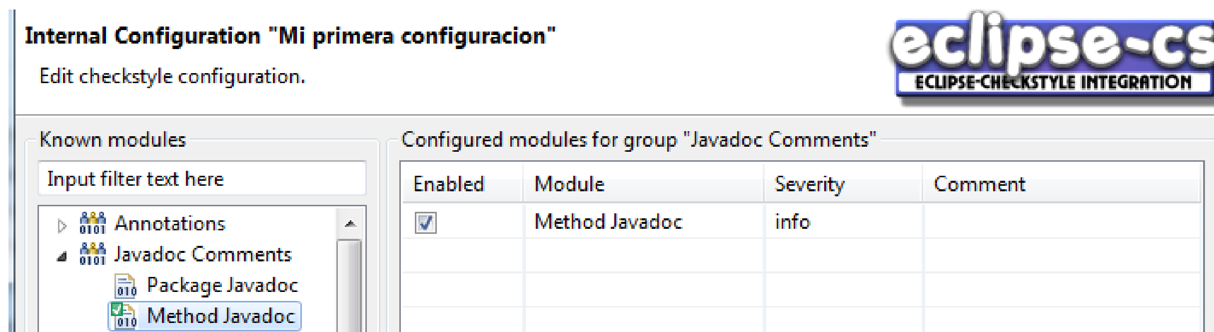


Figura 41. Nuevo módulo añadido

En estos momentos, la configuración que se está creando contiene el *Method Javado*c. Con la casilla *Enabled* seleccionada se permite configurar fácilmente el nivel de gravedad del módulo.

Se da a *Aceptar* en todas las ventanas para guardar los cambios realizados.

5.- Para hacer uso de esta configuración, hay que ir a propiedades del proyecto y seleccionar esta configuración.

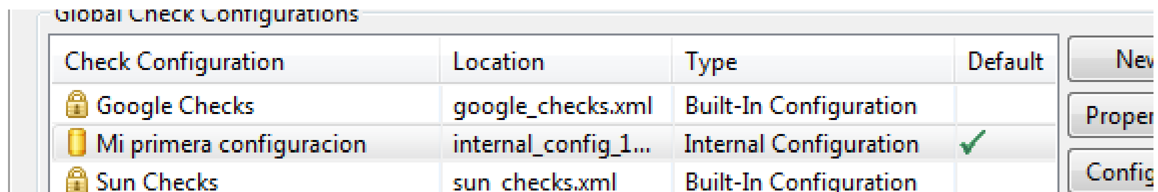


Figura 42. Configuración nueva añadida

Esto se puede hacer con cada módulo.

Hasta ahora se ha cubierto los aspectos básicos de la configuración de Checkstyle Plug-in. En los apartados siguientes se profundiza en aspecto avanzados y se crea una extensa configuración.

M.1.4.- CONFIGURACIÓN AVANZADA

M.1.4.1.- USO DE FILTROS EN PROYECTOS

Este Plug-in da la posibilidad de usar algunos filtros para excluir ciertos archivos. Como ya se ha visto se puede configurar conjunto de archivos y ahora se ve también que se puede aplicar una serie de filtros. Esto parece ser redundante.

Configurando un conjunto de archivos se permite incluir excluir un fichero basado en su nombre. Hay veces que se quiere incluir o excluir dependiendo de otras propiedades como por ejemplo, excluir los archivos que están protegidos contra la escritura.

Los filtros se encuentran en la sección inferior de las propiedades de Checkstyle dentro del proyecto.

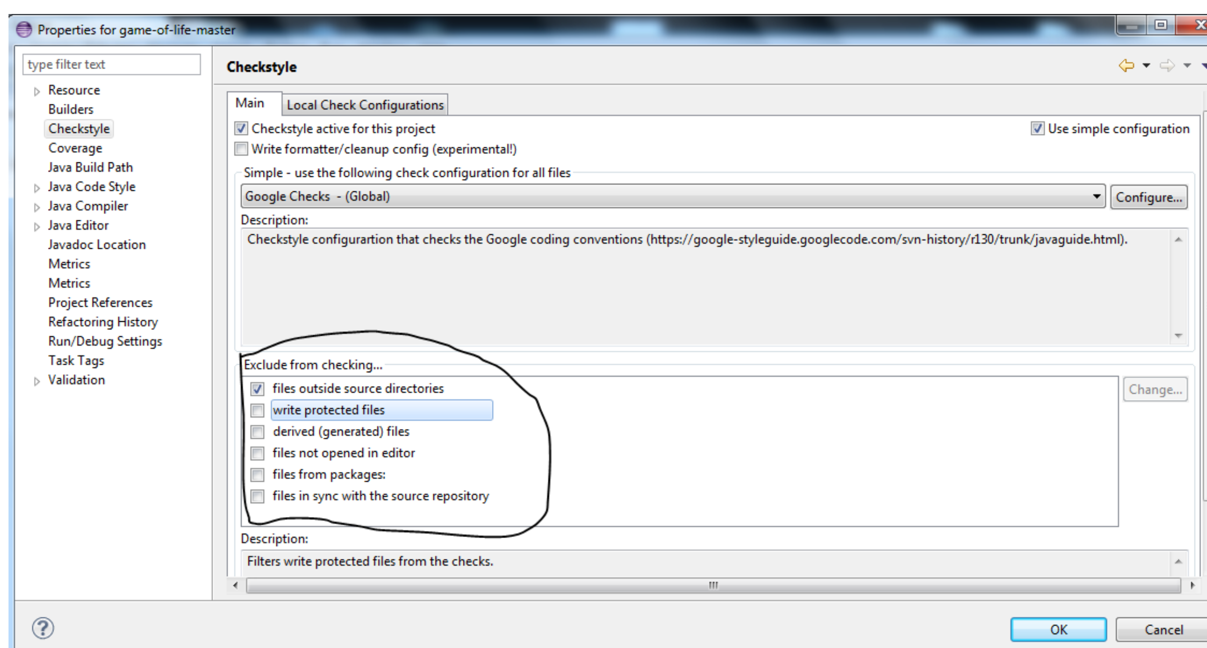


Figura 43. Filtros

Para habilitar un filtro basta sólo con seleccionar la casilla de verificación que le corresponda. Algunos filtros son configurables y al seleccionarse, el botón de *Change*… se habilitará. Si se pulsa en él de abrirá un editor para ese filtro específico.

Los filtros que se incluyen en Checkstyle se muestran en la siguiente tabla:

Nombre del Filtro	Descripción
Files outside source directories	Este filtro está activado por defecto. Algunos proyectos tienen archivos fuente java fuera de las carpetas de origen. Se pueden expluir.
Write protected files	Utilizar este filtro si solo se quiere chequear los archivos grabables. Es útil cuando se usa un sistema de control de versions chek-in/chek-out de semántica que usa archivos protegidos contra la escritura para permitir/denegar modificaciones. Así que se puede restringir el Plug-in para archivos que actualmente están trabajanado.
Derived (generated) files	Se excluye los recursos del proyecto que tienen el indicador derivado como suele ser el caso de las fuentes generadas.

Files not opened in editor	Puede utilizar este filtro para ver sólo los archivos que abrió en un editor de Eclipse. Esto es particularmente útil para proyectos grandes, ya que la comprobación de todos los archivos podría dar lugar a tiempos de construcción inaceptables o consumo de recursos. Si este filtro se activa un archivo de proyecto que se abre en el editor se comprueba automáticamente al abrir, señalando los problemas en este archivo. Del mismo modo, cuando se cierra el archivo de los marcadores de problemas assiciated se borran de la vista Problemas.
Filer fom packages	Se trata de un filtro configurable, lo que le permite especificar los paquetes que desea excluir de la comprobación.
Files in sync with the source repository	Si el proyecto se basa en un repositorio de código fuente (Git, SVN, CVS) es posible que desee utilizar este filtro. Con esto se puede limitar los controles a los archivos que realmente cambiaron. Todos los demás archivos que están en sintonía con el repositorio (por lo tanto, sin cambios) se excluyen de los chequeos.

Tabla 4. Filtros de Checkstyle

Los filtros que se incluyen en Checkstyle se muestran en la siguiente tabla:

M.1.4.2.- TIPOS DE CONFIGURACIÓN

Al crear archivos de configuración se vió en apartados anteriores que hay diferentes tipos de configuración. El Eclipse Checkstyle usa archivos de configuración normal estándar. En este apartado se da una explicación de cada tipo de configuración que existe.

- **Configuración Built-in**. Son las configuraciones de reglas que vienen incluidas en el plug-in. Éstas no se pueden ni editar ni eliminar. Pero se permite crear una configuración propia a partir de una configuración built-in (o de cualquier otro) a partir de la copia de esa configuración.

- **Configuración interna**. Se almacena dentro de los metadatos de Eclipse. La configuración interna se describió en el apartado M.1.3. Es necesario dar un nombre a la configuración. Este archivo también puede ser importado a la configuración interna pulsando sobre el botón *Import…* Hay que tener en cuenta que esto sobrescribirá la configuración existente para esa configuración. Si se necesita el archivo de configuración fuera del plug-in se debe utilizar la funcionalidad de exportación.
  Pueden ser editadas y eliminadas (y que también se elimine el archivo de configuración de Checkstyle).

- **Configuración externa**. Si se tiene el archivo de configuración fuera del área de trabajo de Eclipse se usa esta configuración. Esta configuración externa lo que hace es apuntar a este archivo y lo usa desde su ubicación. Hay que proporcionar un

nombre único para la configuración junto con la ruta absoluta del archivo de configuración. Las rutas relativas no son compatibles.

Configuraciones externas pueden ser editadas por defecto si el archivo de configuración se puede escribir. Si se tiene un archivo de configuración cuidadosamente elaborado (con bonitos comentarios y todo eso) - que no se quiere "destruir" accidentalmente usando el editor de configuración - se puede utilizar la opción de archivo de configuración *Protect Checkstyle*. Cuando se activa el editor de configuración no tocará el archivo aunque esté escritura habilitada. Si elimina una configuración externa del plug-in el archivo de configuración externa no se eliminarán.

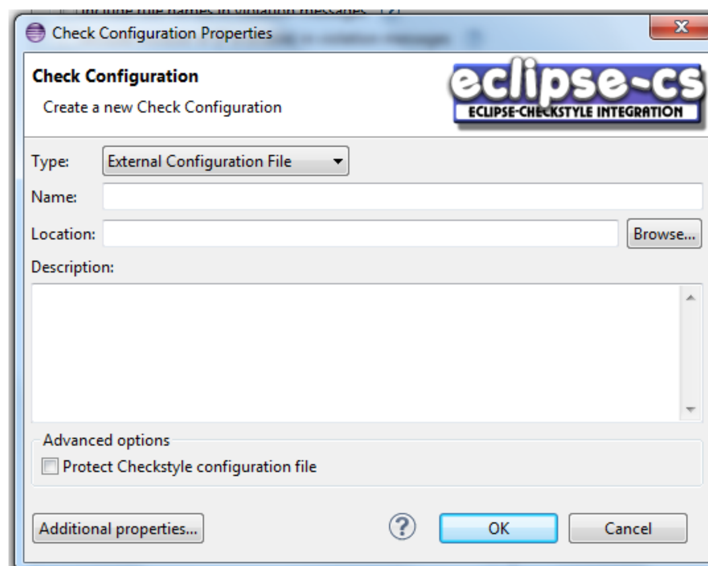


Figura 44. Configuración externa

- **Configuración Remota**. Se utilizan si se quiere publicar su configuración Checkstyle desde un servidor web. Al igual que en las demás configuraciones hay que poner un nombre y la URL del archivo de configuración. Si el servidor requiere autentificación básica HTTP, un cuadro de diálogo emergente solicitará unos credenciales. Existe una configuración avanzada: archivo de configuración de caché que consiste en que si está en un lugar en el que no puede acceder al servidor, con esta opción almacenamos en caché el archivo parar que esté accesible. Esta configuración remota no se puede modificar pero sí eliminar del plug-in.

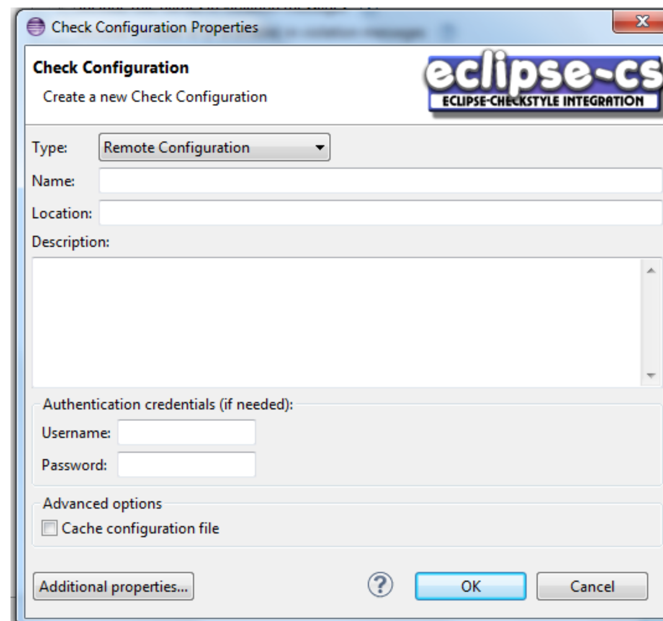


Figura 45. Configuración remota

- **Configuración relativa del Proyecto**. Si se tiene un archivo de configuración Checkstye dentro de un proyecto en su área de trabajo lo más fácil es usar esta configuración. Como siempre hay que proporcionar nombre junto con la ruta relativa del proyecto. Esta configuración se puede editar con el editor de configuración cuando el archivo de configuración se puede escribir. Al igual que con el tipo de configuración externa del archivo de configuración puede ser protegida de la reescritura accidentalmente con el editor de configuración. Cuando se elimina una configuración de este tipo, Checkstyle permanece intacta.

