|  |
| --- |
| **Ingeniería del Software: Laboratorio**  **Cuaderno de prácticas. P7-8 – Métricas de mantenimiento del software** |

Contenido

[OBJETIVO 2](#_Toc37956611)

[RECOMENDACIÓN DE HERRAMIENTAS 2](#_Toc37956612)

[EJERCICIO TUTORIZADO 2](#_Toc37956613)

[Primer paso: Obtener el software sobre el que calcular métricas. 3](#_Toc37956614)

[Segundo paso: Obtener métricas mediante JavaNCSS 6](#_Toc37956615)

[Tercer paso: Obtener métricas mediante CKJM 12](#_Toc37956616)

[Cuarto paso: Obtener métricas mediante Dependency Finder. 15](#_Toc37956617)

[1) Herramienta DependencyFinder 16](#_Toc37956618)

[2) Herramientas OOMetricsGUI y OOMetrics 19](#_Toc37956619)

[Quinto paso. Analizar los resultados 25](#_Toc37956620)

[Análisis de paquetes 25](#_Toc37956621)

[Análisis de clases 26](#_Toc37956622)

[Análisis de métodos 27](#_Toc37956623)

# OBJETIVO



El objetivo de esta práctica es que el estudiante aprenda a utilizar herramientas para el cálculo automático de métricas del software orientado a objetos y a interpretar los resultados, identificando lo elementos del software que tendrán un mayor impacto en el mantenimiento del software.

# RECOMENDACIÓN DE HERRAMIENTAS



Netbeans. JavaNCSS. CKJM.



Dependency Finder.



# EJERCICIO TUTORIZADO

El software sobre el que se van a calcular las métricas se obtendrá del repositorio **sourceforge.net**, siendo un código escrito en Java con un número de clases adecuado para el propósito de esta práctica.

## Primer paso: Obtener el software sobre el que calcular métricas.

En nuestro caso el software será jChecs (programa fuente en Java, diseñado para introducir los conceptos básicos de la programación de juegos de ajedrez). Descargar el archivo

Pasos a seguir:

1. Descargar el archivo [jChecs-0.1.0-src.tgz](http://sourceforge.net/projects/jchecs/files/Sources/v0.1.0/jChecs-0.1.0-src.tgz/download) de la página web [https://sourceforge.net/projects/jchecs/files/Sources/v0.1.0/ *.*](https://sourceforge.net/projects/jchecs/files/Sources/v0.1.0/)
2. Crear un proyecto vacío en NetBeans.

- Abrir NetBeans.

- Crear un nuevo proyecto: File > New Project. En la ventana emergente **New Project** seleccionar Java with Ant como Categoría, y Java Application como Proyecto.

- En la ventana emergente **New Java Application** colocar en el campo **Project Name** el nombre *jChecs* y desmarcar Create Main Class.

- Pulsar el botón Finish. Entonces aparecerá el proyecto ***jChecs*** con sus carpetas asociadas, el cual puede verse en la ventana **Projects**.

- Seleccionar la carpeta Source Packages e iniciar la creación de un test de tipo JUnit, pero cancelar en la ventana Create/Update Test. Sólo se hace esto para que se creen las carpetas “Test Packages” y “Test Libraries”.

c. Integrar el código fuente de JChecs en el proyecto de NetBeans:

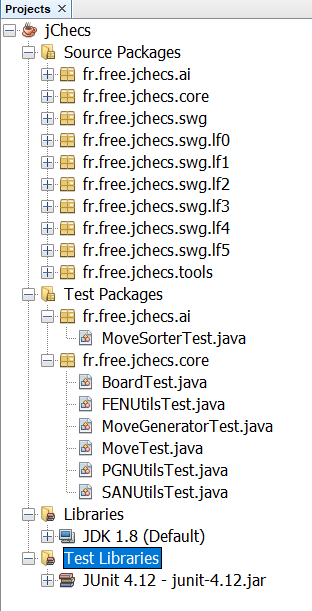
- Descomprimir dos veces el archivo *jChecs-0.1.0-src.tgz*, en cualquier parte (carpeta Documentos, por ejemplo). Se generará el directorio *jChecs-0.1.0*, que a su vez contiene los directorios *build* y *src*.

- Sustituir los directorios equivalentes que genera NetBeans, por los obtenidos anteriormente.

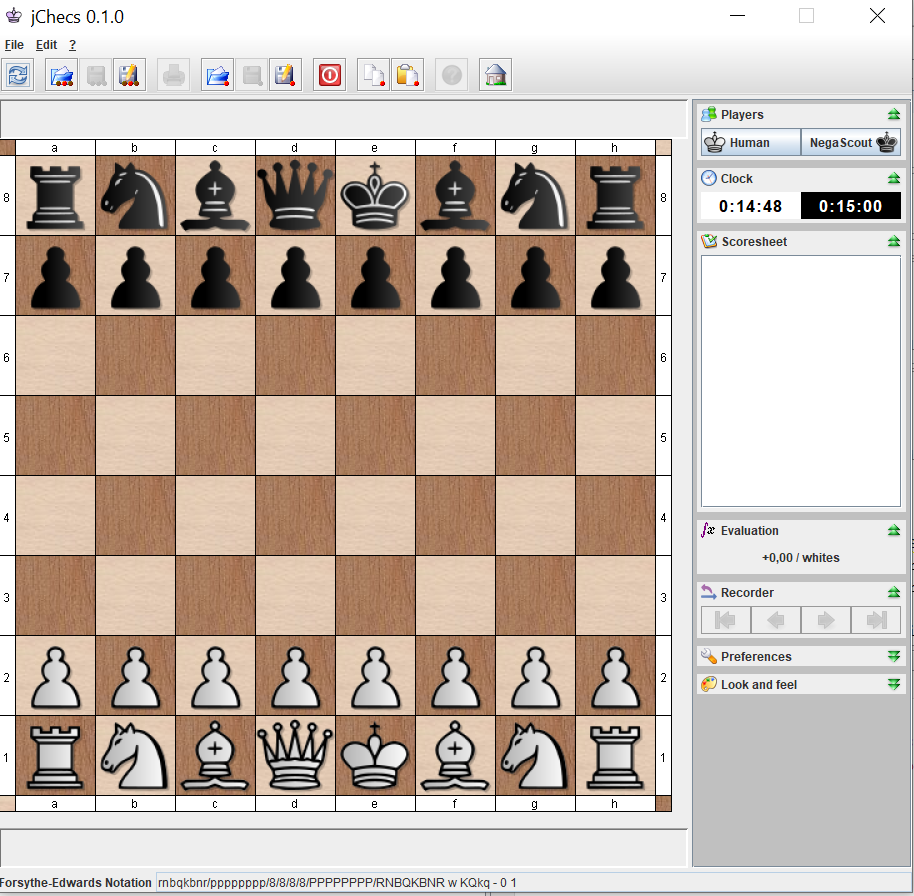
- En NetBeans aparecen errores en dos paquetes del proyecto, debido a que al autor de este programa ubicó los archivos de test junto con el código fuente del programa, y ahora están dentro de la carpeta “Source Packages” y hay que moverlos a la carpeta “Test Packages. Pata ello crear los siguientes paquetes en la carpeta “Test Packages”: **fr.free.jchecs.ai** y **fr.free.jchecs.core,** y **mover las clases de test que estásn marcadas con errores al paquete correspondiente**.

- Siguen apareciendo errores en los archivos de test, debido que están programados usando la librería JUnit 4, y hay que importarla, para ello seleccionar la carpeta “Test Libraries”, con el botón derecho del ratón elegir Add Library > JUnit 4.12.

* El contenido final de la ventana Projects debe ser el siguiente:



d. Compilar y ejecutar el programa, pulsando F6 o Run > Run Project, y eligiendo en la ventana emergente como clase principal la clase fr.free.jchecs.swg.Start.



En el directorio build del proyecto de NetBeans se habrá creado la carpeta build con los archivos compilados (.class), que se usarán en una sección posterior de esta práctica. Se puede cerrar NetBeans, pues no se utilizará en el resto de la práctica.

## Segundo paso: Obtener métricas mediante JavaNCSS

**JavaNCSS** es una herramienta que calcula métricas a partir del código fuente de archivos .java. Se obtiene gratuitamente de

[http://www.kclee.de/clemens/java/javancss/*.*](http://www.kclee.de/clemens/java/javancss/)

1. Descargar y descomprimir el archivo javancss-32.53.zip

(se recomienda descomprimirlo en el directorio raíz C:\).

1. (Opcional) Agregar a CLASSPAH las siguientes rutas (no es necesario si cuando se escriba el comando para usar JavaNCSS se incluyen las rutas en el propio texto del comando):

C:\javancss-32.53\lib\javancss.jar; C:\javancss-32.53\lib\ccl.jar; C:\javancss-32.53\lib\jhbasic.jar; C:\javancss-32.53\lib\javacc.jar

1. Copiar el proyecto NetBeans del programa JChecs que se va a evaluar en C:\jChecs.
2. Obtener las métricas del programa JChecs mediante JavaNCSS:

- Seleccionamos el menú Inicio de Windows, y dentro de éste

Ejecutar.

- Dentro de la ventana Ejecutar escribimos cmd para abrir una ventana de comandos.