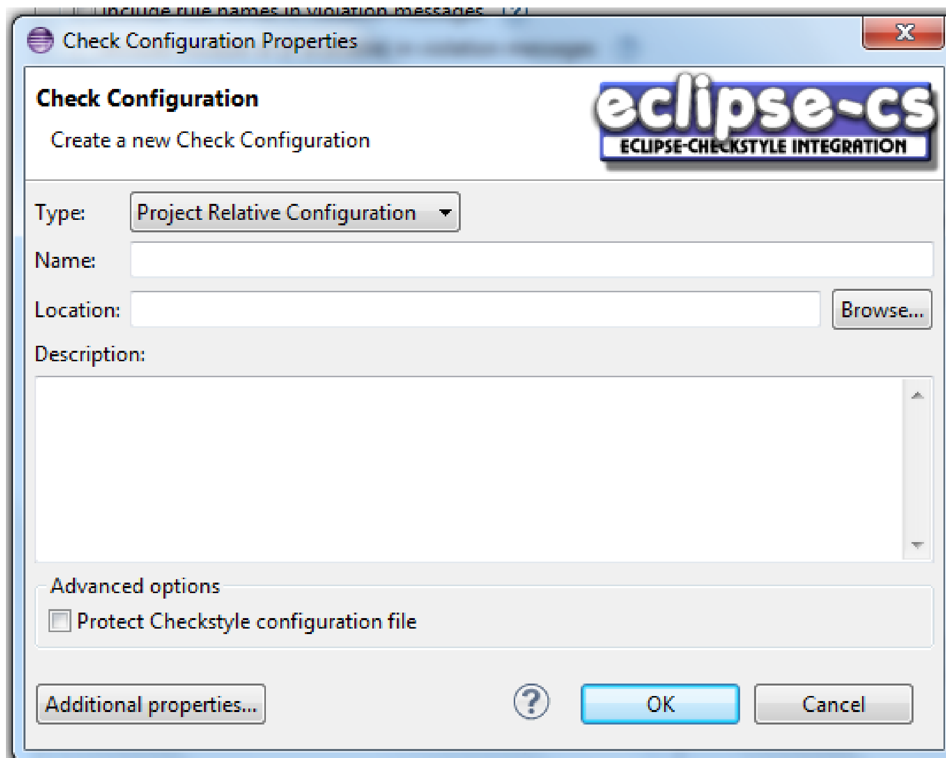


Figura 46. Configuración relativa del proyecto

M.1.4.3.- CONFIGURACIÓN DE UN CONJUNTO DE ARCHIVOS

En la sección de instalación inicial configuramos nuestro proyecto utilizando el enfoque de configuración simple.

Con la configuración sencilla todos los archivos en su proyecto se comprueban por la misma configuración de verificación.

Figura 47. Configuración de un conjunto de archivos (I)

Pero puede que en algunos casos se necesite más control:

- Sólo se quiere chequear o excluir ciertos archivos basados en un nombre de archivo.
- Sólo se quiere chequear partes de tu proyecto con una configuración concreta y chequear otra parte del proyecto con otra configuración.

Para dar un control fino sobre que archivos se chequean con un archivo de configuración y que archivos se chequean con otro archivo de configuración, el plug-in da la opción de configurar este conjunto de archivos. Un conjunto de archivos es una colección de archivos dentro de un mismo proyecto que necesita ser chequeado con la misma configuración. Se puede desactivar la casilla de verificación *Use simple configuration*.

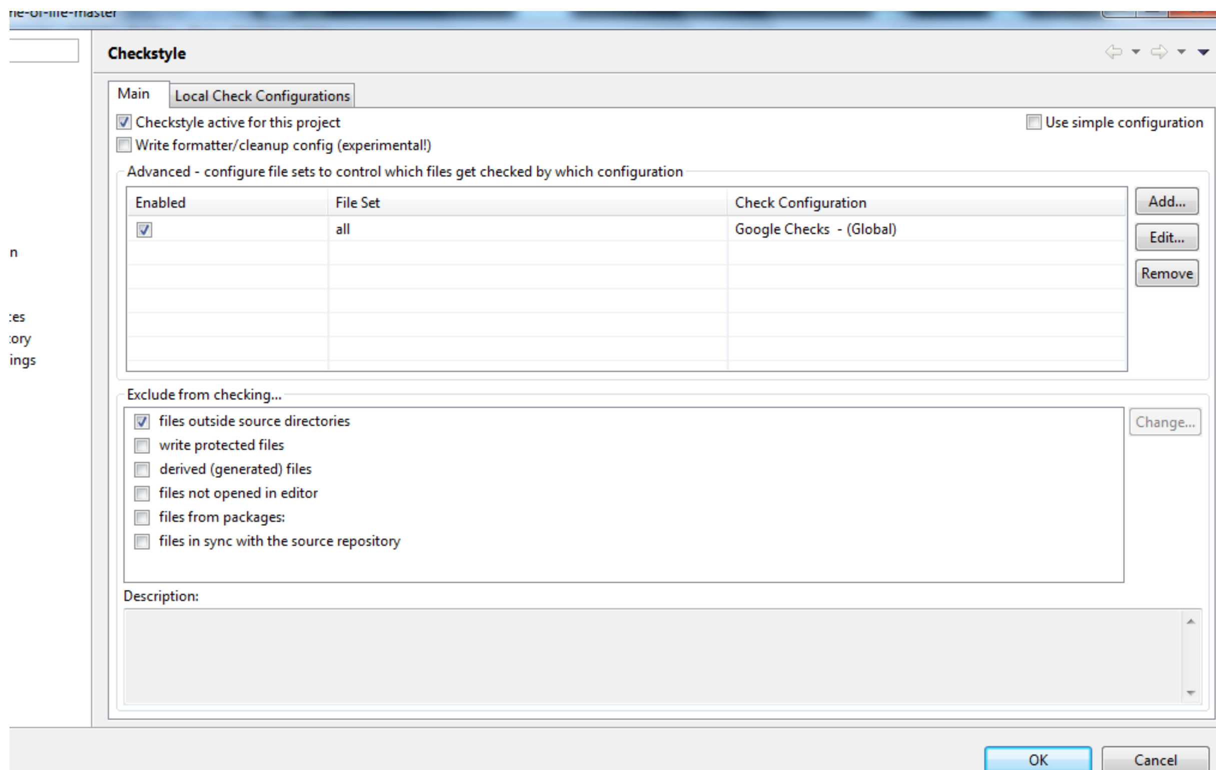


Figura 48. Configuración de un conjunto de archivos (II)

Ahora una tabla aparece conteniendo e conjunto de archivos de configuración para este proyecto. Por defecto existe uno llamado *all*.

Para crear un nuevo conjunto de archivos se pulsa el botón *Add*…. Para editar un axistente conjunto de archivos se pulsa el botón *Edit*… Una vez pulsado uno de esos botones sale una ventana como la siguiente imagen:

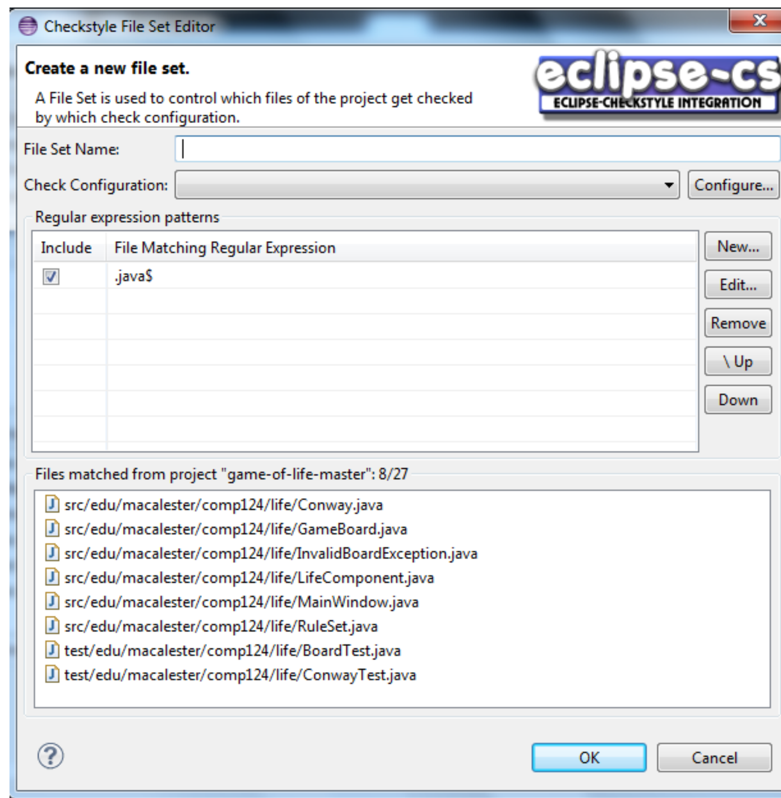


Figura 49. Creación de un nuevo conjunto de archivos

Hay que poner un nombre y seleccionar que configuración quiere (Google, Sun u otra que haya propia).

A continuación, introduzca una serie de expresiones regulares para seleccionar los archivos que desea incluir en el conjunto de archivos. Cada expresión regular puede ser utilizado para incluir o excluir archivos (las expresiones regulares por defecto coincide con todos los archivos que terminan en ".java").

La expresión regular se compara con el nombre de archivo completo dentro del proyecto. Es decir, toda la ruta de acceso al archivo empezando por el nombre de la carpeta de nivel superior dentro del proyecto y progresar a través de cada carpeta hasta el archivo, con '/' se utiliza como separador de ruta.

Para ajustar el orden de las expresiones regulares, utilice los botones *Up* y *Down* para mover la expresión seleccionada arriba o abajo en la lista. En la parte de abajo aparecen los archivos que están incluidos en el conjunto.

Cada proyecto en el espacio de trabajo de Eclipse puede tener cualquier número de conjuntos de archivos definido, cada uno con un conjunto diferente de archivos y cada uno con un cheque de configuración diferente.

También puede desactivar la comprobación de un Conjunto de archivos mediante la casilla de verificación *Enable* en la ventana de propiedades.

M.1.4.4.- CONFIGURACIÓN DE PREFERENCIAS AVANZADAS

En *Window > Preference…* Aparecen una serie de ítems que se pueden configurar. En este apartado se explica para que sirve cada uno de ellos.

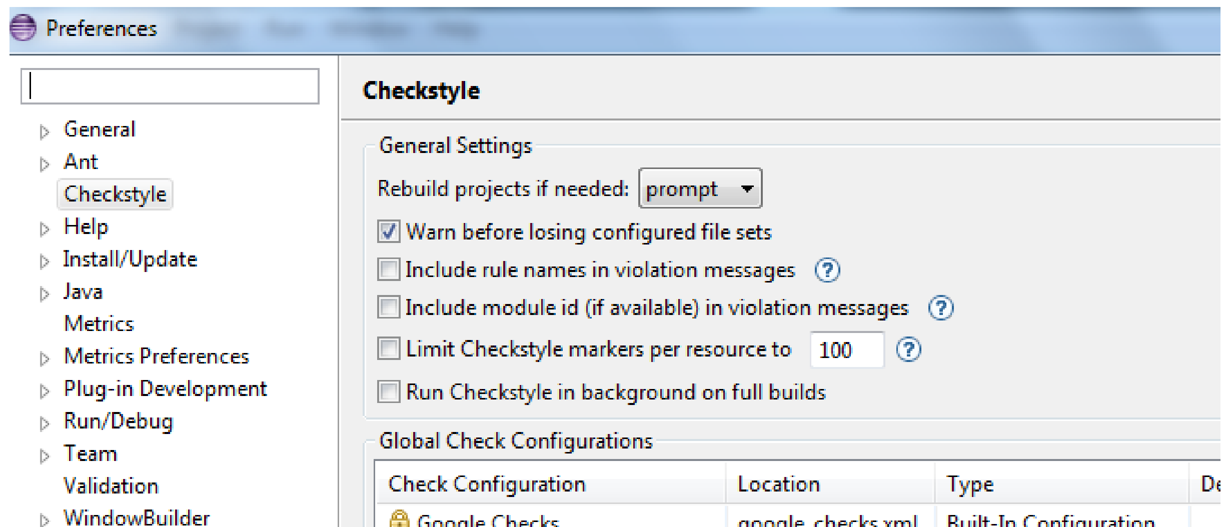


Figura 50. Configuración de preferencias avanzadas

- **Rebuild projects if needed**. Determina el comportamiendo del plug-in cuando una configuración es cambiada mediante el editor de configuración. Esta opción se puede configurar para reconstruir automáticamente los proyectos afectados, nunca para reconstruir estos proyectos.