- Nos situamos en el directorio raíz y escribimos:

|  |
| --- |
| java -classpath <<librerías>> javancss.Main -all -recursive –gui C:\jChecs\src |

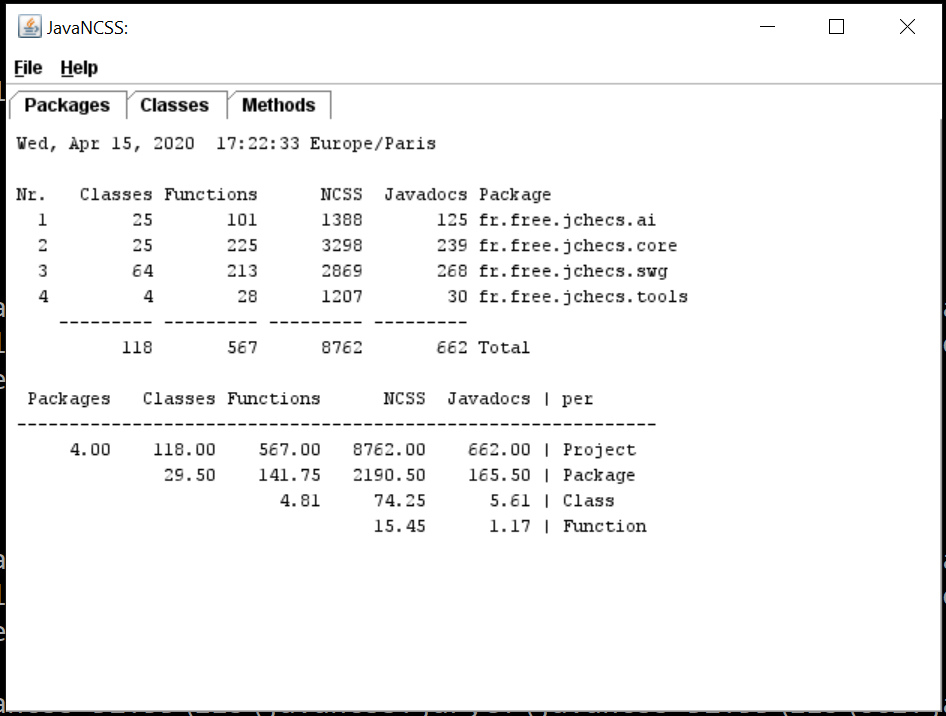
- Donde sustituimos <<librerías>> por la ruta de las librerías de JavaNCSS que vienen en la carpeta lib, separadas por punto y coma y sin espacios:

|  |
| --- |
| java -classpath .;C:\javancss-32.53\lib\javancss.jar;C:\javancss-32.53\lib\ccl.jar;C:\javancss-32.53\lib\jhbasic.jar;C:\javancss-32.53\lib\javacc.jar javancss.Main -all -recursive –gui C:\jChecs\src |

- La opción –recursive indica que se examinarán todos los archivos .java que cuelguen del directorio indicado.

- El argumento –gui invoca la interfaz gráfica.

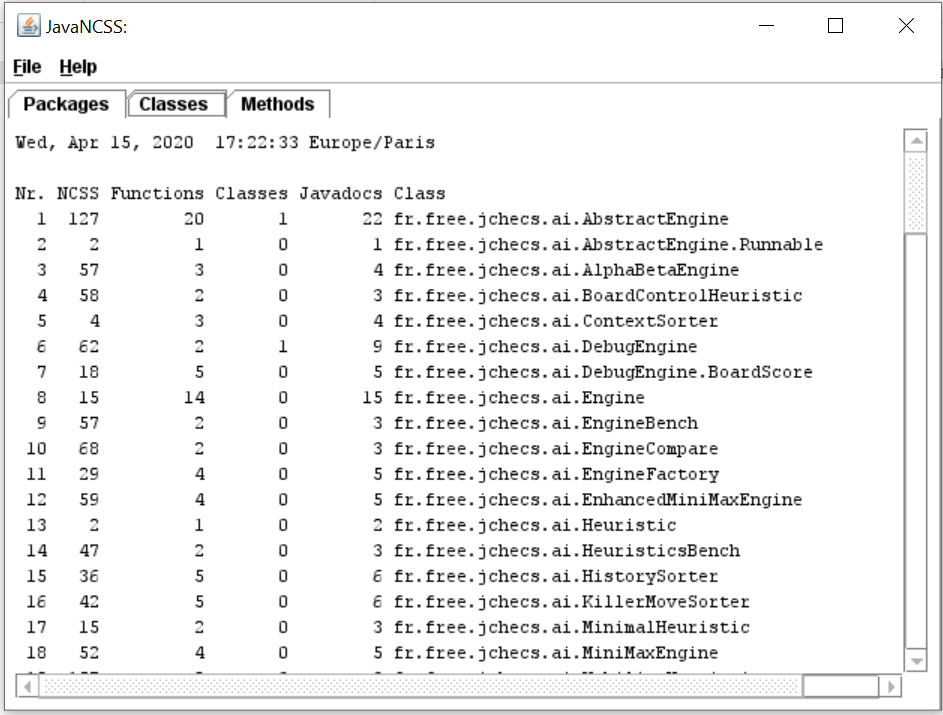
- El último argumento establece los archivos fuente para los que se realizará el análisis. En este caso los archivos ubicados en el directorio src. Si se desean analizar otros archivos habrá que indicar su ruta.



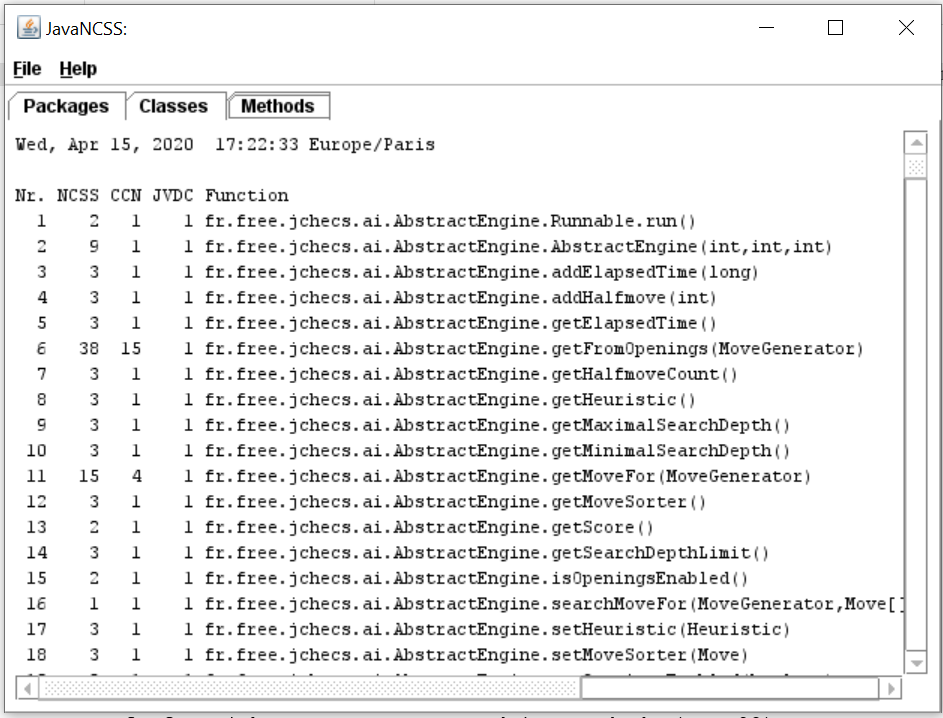
Donde:

* NCSS (Non Commenting Source Statements) son las líneas de código.
* Javadocs son los comentarios.
* Functions son los métodos de las clases.

En la pestaña Classes puede verse el número de funciones (métodos) que tiene cada clase, que puede dar idea de su complejidad:

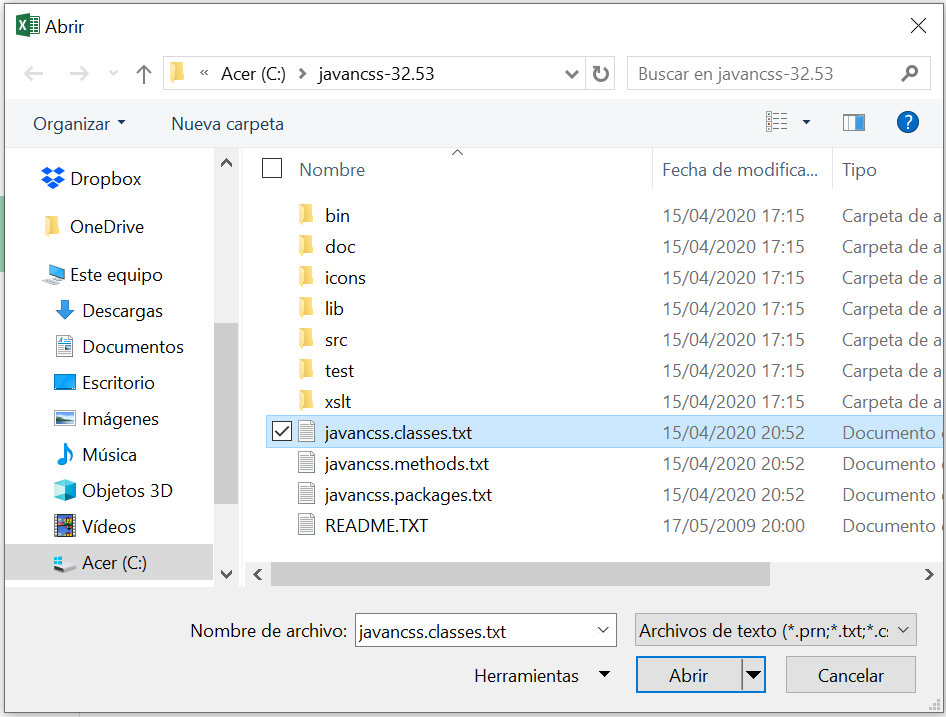


En la pestaña Methods, puede verse la complejidad ciclomática (CCN) de cada método, líneas de código (NCSS) y comentarios (JVDC).