- **Warn before losing configured file sets**. Establecer si el plugin debe advertir que se está a punto de perder los conjuntos de archivos configurados en las propiedades del proyecto.

- **Include rule names in violation messages**. Controla si el nombre del módulo Checkstyle informando de un problema se antepone al mensaje producido. Si se activa se puede utilizar para ordenar los problemas a través de la vista Problemas alfabéticamente y agrupar las advertencias juntas.

- **Include module id (if available) in violation messages**. En un archivo de configuración Checkstyle cada módulo se le puede asignar un identificador único. Esto se puede utilizar para distinguir varias instancias del mismo módulo Checkstyle utilizando diferentes configuraciones. Esta preferencia de configuración de directiva controla si este módulo id se antepone a los mensajes Checkstyle producidos.

- **Limit Checkstyle markers per resource to**. Utilice esta opción para limitar la cantidad máxima de los marcadores reportados para un solo archivo. Una posible razón para usar esto es para evitar problemas de rendimiento en caso de que se reporta una gran cantidad de problemas para los archivos individuales

- **Run Checkstyle in background in full builds**. Por defecto la ejecución Checkstyle se ejecuta dentro de la infraestructura de construcción Eclipses, lo que significa una generación de proyecto completo solamente terminará una vez que el análisis Checkstyle ha concluido. Para proyectos muy grandes esto puede ser poco práctico, en cuyo caso se puede utilizar este ajuste para desacoplar la ejecución Checkstyle de la acumulación de Eclipse. En este caso Checkstyle todavía se activará al proyecto de construcción pero no bloqueará la construcción en sí.

- **Botón de recarga (arriba a la derecha)**. Por motivos de rendimiento el plugin almacena en caché internamente configuraciones Checkstyle (que no es el archivo de configuración, pero la estructura resultante en memoria objeto de módulos Checkstyle). Modificaciones en el archivo de configuración Checkstyle sí son descubiertos por el plugin de forma automática y los cachés se actualizarán correctamente. Sin embargo, un archivo de configuración Checkstyle también puede hacer referencia a los archivos de configuración suplementarios (definiciones de cabecera, archivos de supresión etc.). En este punto, el plugin no es lo suficientemente inteligente como para detectar modificaciones para ver estos archivos adicionales. Si está modificando estos archivos puede que tenga que borrar manualmente cachés internas del plugin usando este pequeño botón.

Figura 51. Botón de recarga

M.2.- JACOCO

M.2.1.- INTRODUCCIÓN

JaCoCo se puede integrar dentro del IDE Eclipse para realizar un análisis de cobertura de código cuando se ejecutan una serie de casos de pruebas.

- Ciclo de pruebas y desarrollo rápido: Se lanza desde el entorno de trabajo.
- Buen análisis de cobertura en el que los resultados son inmediatamente resumidos y resaltados.
- No requiere la modificación de los proyectos u otra configuración.

Características.

- Lanzamiento: se lanza igual que las existentes *Run* y *Debug*. Se puede activar desde el menú o desde el icono siguiente que está en la barra de herramientas:

Figura 52. JaCoCo. Botón lanzamiento.

- Análisis: la información sobre la cobertura se muestra en el entorno de trabajo de Eclipse.
  - Resumen de cobertura de todo el proyecto.
  - Resaltado del código en los editores mediante colores personalizables para indicar las líneas y ramificaciones que se han cubierto completamente, parcialmente o nada.

  Además, hay un análisis de apoyo:
  - Selecciona las instrucciones, las ramas, líneas, métodos, tipos complejidad ciclómatica que debe resumirse.
  - Sesiones de cobertura múltiples.
  - Si diferentes test deben ser considerados para el análisis de cobertura, pueden combinarse.

- Importar/Exportar: permite importar datos de ejecución .exec y exportar informes en formato HTML, XML o CSV.

M.2.2.- INSTALACIÓN

Como prerrequisito para poder instalarlo, se necesita Eclipse 3.5 o superior y Java 1.5 o superior. En cuanto a la instalación, hay tres opciones para instalar JaCoCo en Eclipse.

Opción 1: Instalar desde MarketPlace de Eclipse.

1.- En Eclipse: en el menú seleccionar *Help -> Eclipse Marketplace*.

2.- Buscar “EclEmma”.

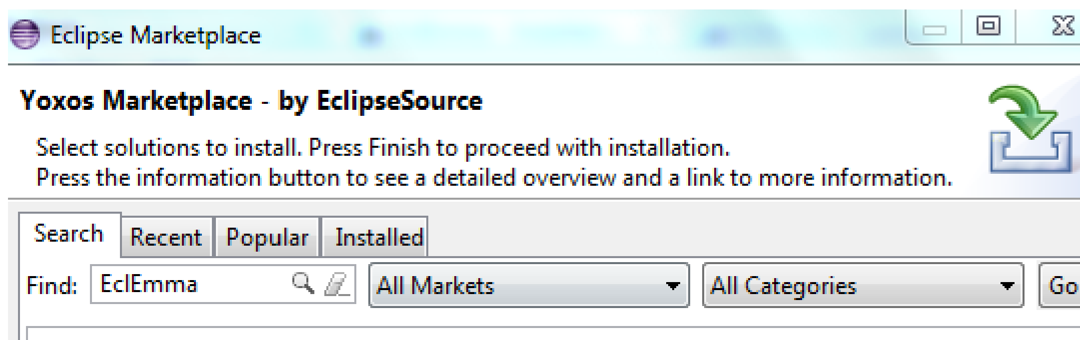


Figura 53. JaCoCo. Instalación por Marketplace(I)

3.- Seleccionar “Install” para “EclEmma Jaca Code Coverage”.

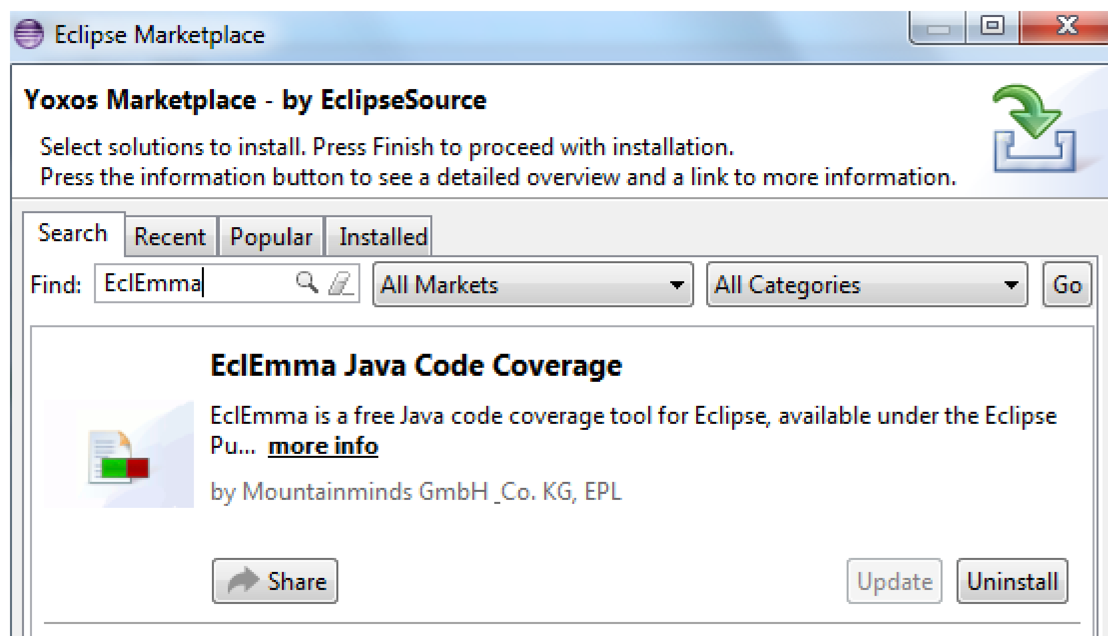


Figura 54. JaCoCo. Instalación por Marketplace(II)

4.- Seguir los pasos para la instalación.

Opción 2: Instalación desde el sitio de actualizaciones de Eclipse.

1.- En Eclipse: en el menú seleccionar *Help -> Install new software*.

2.- Clic en *Add* y añade lo siguiente:

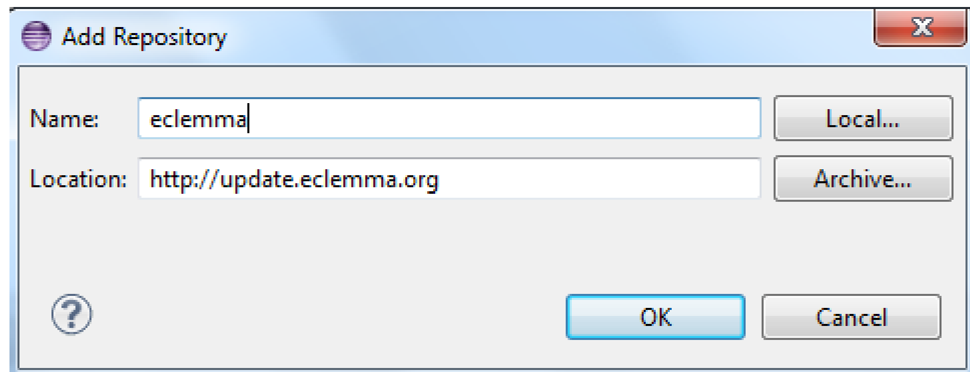


Figura 55. JaCoCo. Instalación desde el sitio de actualizaciones (I)

Clic en "OK" y en el cuadro de instalación seleccionas eclEmma- http://update.eclemma.org/ en el campo de texto "*Work with*".

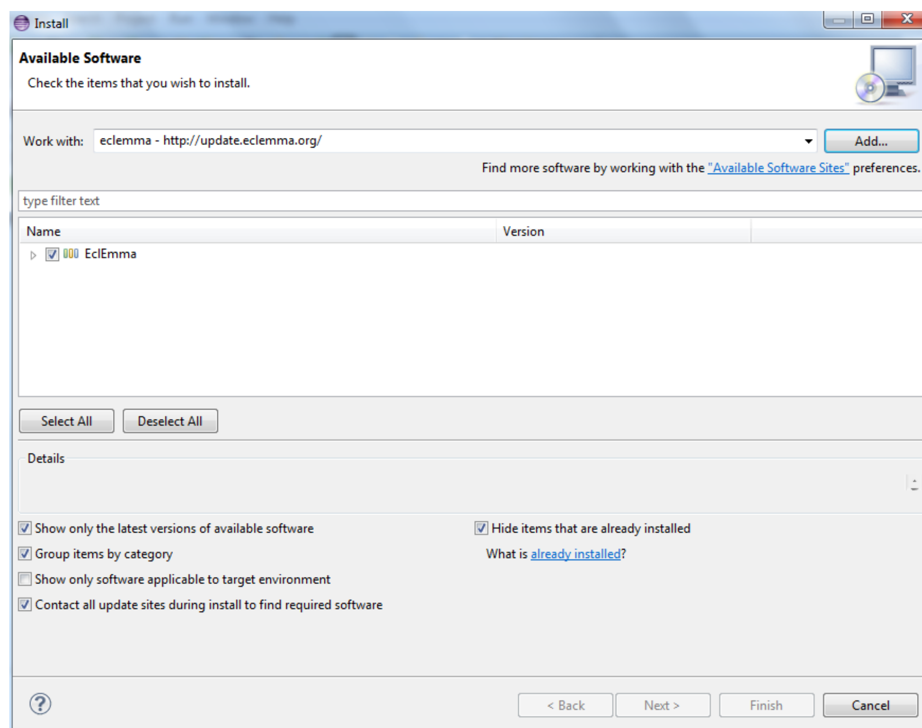


Figura 56. JaCoCo. Instalación desde el sitio de actualizaciones (II)

3.- Seleccionar la última versión y clic en "*Next*".

4.- Seguir los pasos de la instalación.

Opción 3: Descargando el manual e instalación

1.- Descargar del siguiente enlace la última versión: http://www.eclemma.org/download.html

The official releases builds are available for download below. Alternatively EclEmma can be dire

Download	Source	Release	Size	MD5 Checksum
eclemma-2.3.2.zip	2.3.2	2014/09/14	1.2 MB	c1a80f7dc8a30474fe1a6143cbe295a9

Figura 57. JaCoCo. Instalación descargando manual (I)

2.- Descomprimir el archivo dentro de la carpeta "*dropins*" de la carpeta de instalación de Eclipse y reinicia Eclipse.

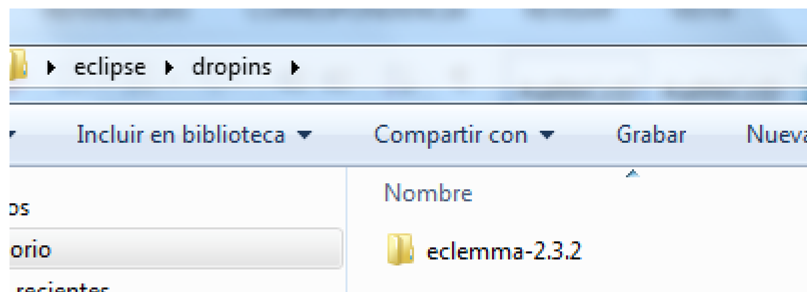


Figura 58. JaCoCo. Instalación descargando manual (II)

Para **verificar** que se ha instalado correctamente, tenemos que ver el símbolo en la barra de herramientas.