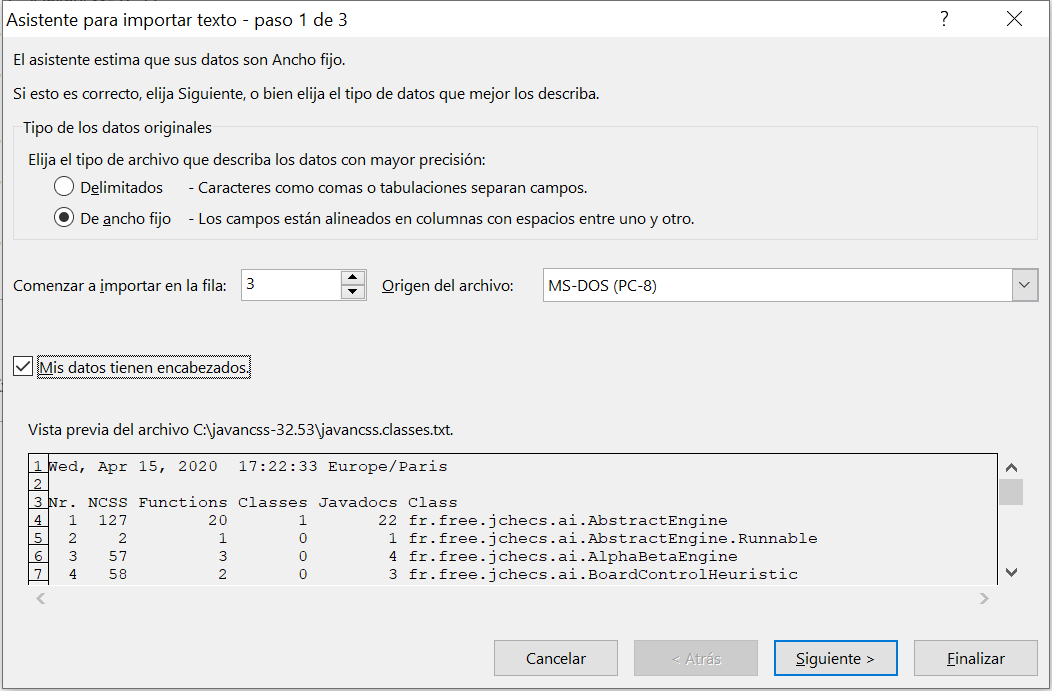
La información obtenida se puede grabar seleccionando File > Save, generándose tres archivos en el directorio javancss-32.53: javancss.packages.txt, javancss.classes.txt y javancss.methods.txt, los cuales contienen respectivamente la información sobre paquetes, clases y métodos.

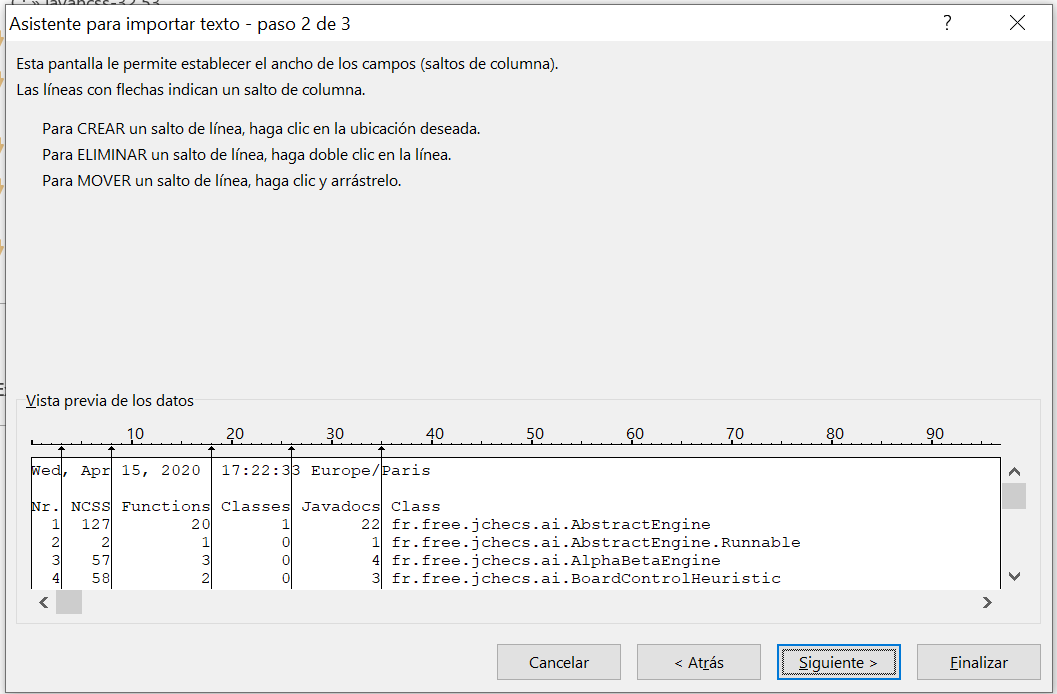
Para ordenar la información, se pueden convertir a hojas Excel. Para ello se abre cada archivo con Excel. Por ejemplo, el archivo javancss.clases.txt.