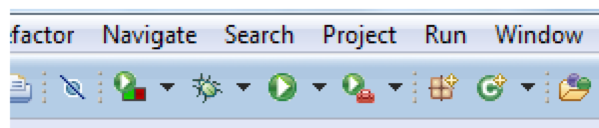


Figura 59. JaCoCo. Verificación de instalación

M.2.3.- GUÍA DEL USUARIO

El análisis de cobertura implica dos pasos:

- Ejecutar el programa.

- Analizar los datos de cobertura.

Se suelen automatizar con pruebas unitarias JUnit.

Para ejecutar un análisis de cobertura sólo hay que presionar su botón correspondiente. El resultado se verá automáticamente en la vista de cobertura y resaltado con colores en el editor de código Java.

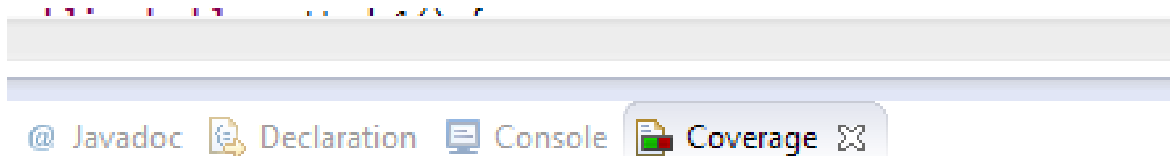


Figura 60. JaCoCo. Guía del usuario (I)

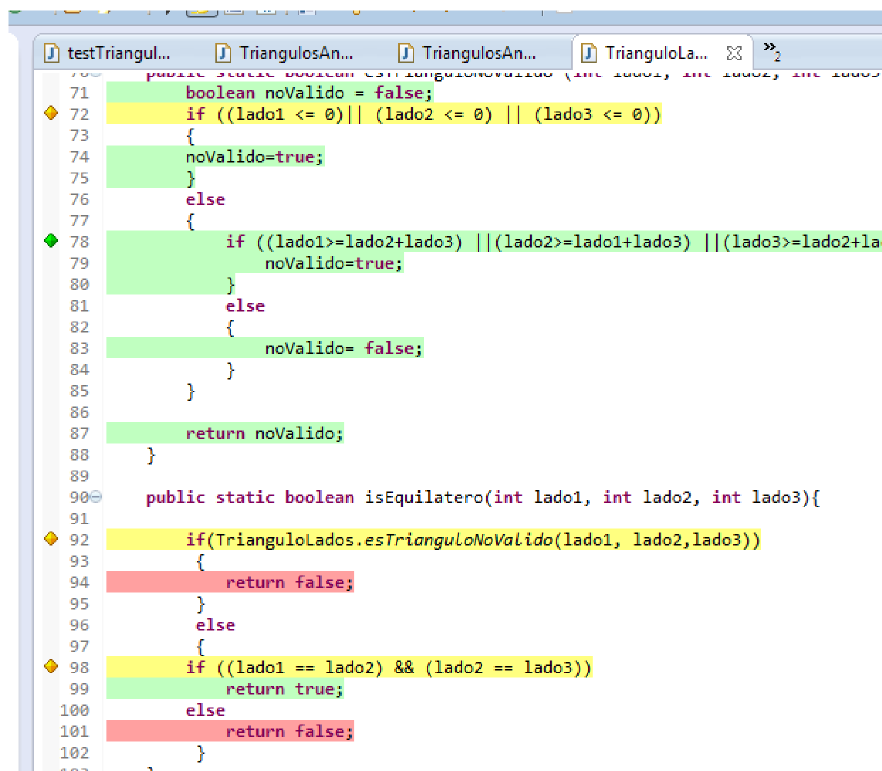


Figura 61. JaCoCo. Guía del usuario (II)

A) Lanzamiento.

El lanzamiento está disponible desde el icono de la barra de herramientas, desde el menú *Run -> Coverage as* o haciendo clic con el botón derecho en el nombre del proyecto. Si se requiere algunos ajustes se pueden modificar *“Coverage As”->“Coverage Configurations”*

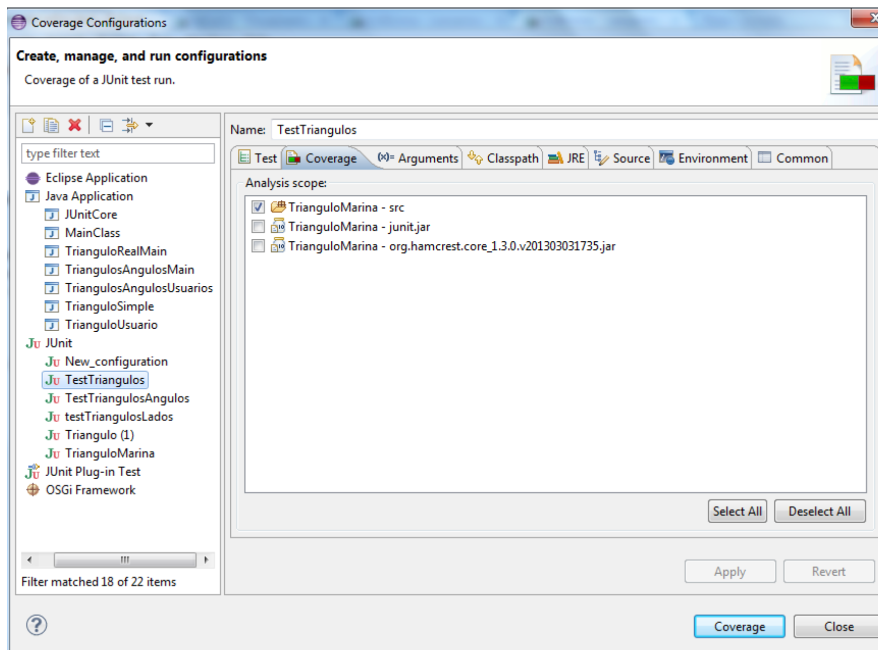


Figura 62. JaCoCo. Guía del Usuario (Lanzamiento)

B) Vista de la Cobertura (Cobertura View).

Cuando se ejecuta el análisis de cobertura, la vista aparece automáticamente. Si no apareciera, podemos hacerla aparecer en *Window -> Show View -> Other*

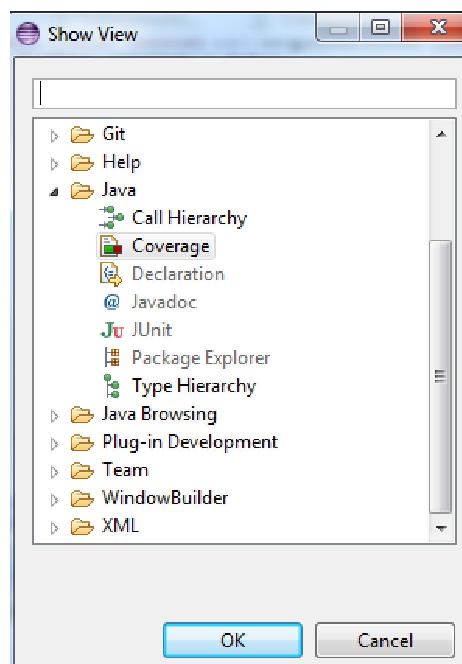


Figura 63. JaCoCo. Guía del Usuario (Selección de Vista de cobertura)

La vista de la cobertura tiene la siguiente apariencia:

Triangulo (1) (12-mar-2015 19:02:01)

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
test	100,0 %	453	0	453
testTriangulosLados.java	100,0 %	453	0	453
testTriangulosLados	100,0 %	453	0	453
testID1()	100,0 %	25	0	25
testID10()	100,0 %	25	0	25
testID11()	100,0 %	25	0	25
testID12()	100,0 %	25	0	25
testID13()	100,0 %	25	0	25
testID14()	100,0 %	25	0	25
testID15()	100,0 %	25	0	25

Figura 64. JaCoCo. Guía del Usuario (Vista de cobertura)

En la vista de la cobertura podemos ver que se muestran todos los elementos de Java analizados dentro de la jerarquía del proyecto. Hay columnas individuales:

- Tasa de cobertura.
- Instrucciones/Ramas/Líneas/Métodos/Tipos/Complejidad Ciclomática cubiertas.
- Instrucciones/Ramas/Líneas/Métodos/Tipos/Complejidad Ciclomática no cubiertas.
- Total.

Haciendo clic en el encabezado de la columna, podemos ordenar los elementos en orden ascendente o descendente. Además, haciendo doble clic en el elemento, nos lleva a la parte del editor de código donde se encuentra ese elemento resaltado.

<u>Barra de herramienta y menú desplegable.</u>

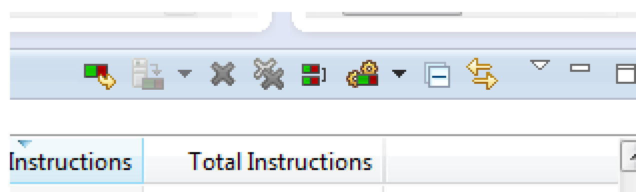


Figura 65. JaCoCo. Guía del Usuario (Menú desplegable)

La barra de herramientas de vista de cobertura ofrece las siguientes acciones:

Figura 66. JaCoCo. Guía del Usuario (Lanzamiento de la última sesión de cobertura)

Figura 67. JaCoCo. Guía del Usuario (Volcado de ejecución de los datos)

Volcado de ejecución de los datos a partir de un proceso en ejecución y crear una nueva sesión de los datos. Esto sólo se activa cuando al menos un proceso se ejecuta en modo cobertura.

Figura 68. JaCoCo. Guía del Usuario (Eliminar la sesión activa)

Figura 69. JaCoCo. Guía del Usuario (Eliminar todas las sesiones de cobertura)

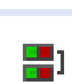


Figura 70. JaCoCo. Guía del Usuario (Combinar sesiones: varias sesiones en una sola.)

Figura 71. JaCoCo. Guía del Usuario (Activar la sesión.)

Figura 72. JaCoCo. Guía del Usuario (Colapsar todo)

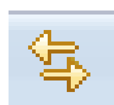


Figura 73. JaCoCo. Guía del Usuario (Enlazar con la selección actual)

Enlazar con la selección actual. Si este conmutador se comprueba la vista cobertura revela automáticamente el elemento Java seleccionado actualmente en otros puntos de vista o editores.

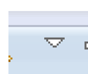


Figura 74. JaCoCo. Guía del Usuario (Menú desplegable)

Algunas de las acciones se desactivan si no hay sesión o sólo una única sesión. En el menú desplegable de la vista de cobertura hay más ajustes:

- Mostrar Elementos: selecciona los elementos Java que se mostrarán en el árbol de cobertura: Proyectos, raíces de fragmentos de paquetes, fragmento de paquetes o tipos).
- Modo de Contador: se puede seleccionar los diferentes modos: instrucciones de código de bytes, ramas, líneas, métodos, tipos y complejidad ciclomática.
- Ocultar los elementos no utilizados. Oculta los elementos en la vista de cobertura que no han sido ejecutados durante la sesión de cobertura. Si se está trabajando en una unidad en particular se puede filtrar todas las clases que no se han cargado durante la realización de la prueba.

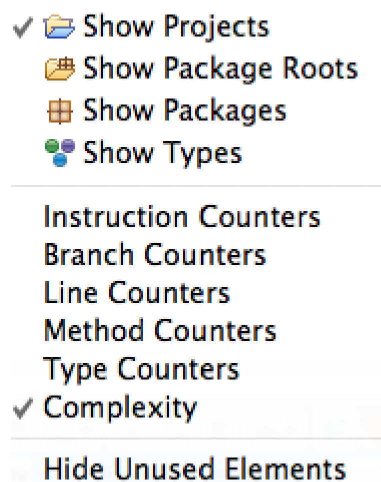


Figura 75. JaCoCo. Guía del Usuario (Opciones del menú desplegable)

C) Anotaciones en el Código Fuente.

Cuando se hace el análisis de cobertura se muestra en el editor de código la cobertura de línea y de ramificaciones por medio de líneas y diamantes de colores respectivamente como se ha descrito en apartados anteriores.

```
70  public static boolean esTrianguloNoValido (int lado1, int lado2, int lado3){
71      boolean noValido = false;
72      if ((lado1 <= 0)|| (lado2 <= 0) || (lado3 <= 0))
73      {
74      noValido=true;
75      }
76      else
77      {
78          if ((lado1>=lado2+lado3) ||(lado2>=lado1+lado3) ||(lado3>=lado2+lado1)){
79              noValido=true;
80          }
81          else
82          {
83              noValido= false;
84          }
```

Figura 76. JaCoCo. Guía del Usuario (Anotaciones en el código fuente)

Los colores se pueden personalizar en *Preferencias*. Por defecto, los colores para las líneas son:

- Color rojo: ninguna instrucción de la línea ha sido ejecutada.
- Color amarillo: sólo una parte de la línea ha sido ejecutada.
- Color verde: todas las instrucciones han sido ejecutadas.

Y, teniendo una semántica similar a los colores de las líneas, los colores para los diamantes de las ramificaciones:

- Color rojo: no ha sido ejecutada.
- Color amarillo: sólo una parte ha sido ejecutada.
- Color verde: todas han sido ejecutadas.