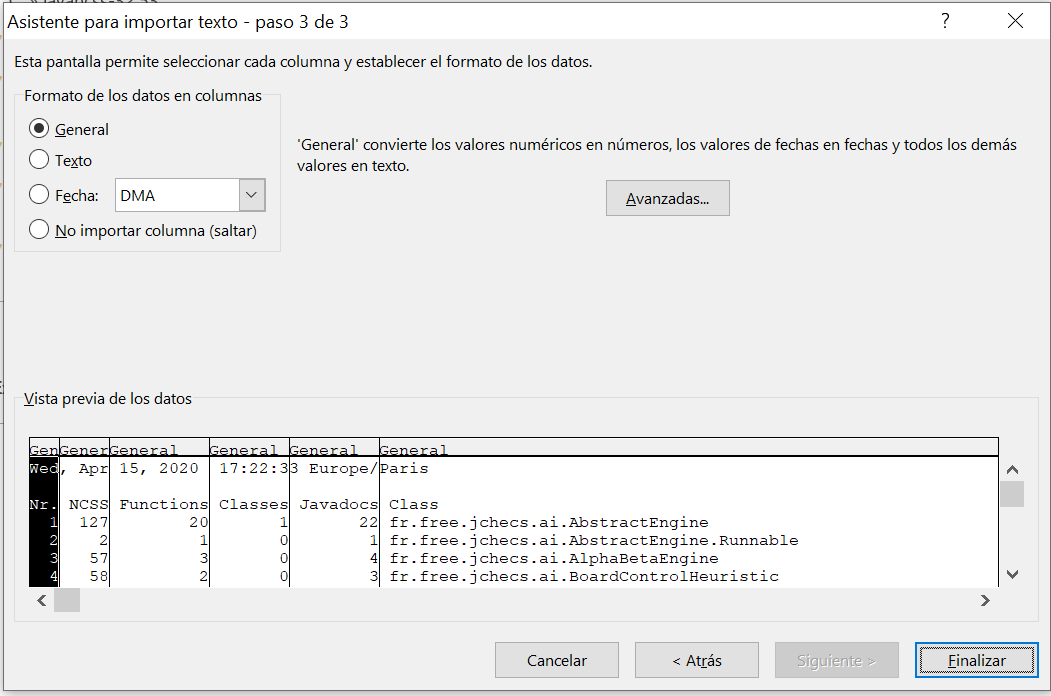


Y en el asistente para importar texto que aparece se marca (en el caso de classes.txt):

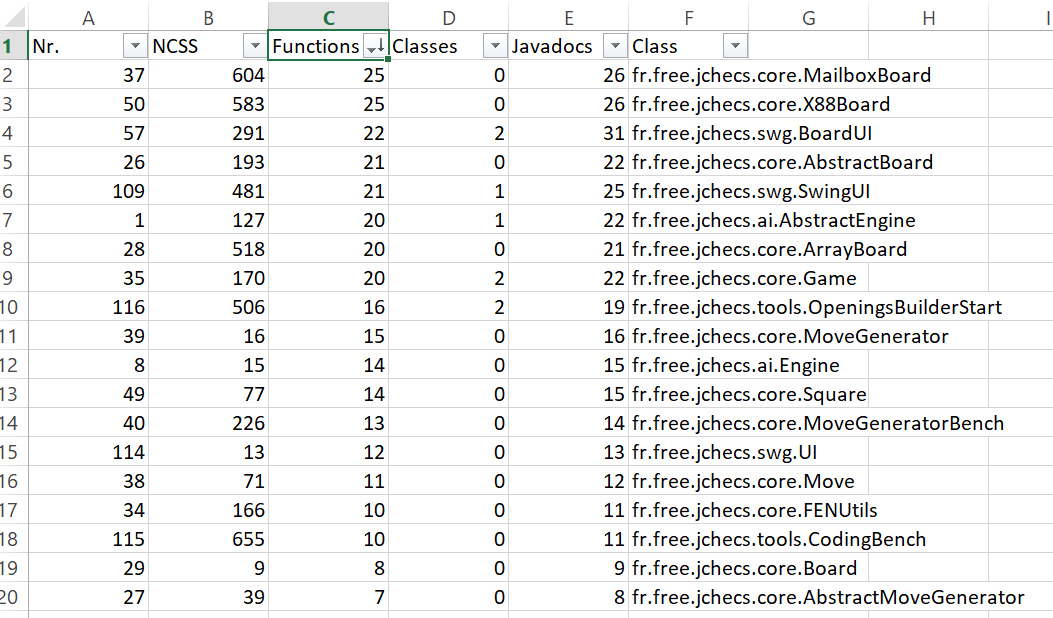
* Paso 1: Tipo: Ancho fijo, Comenzar a importar en la fila: 3, Mis datos tienen encabezado
* Paso 3: Formato de los datos en columnas: General







Y se obtiene un archivo Excel que se puede ordenar por columnas, por ejemplo por número de funciones que incluye cada clase:



Si aparecen filas duplicadas, se pueden eliminar desde Datos > Quitar duplicados.

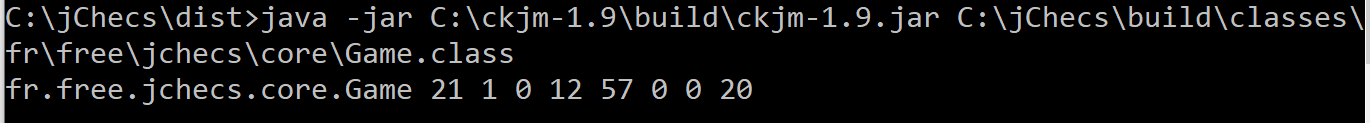
En el archivo “métricas-jchecs.xlsx” que se adjunta como anexo a esta práctica pueden encontrarse en diferentes hojas las métricas obtenidas.

## Tercer paso: Obtener métricas mediante CKJM

**CKJM** es una herramienta que calcula las métricas para software orientado a objetos propuestas por los autores Chidamber y Kemerer, procesando el bytecode de un fichero java compilado, es decir, los archivos .class.

a. CKMJ puede descargarse gratuitamente de: <http://www.spinellis.gr/sw/ckjm/>. Tras descargar y descomprimir el fichero gz o el fichero zip, invocamos el programa ejecutando java con la opción –jar pasándole como argumento la localización del archivo ckjm.jar que obtuvimos anteriormente. Además especificamos el fichero .class que queremos analizar, en el siguiente ejemplo seria Game.class.

|  |
| --- |
| java –jar C:\ckjm-1.9\build\ckjm-1.9.jar C:\jChecs\build\classes\fr\free\jchecs\core\Game.class |



Los datos obtenidos son (en ese orden):

Weighted methods per class (WMC) Depth of Inheritance Tree (DIT) Number of Children (NOC)



Coupling between object classes (CBO)

Response for a Class (RFC)



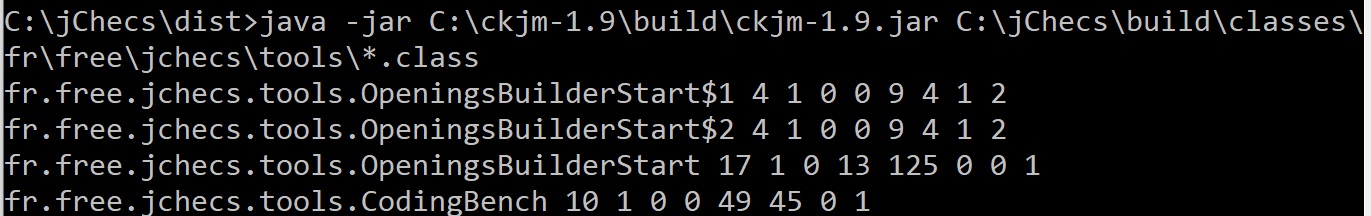
Lack of cohesion in methods (LCOM) Afferent coupling (not a C&K metric) (Ca)

Number of Public Methods for a class (not a C&K metric) (NPM)



d. Para analizar todas las clases de un paquete especificamos la carpeta que las contiene y dentro de ella todos los archivos con extensión .class, así vemos las métricas del paquete fr.free.jchecs.tools.

|  |
| --- |
| java –jar C:\ckjm-1.9\build\ckjm-1.9.jar C:\jChecs\build\classes\fr\free\jchecs\tools\\*.class |



1. Si se quieren obtener las métricas para todas las clases de un proyecto en varios directorios o carpetas, habría que obtener todos los archivos del directorio build y de sus subdirectorios con el comando dir y la opción /s y también /b para que sólo se obtenga el nombre de cada archivo. A continuación se filtrarían los archivos que posean la extensión .class. Esa salida se la pasaríamos a la herramienta ckjm como se muestra a continuación.