Cuando se inicia la edición del código o se borra la sesión de cobertura, los colores en el código desaparecen.

Para la edición de los colores hay que ir a: Windows -> Preferences, en la sección General -> *Appearance* -> *Editors* -> *Text Editors* -> *Annotations*. Las entradas son: Full coverage, No coverage, Partial coverage.

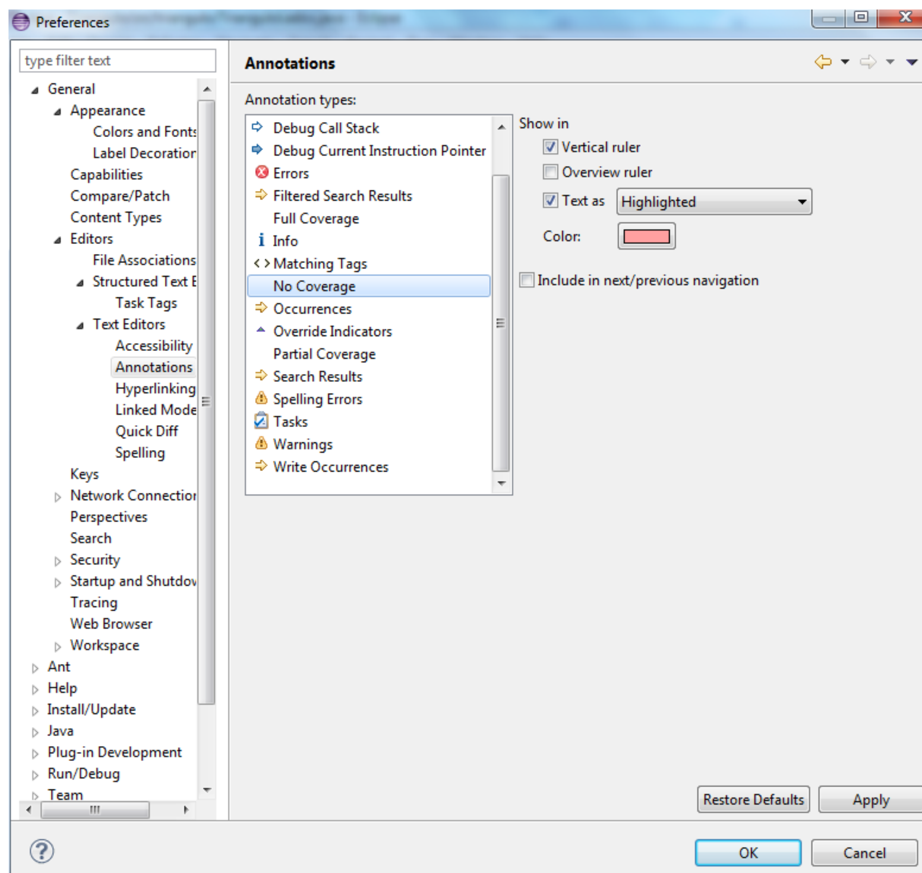


Figura 77. JaCoCo. Guía del Usuario (Anotaciones en el código fuente - Edición de colores)

D) Propiedades de Cobertura.

Para cada elemento Java existe un resumen de sus propiedades de cobertura con todos sus contadores. Pueden verse en Propiedades de cada elemento:

Properties for TrianguloLados.java

Resource
Coverage
Run/Debug Settings

Coverage

Session: testTriangulosLados (12-mar-2015 11:45:19)

Counter	Coverage	Covered	Missed	Total
Instructions	89,7 %	139	16	155
Branches	67,5 %	27	13	40
Lines	84,1 %	37	7	44
Methods	100,0 %	12	0	12
Types	100,0 %	1	0	1
Complexity	59,4 %	19	13	32

OK Cancel

Figura 78. JaCoCo. Guía del Usuario (Propiedades de cobertura)

E) Decoración de Cobertura.

Para que en nuestro explorador de paquete se muestre al lado de cada clase la tasa de cobertura, vamos a *Window -> Preference*, en la sección de *General -> Appearance -> Label Decorators, Seleccionamos Java Code Coverage* y presionamos OK.

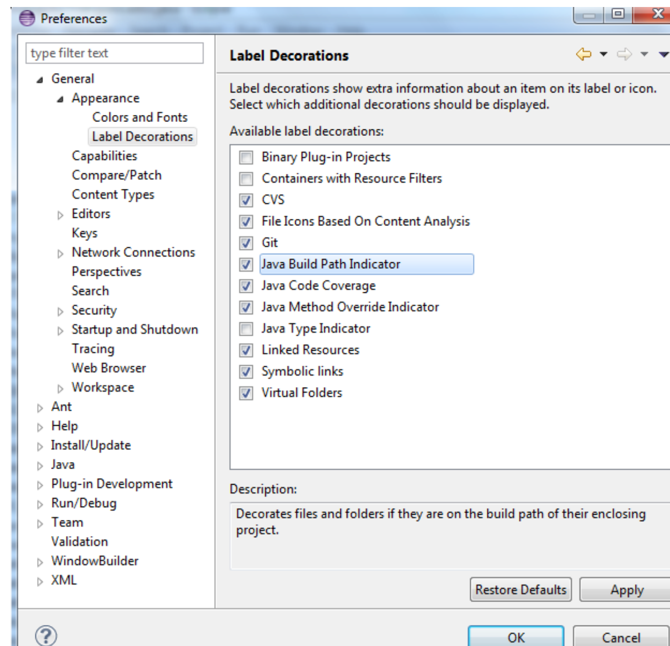


Figura 79. JaCoCo. Guía del Usuario (Decoración de la cobertura I)

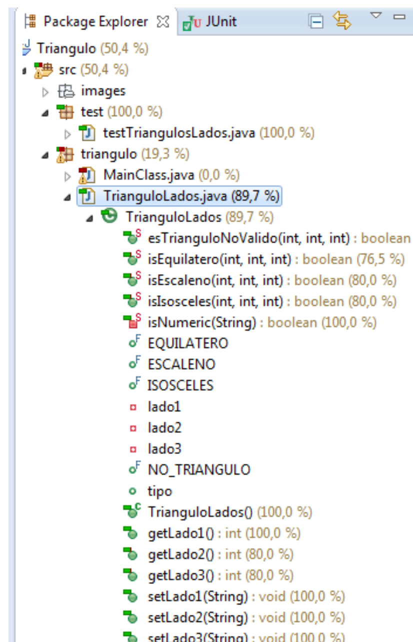


Figura 80. JaCoCo. Guía del Usuario (Decoración de la cobertura II)

F) Administración de Sesiones Cobertura

Una **sesión de cobertura** es la información de cobertura de un concreto programa en ejecución. Contiene la lista de clases de Java junto con los detalles de cobertura.

Una sesión de cobertura es automáticamente creada al final de cada lanzamiento de cobertura o cada vez que una ejecución de los datos ha sido lanzado por el usuario. Alternativamente, las sesiones pueden ser importadas externamente. Mediante la vista de cobertura se pueden eliminar sesiones.

Cuando Eclipse se cierra, todas las sesiones de cobertura se cierran.

Cuando hay múltiples sesiones de cobertura, una sesión es la activa. La sesión activa se selecciona del menú desplegable de la vista de cobertura.

Si el conjunto de pruebas consiste en múltiples lanzamientos de pruebas, ellos serán una múltiple sesión de cobertura diferente por lo que para su análisis se puede combinar esas sesiones en una simple mediante el comando se Combinación de sesiones. Este comando permite seleccionar un subconjunto de las sesiones existentes y combinándolas en una sola sesión de cobertura.

G) Importar/ Exportar Sesiones.

Importar Sesiones.

Para importarlo nos vamos a *File-> Import*...del menú. En el menú siguiente hay que especificar el origen de los datos mediante tres posibles:

- Local: localización de un archivo *.exec
- Remoto: archivo URL *.exec
- IP address y puerto de un agente JaCoCo junto a un proceso en ejecución.

Además también se puede especificar el modo de importación. Si se mantiene referencia a la fuente de datos original o si se quiere crear una copia. Con la primera opción puedes re-importar los datos cada vez que se quiera simplemente actualizando o pulsando la tecla de F5.

En el siguiente paso se puede especificar el nombre de la sesión y el alcance (carpetas a considerar y librerías).

Una advertencia es que se debe importar los que sean de la misma clase que se usan en Eclipse ya que si se creó, por ejemplo, en otro compilador, al importar no se mostrará ninguna cobertura.

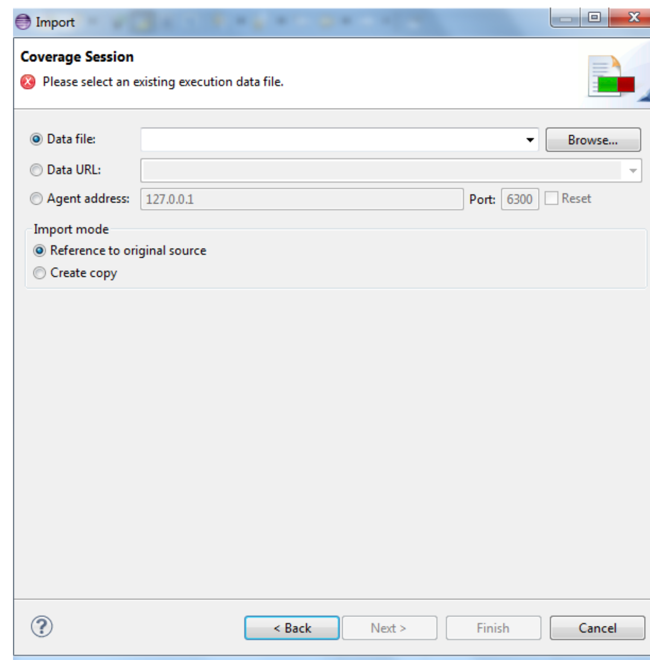


Figura 81. JaCoCo. Guía del Usuario (Importar sesiones)

<u>Exportar Sesiones.</u>

Se puede exportar las sesiones de cobertura en uno de los siguientes formatos (*File -> Export*...):

- HTML: informa detallado en conjunto de archivos HTML.
- Zipped HTML: igual que el anterior pero comprimido en un solo archivo.
- XML: un único archivo XML estructurado.
- CSV: Datos de cobertura detallados en niveles separados por comas.
- Archivos de ejecución de datos: nativo formato de ejecución de datos JaCoCo.

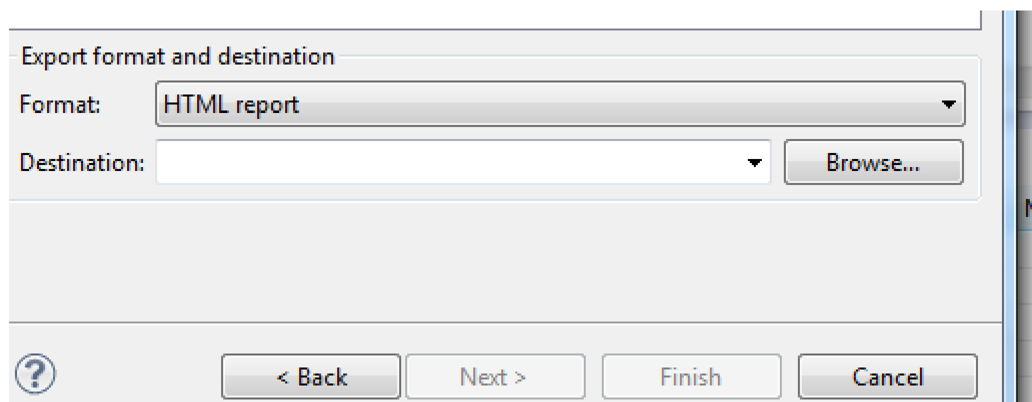


Figura 82. JaCoCo. Guía del Usuario (Exportar sesiones)

H) Uso del Teclado.

Atajos de inicio

Secuencia de teclas	Comando
Ctrl + Shift + F11	Relanzamiento último lanzamiento del programa en el modo de cobertura
Alt + Shift + E, J	Lanzamiento selección actual como aplicación Java en el modo de cobertura
Alt + Shift + E, T	Lanzamiento selección actual como prueba JUnit en el modo de cobertura
Alt + Shift + E, E	Lanzamiento selección actual como una aplicación Eclipse en el modo de cobertura
Alt + Shift + E, P	Lanzamiento selección actual como prueba plug-in JUnit en el modo de cobertura
Alt + Shift + E, R	Lanzamiento selección actual como prueba plug-in JUnit RAP en el modo de cobertura
Alt + Shift + E, L	Lanzamiento selección actual como aplicación Scala en modo de cobertura
Alt + Shift + E, S	Lanzamiento selección actual como prueba SWTBot en modo de cobertura
Alt + Shift + E, N	Lanzamiento selección actual como prueba TestNG en modo de cobertura
Alt + Shift + E, G	Lanzamiento selección actual como privado TestNG en modo de cobertura

Tabla 5. JaCoCo. Guía del Usuario (Atajos de inicio)

Accesos directos en la vista de cobertura.

Secuencia de teclas	Comando
F5	Actualizar cobertura actualizada, útil para los datos importados de los lanzamientos externos
DEL	Retire sesión cobertura actual

Tabla 6. JaCoCo. Guía del Usuario (Accesos directos en la vista de cobertura)

Personalización

Las combinaciones de teclas se pueden ajustar en *Window -> Preferences*, sección *General -> Keys*.

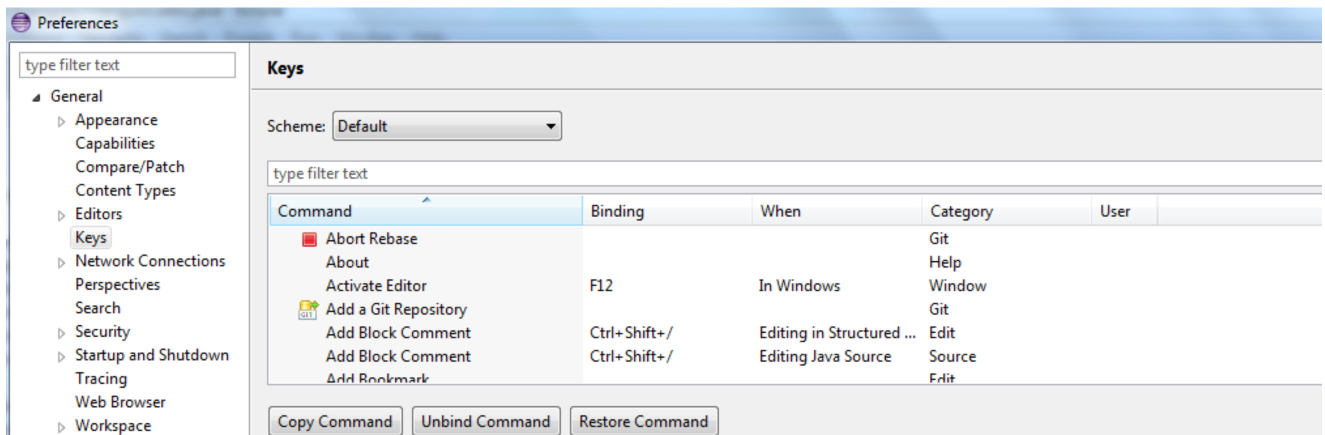


Figura 83. JaCoCo. Guía del Usuario (Personalización atajos de teclado)

I) Preferencias De Cobertura de Código.

Para escoger las preferencias: *Window -> Preference*, sección *Java -> Code Coverage*.

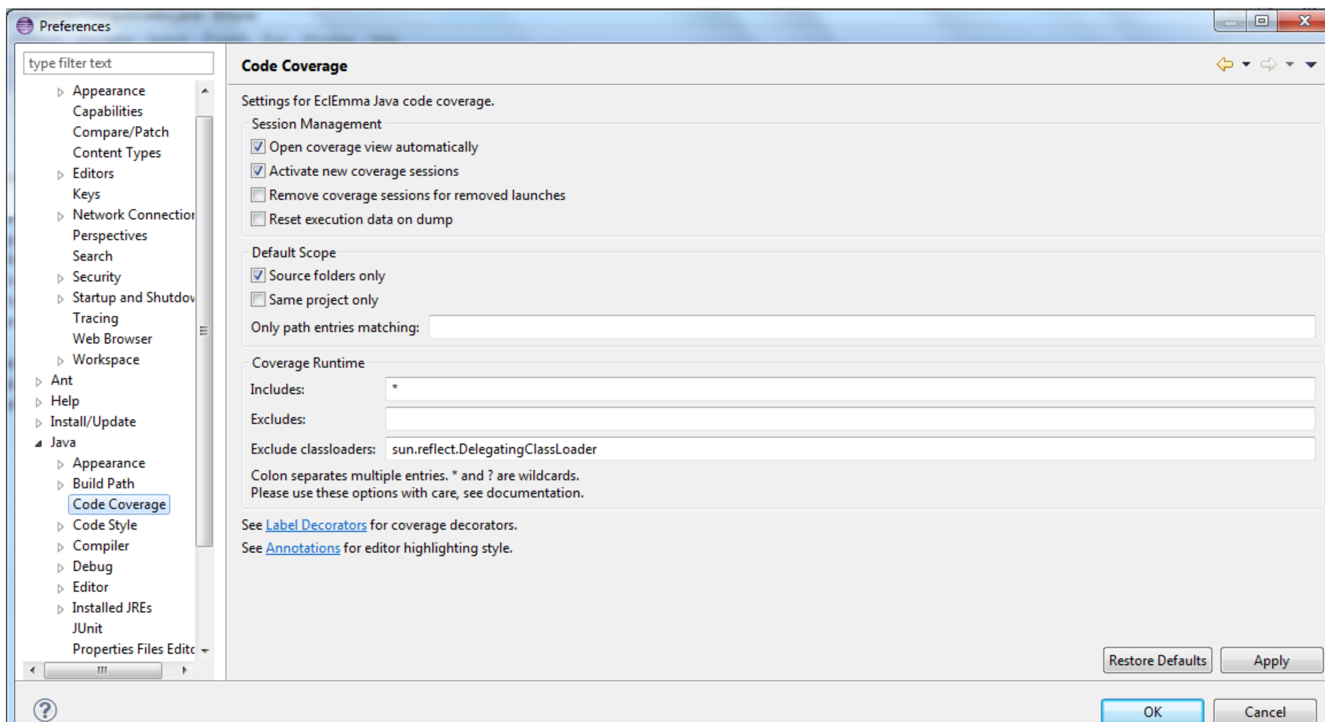


Figura 84. JaCoCo. Guía del Usuario (Preferencias de cobertura de código)

Las opciones que hay son las siguientes:

1.- Gestión de la sesión (*Session Management*):

- Abrir vista de cobertura automáticamente (*Open coverage view automatically*). Por defecto está activado y es que cuando se hace una nueva sesión de cobertura, ésta se muestra automáticamente en la ventana del entorno de trabajo.

- Activar nuevas sesiones de cobertura (*Activate new coverage sessions*). Por defecto está activado. Cuando se termina un lanzamiento de cobertura o se importa una sesión se crea una nueva sesión. Esta opción determina si el nuevo periodo de sesiones se convierte automáticamente en activo, es decir, sus datos se muestra en la vista de cobertura y en el editor de código en Eclipse.

- Retire sesiones de cobertura para los lanzamientos retirados (*Remove coverage sessions for removed launches*). Sesión de cobertura estará disponible hasta que se eliminen manualmente en la vista de cobertura. Por defecto está desactivado.

- Restablecer datos de ejecución (*Reset Execution data in dump*). Esta opción determina si los datos de ejecución se restablecen después de volcado intermedio. Por defecto está desactivado.

2.- Ámbito (*Default Scope*).

- Sólo carpetas de origen (*Source folders only*). Por defecto está activado. Considerar origen basado en la ruta de entradas de clase. la ruta de clase basada en entradas

- Sólo el mismo proyecto (*Same project only*). Por defecto está desactivado. Escoja sólo las entradas de ruta de clases de un mismo proyecto. Esta opción sólo funciona para las configuraciones de lanzamiento que tienen un proyecto asociado.

- Sólo las entradas de ruta coincidentes (*Only path entries matching*). Lista de cadenas de texto separadas por comas que deben coincidir con la entrada de ruta de clases. Por ejemplo: “src/main/java”. Por defecto está sin filtro.

3.- Tiempo de ejecución (*Coverage Runtime*).

Por razones técnicas es posible que se quiera excluir ciertas clases de análisis de cobertura de código. Las entradas de la lista están separadas por dos puntos (:) y pueden utilizar caracteres comodín (* y?).

- Incluir (*Includes*): lista de nombre de las clases que deben ser incluidos en el análisis de ejecución. (Por defecto: *)

- Excluir (*Excludes*): Una lista de los nombres de las clases que se debe excluir del análisis ejecución. (Por defecto: *vacío*).

- Excluir cargadores de clases (*Exclude classloaders*). Una lista de nombres de cargadores de clases que se debe excluir del análisis ejecución. Esta opción puede ser necesaria en caso de *frameworks* especiales que entran en conflicto con la instrumentación de código JaCoCo. (Por defecto: *sun.reflect.DelegatingClassLoader*)

M.3.- METRICS

M.3.1.- INSTALACIÓN

Opción 1: Instalar desde MarketPlace de Eclipse.

1.- En Eclipse: en el menú seleccionar *Help -> Eclipse Marketplace*.

2.- Buscar "Metrics".

3.- Seleccionar "Install" para "Eclipse Metrics".

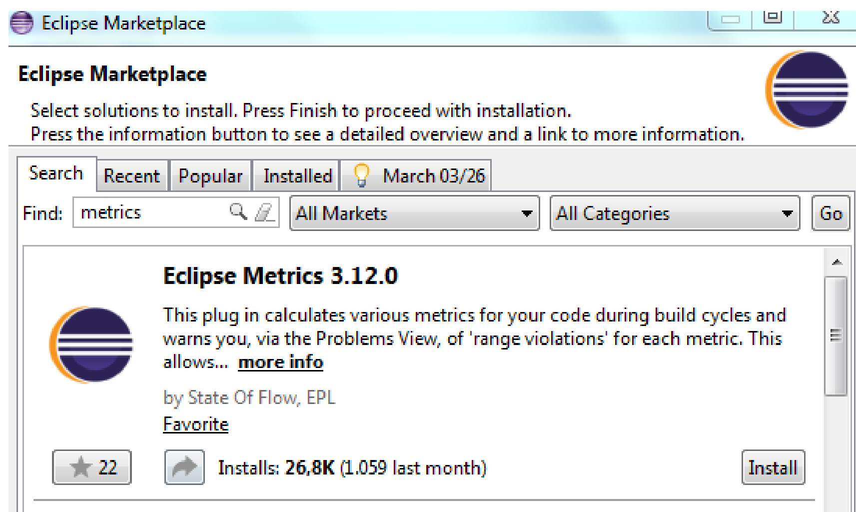


Figura 85. Metrics. Instalación desde Marketplace (I)

4.- Seguir los pasos para la instalación.

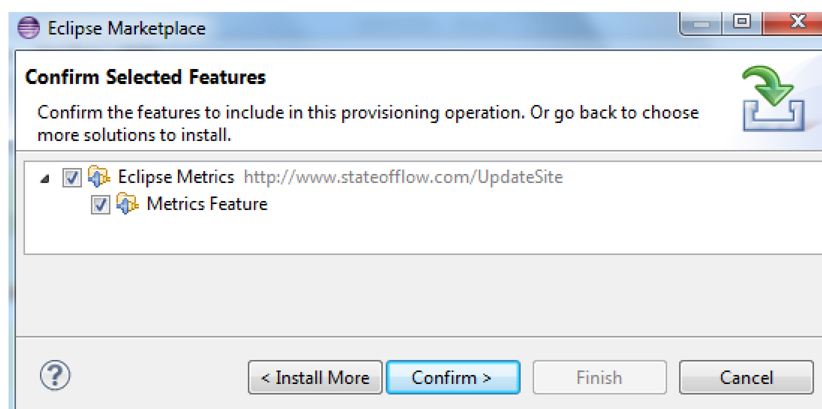


Figura 86. Metrics. Instalación desde Marketplace (II)

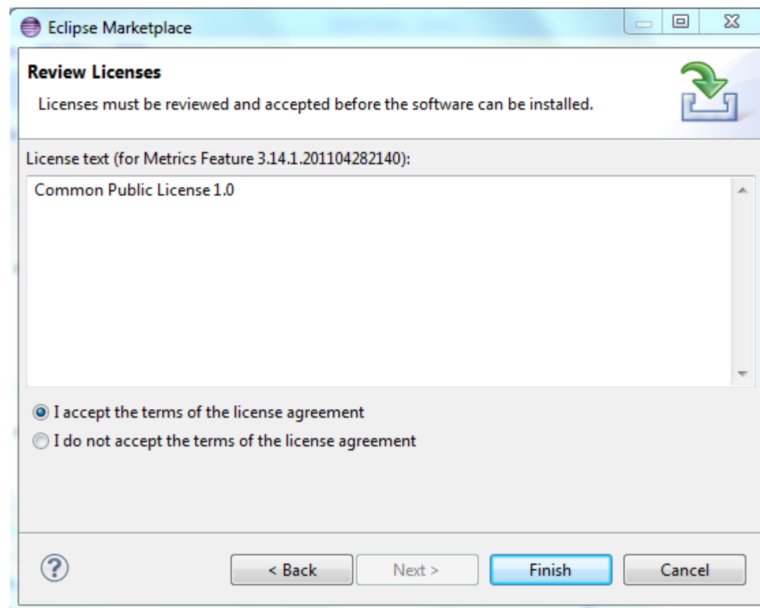


Figura 87. Metrics. Instalación desde Marketplace (III)

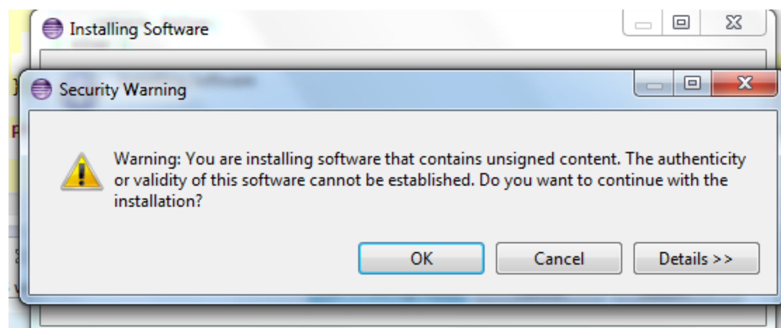


Figura 88. Metrics. Instalación desde Marketplace (IV)

Opción 2: Instalación desde el sitio de actualizaciones de Eclipse.

1.- En Eclipse: en el menú seleccionar *Help -> Install new software*.

2.- Pulsar en e botón Add... Añadir un nuevo sitio con la URL: http://metrics2.sourceforge.net/update/

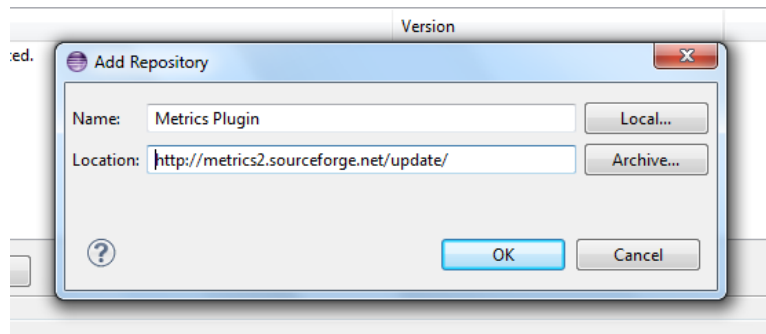


Figura 89. Metrics. Instalación desde el sitio de actualizaciones (I)

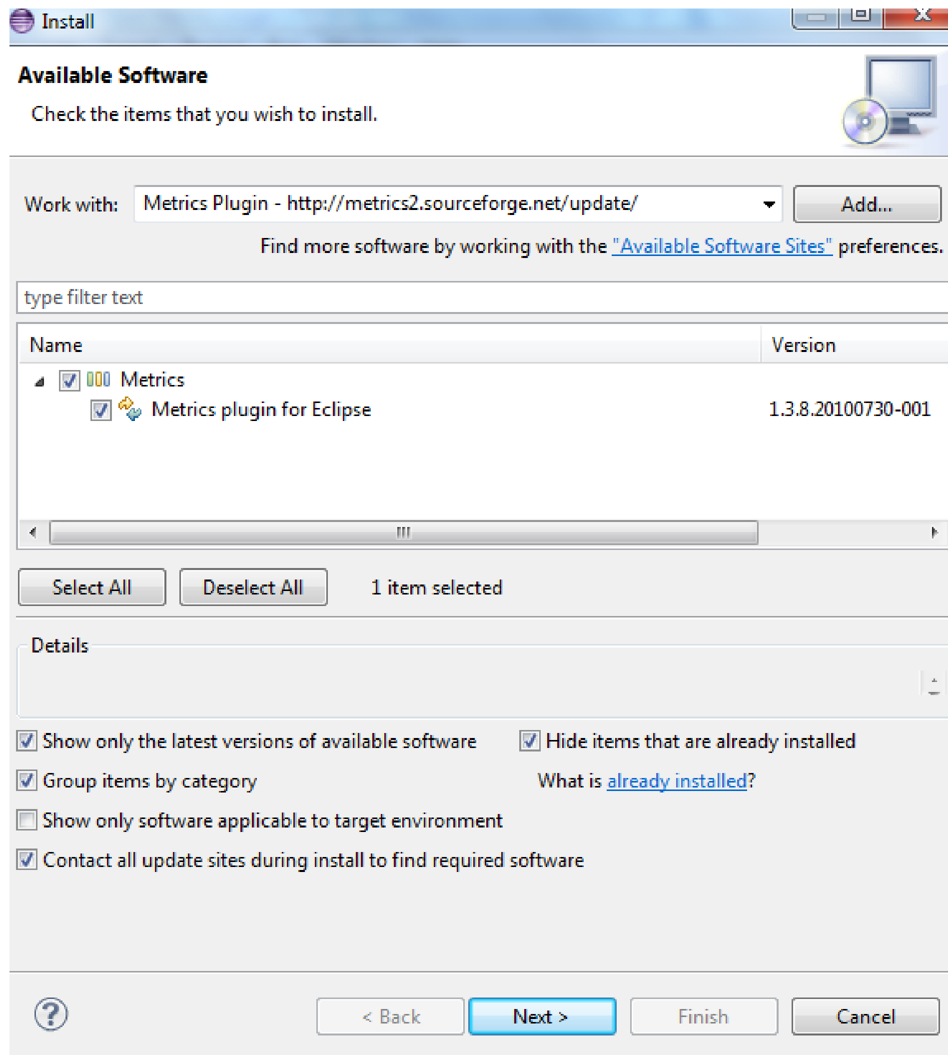


Figura 90. Metrics. Instalación desde sitio de actualizaciones (II)

3.- Hacer clic en "*Next*".

4.- Seguir los pasos de la instalación.

M.3.2.- GUÍA DEL USUARIO

1.- Para empezar a utilizar la vista de las Metrics: *Windows -> Show View -> Others* y la vista de Metrics aparecerá como en la imagen siguiente:

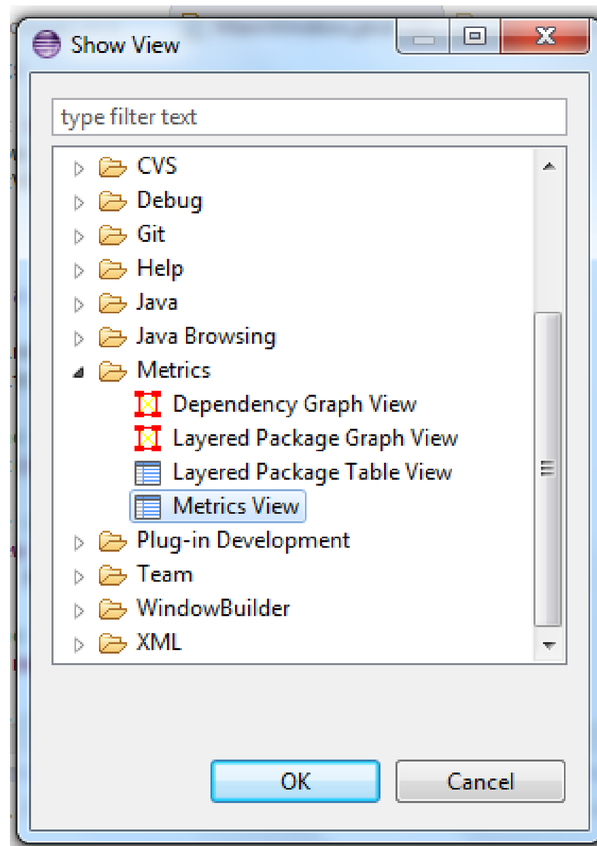


Figura 91. Metrics. Guía del usuario (Mostrar Vista)

Inicialmente no aparecerá ningún análisis ya que no se ha indicado que se calcule ninguna métrica a ningún proyecto.

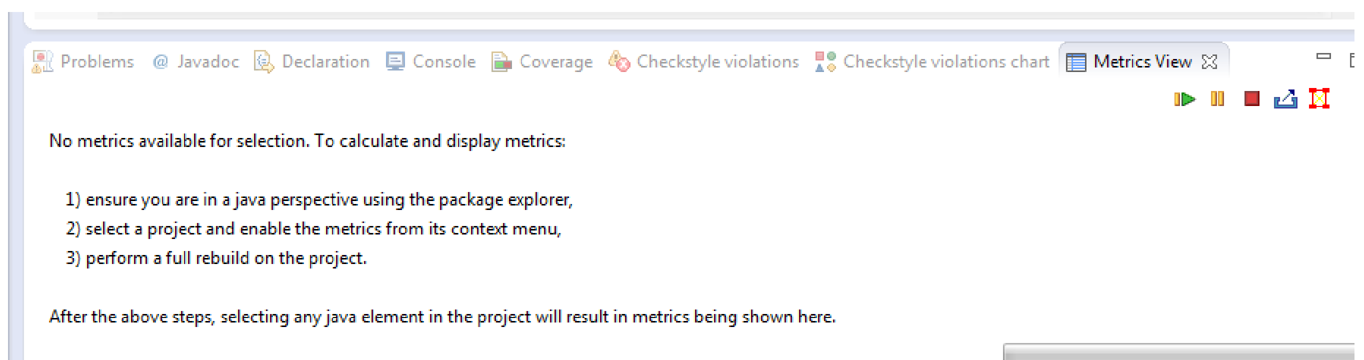


Figura 92. Metrics. Guía del usuario (Vista)

2.- El siguiente paso, para comenzar a recoger las métricas para un proyecto, haga clic derecho en el proyecto y en el menú emergente seleccione "Metrics-> Enable" (o, alternativamente, usar la página de propiedades).

Esto le dirá a Eclipse que hay que calcular métricas cada vez que haya una compilación. Ahora que ha habilitado un proyecto, la forma más fácil de calcular todos sus indicadores es hacer una reconstrucción completa de ese proyecto. La vista métricas indicará el progreso de los cálculos de métricas, ya que se están realizando en el fondo. Cuando todo está hecho, la opinión de las métricas se verá algo como esto:

Metrics - /net.sourceforge.metrics

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Packages	16					
Number of Methods (avg/max per type)	1310	6.65	8.553	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
tgsrc	489	7.191	11.544	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
src	761	6.238	6.553	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core.sources	108	15.429	12.129	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui	77	9.625	10.111	33	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core	198	6.6	7.093	27	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.preferences	52	6.5	7.467	26	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.dependencies	95	5.588	3.727	15	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.persistence	18	4.5	4.33	12	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler.implementa...	54	5.4	2.871	10	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.xml	41	4.1	2.022	9	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.calculators	79	4.158	2.254	8	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.propagators	31	5.167	1.067	7	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.tests	8	2.667	1.886	4	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler	0	0	0			
classycle	60	8.571	2.556	13	/net.sourceforge.metrics/classycle/classycle/g...	
Lines of Code (avg/max per type)	6593	33.467	49.02	339	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
Number of Interfaces (avg/max per packageFragment)	16	1	1.414	4	/net.sourceforge.metrics/src/net/sourceforge/...	
Lines of Code (avg/max per method)	6593	4.812	7.355	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
classycle	324	5.4	9.94	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
tgsrc	2321	4.661	8.278	59	/net.sourceforge.metrics/tgsrc/com/touchgrap...	scrollSelectPanel
src	3948	4.862	6.473	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
net.sourceforge.metrics.ui	544	6.8	8.707	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable.java	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
setMetrics	52					

Figura 93. Metrics. Guía del usuario (Resultado de métricas en su vista)

La tabla de métricas es en realidad un árbol, lo que le permite ampliar cada métrica para mostrar los valores en niveles por debajo del recurso seleccionado.

M.3.2.1.- PREFERENCIAS

En *Window > Preference*…Se selecciona *Metrics Preference*.

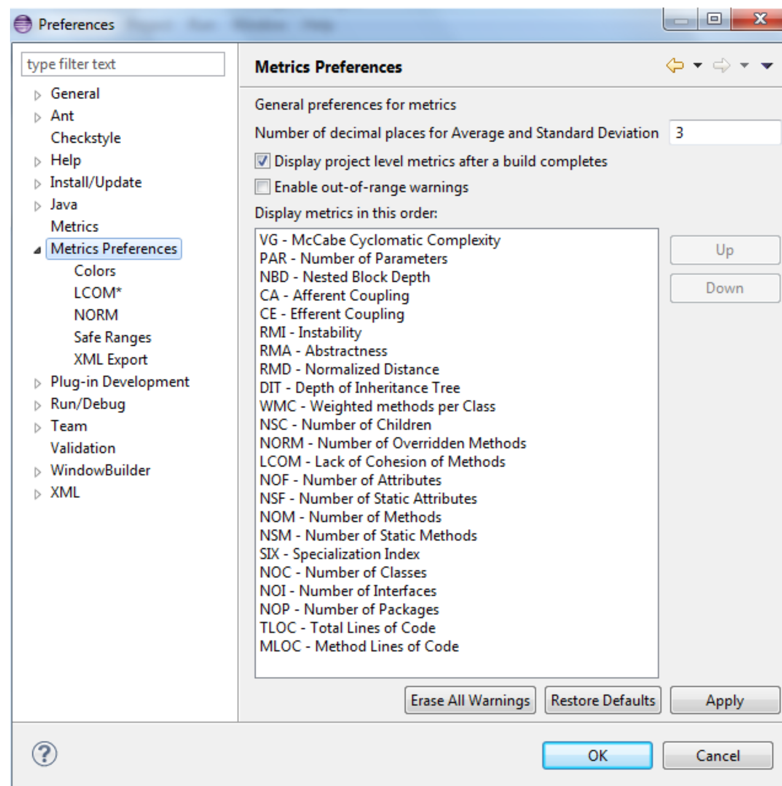


Figura 94. Metrics. Preferencias

Las preferencias de métricas permiten cambiar el orden de presentación de las métricas y la base de datos de métricas para ser borradas (forzar a recalcular todas las métricas). Además, la página de referencia de nivel superior sirve como una categoría para páginas de preferencias de métricas individuales.

Hay diferentes subapartados:

- **Colors**: se puede cambiar lo colores con los que aparecerá las mediciones en la vista de métricas. Se utilizan tres colores:
    - Color para los valores dentro del alcance (Por defecto, negro).
    - Color para los valores fuera del rango (por defecto, rojo). Se pueden acceder haciendo doble clic.
    - Color para los colores dentro del rango y permite acceder a la fuente donde se encuentra haciendo doble clic sobre él (por defecto, azul).

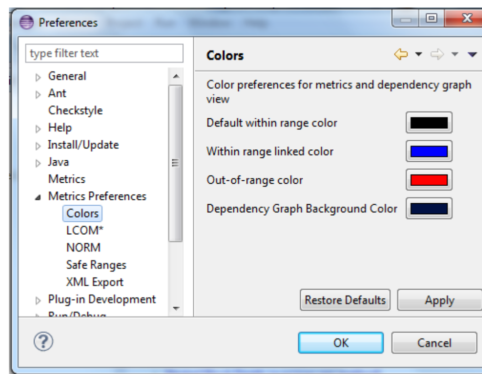


Figura 95. Metrics. Preferencias (Color)

- **LCOM**: hace referencia a la falta de cohesión de métodos. En la siguiente imagen se ve que se puede escoger si controlar los campos estáticos y/o métodos estáticos o no.

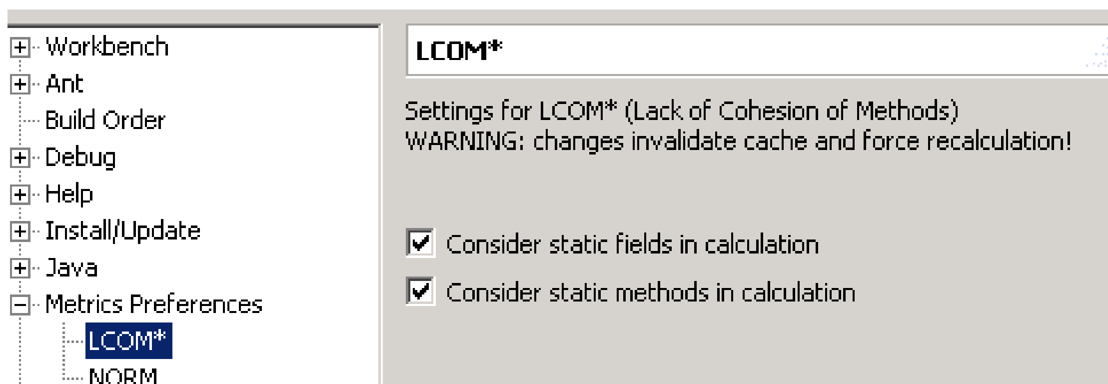


Figura 96.Metrics. Preferencias(Personalización de la métrica LCOM)

- **NORM**: se refiere al número de métodos sobrecargados. El cálculo de NORM puede personalizarse como se muestra en la imagen siguiente:

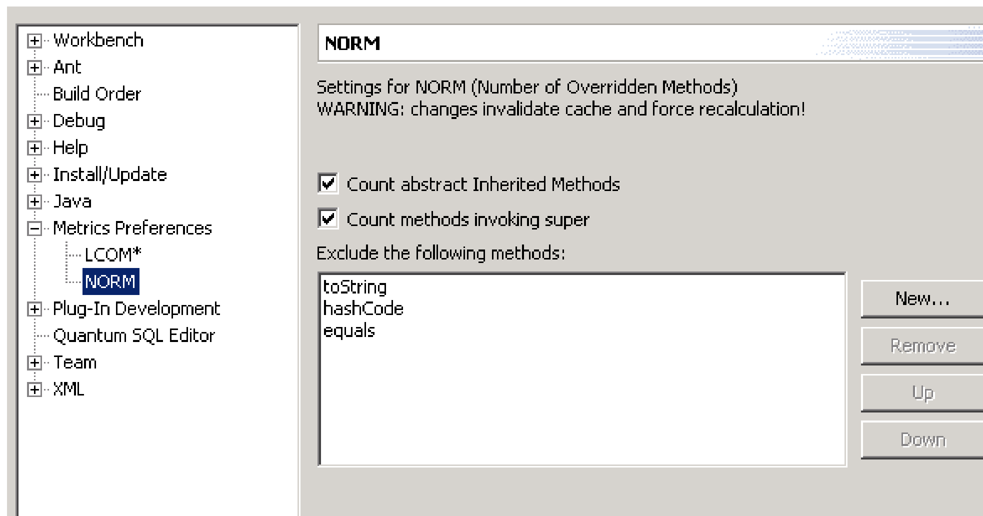


Figura 97. Metrics. Preferencias (Personalización de la métrica NORM)

Aquí se puede controlar si se debe contar los métodos abstractos, métodos que invocan a *super*. Algunos métodos como *toString*, *equals* y *hashCode* puede ser excluidos. También se pueden añadir otros.

- **Safe Ranges**: aquí es donde se puede modificar los rangos de valores de cada métrica. El valor de las métricas que se salgan fuera del rango se pondrán como advertencias.

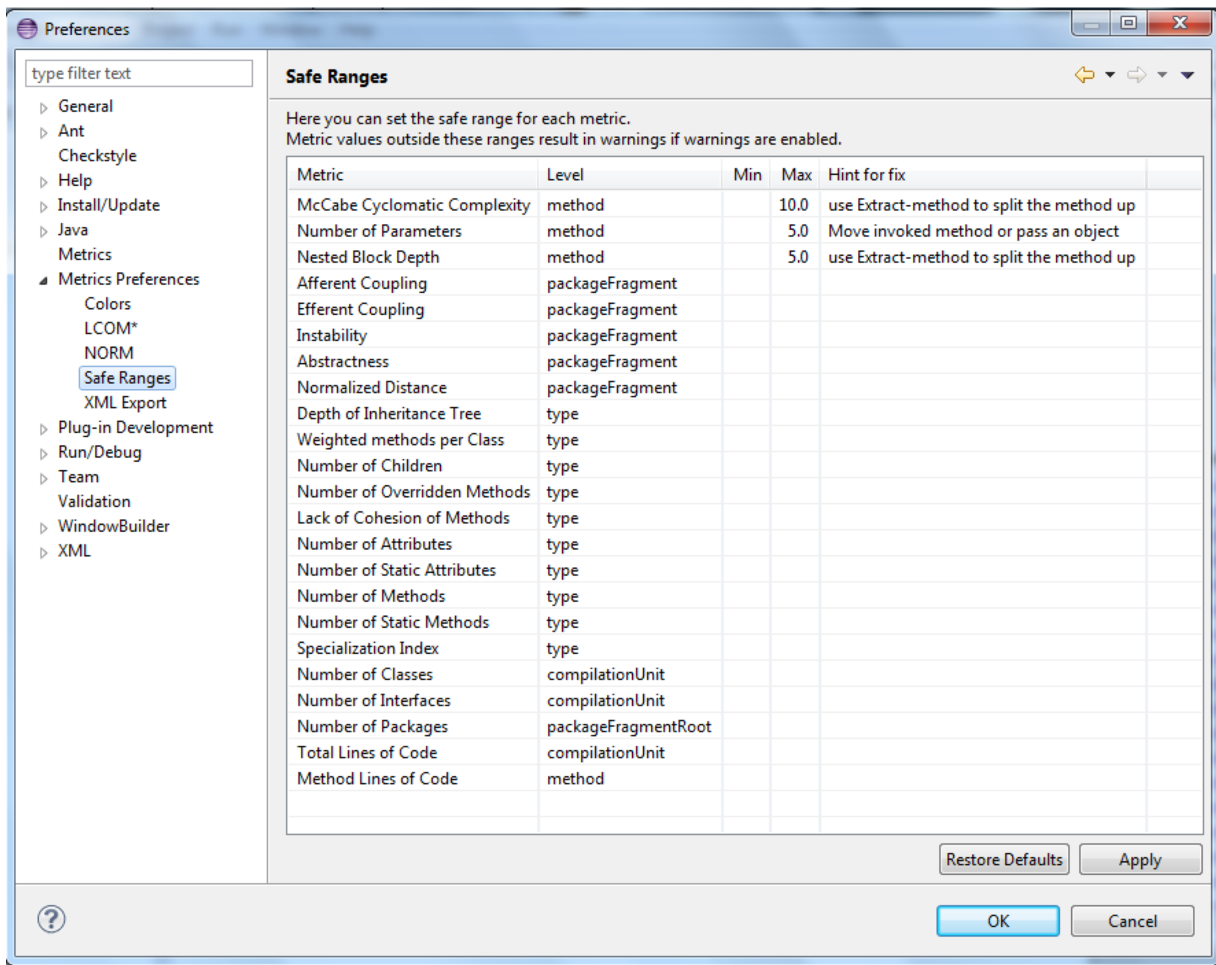


Figura 98. Metrics. Preferencias (Rango de valores)

Para establecer el rango de cada métrica se pulsa en el valor que se quiera modificar y se edita. Por ejemplo, si indico que el número total de líneas de código es 100, reconstruyo el proyecto y como es superior, esa métrica sale en color rojo.

Number of Classes (avg/max per packageFragment)	9	4,5	2,5	7	/game-of-life-master/src/edu/ma
Number of Interfaces (avg/max per packageFragment)	1	0,5	0,5	1	/game-of-life-master/src/edu/ma
Number of Packages	2				
Total Lines of Code	530				
Method Lines of Code (avg/max per method)	335	6,204	8,04	47	/game-of-life-master/src/edu/ma

Figura 99. Metrics. Fuera del rango

- **XML Exports**: Las métricas se pueden exportar a un archivo XML que se procesen con XSL en cualquier tipo de informe que desee. Para exportar métricas, se selecciona el ámbito (proyecto, paquete, etc.) por lo que se muestra en la vista. A continuación, se usa el menú desplegable de la barra de herramientas o la vista para seleccionar la función de exportación.

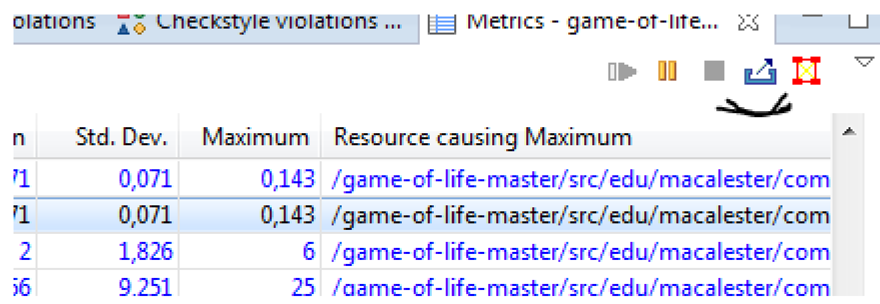


Figura 100. Metrics. Preferencias (Botón para exportar)

BIBLIOGRAFíA

(1) Fernández, L. y Lara, P., 2004, "Proceso y Herramientas para la productividad en el aseguramiento y medición de calidad en desarrollos Java" en *Revista de Procesos y Métricas de las Tecnologías de la Información* (RPM) (1), 2, 31-41

(2) IEE Standard for Software Quality Assurance Processes, 2014. IEEE Std 730-2014, P.1-138.

(3) CheckStyle. Año. 2010. Web. http://checkstyle.sourceforge.net/

(4) SUN, 1999, Code Conventions for the Java Programming Language. http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

(5) Google Java Style. Last changed: March 21, 2014 http://google-styleguide.googlecode.com/svn/trunk/javaguide.html

(6) Fenton, N. E. y Pfleeger, S.L., Software metrics. A rigorous and practical approach, PWS Pub., 1997.

(7) Pressman, R. "Ingeniería del software. Un enfoque práctico". McGraw-Hill, 2005.

(8) Archer C., Stinson M. *"Object-Oriented SoftwareMeasures".* Technical Report CMU/SEI-95-TR-002, Abril 1995.

(9) Garcia D, Harrison R., 2000, "Medición en la Orientación a Objetos"en L. Fernadez y J. Dolado, Medición para la gestión en la Ingeniería del Software, Ra-Ma, pp. 75-92

(10) Brito e Abreu F., Melo W. "Evaluating the impact of Object- Oriented Design on Software Quality". *Proceedings of 3rd International Software Metrics Symp*., Berlin, 1996.

(11) Chidamber S. R., Kemerer C. F. "A metric suite forsz object oriented design", *IEEE Transactions on Software Engineering*, pp. 467–493, 1994.

(12) Lorenz M., Kidd J. *Object Oriented Metrics*. Englewood, NJ: Prentice Hall, 1994.

(13) McCabe, T. J. "A Complexity Measure", *IEEE Transactions on Software Engineering*, 2(4), 308-320, 1976.

(14) Robert Martin. OO design quality metrics; an analysis of dependencies, October 1994

(15) W. Li y S. Henry, "Maintenance Metrics for the Object Oriented Paradigm", 1st International Software Metrics Symposium, pp. 52-60, 1993.

(16) Henderson-Sellers, B. "Object-Oriented Metrics, measures of Complexity", Prentice Hall, 1996

(17) Houari A. Sahraoui, Robert Godin, Thierry Miceli: Can Metrics Help Bridging the Gap Between the Improvement of OO Design Quality and Its Automation?

(18) Laranjeira A. Luiz. Software Size Estimation of Object-Oriented System, IEEE Transactions on Software Engineering. Vol 16, No.5 May 1990