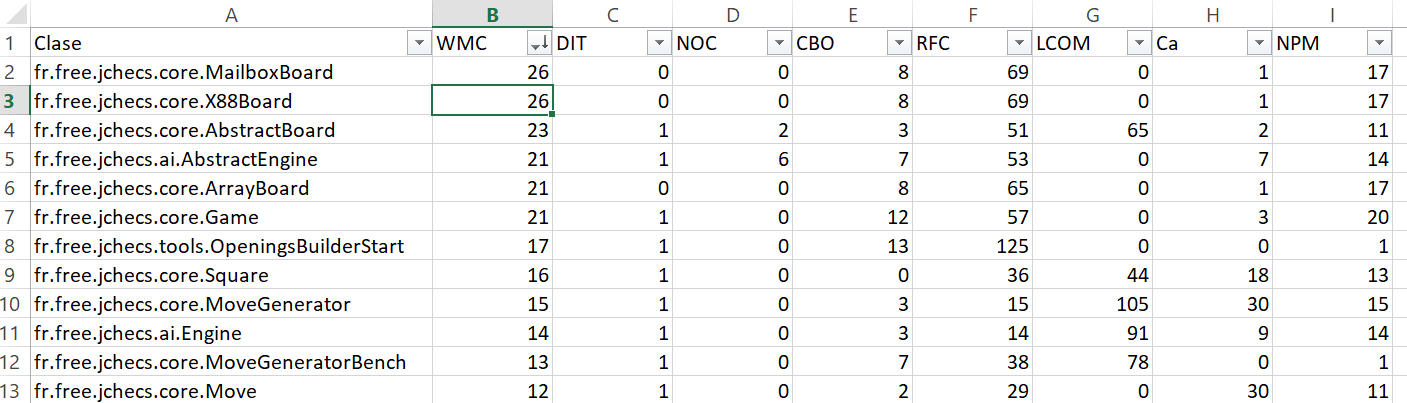
|  |
| --- |
| dir C:\jChecs\build /s /b | find ".class" | java –jar C:\ckjm-1.9\build\ckjm-1.9.jar |

NOTA: Es posible que surja algún error al tratar las clases del directorio “swg”, depende de la versión del compilador utilizado al construir el proyecto con NetBeans. Para este ejemplo se recomienda eliminar dicho directorio.

1. Se puede guardar el resultado obtenido, se debe redireccionar la salida obtenida hacia un fichero de texto, así como la salida con los mensajes de error.

|  |
| --- |
| dir C:\jChecs\build /s /b | find ".class" | java –jar C:\ckjm-1.9\build\ckjm-1.9.jar > C:\InformeCkjm.txt 2> C:\InformeCkjmErrores.txt |

En el primer archivo se encuentran los resultados válidos, que se pueden convertir a hoja Excel como se hizo con los de la herramienta anterior.



Se podrían integrar en el mismo archivo Excel para tener todos los resultados agrupados y poderlos comparar. Por ejemplo, la métrica WMC que representa los métodos por clase, en este caso el valor mayor lo tienen las mismas clases que con JavaNCSS, pero el valor no es exactamente igual:

* fr.free.jchecs.core.MailboxBoard (JavaNCSS=25, ckjm=26)
* fr.free.jchecs.core.X88Board (JavaNCSS=25, ckjm=26)

## Cuarto paso: Obtener métricas mediante Dependency Finder.

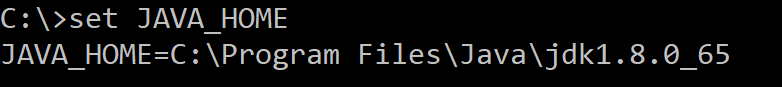
**Dependency Finder** es una *suite* de herramientas para analizar cualquier código Java compilado (.jar, .zip, .class). Analizando las características internas del código, es capaz de obtener grafos de dependencias, así como otras medidas de utilidad. Puede descargarse gratuitamente de

[http://depfind.sourceforge.net/.](http://depfind.sourceforge.net/)

Una dependencia se produce cuando una clase usa servicios de otra, por ejemplo posee un atributo de otra clase o sus métodos llaman a métodos ajenos. La dependencia se manifiesta cuando un elemento A requiere la presencia de otro B, entonces decimos que A depende de B y lo representamos por A B.



a. Después de descargar y descomprimir el fichero zip, comprobamos que la variable de entorno JAVA\_HOME posee la dirección del JDK instalado. Se puede comprobar por línea de comandos ejecutando set JAVA\_HOME, por ejemplo, en este caso sí existe y tiene el valor C:\Program Files\Java\jdk1.8.0\_65



Si esta variable no existe habrá que crearla. Además hay que incluir en

la variable PATH la dirección a la carpeta bin dentro de

DependencyFinder-1.2.1-beta5:

JAVA\_HOME=C:\Program Files\Java\jdk1.8.0\_65

PATH=C:\DependencyFinder-1.2.1-beta5\bin (y otros …)

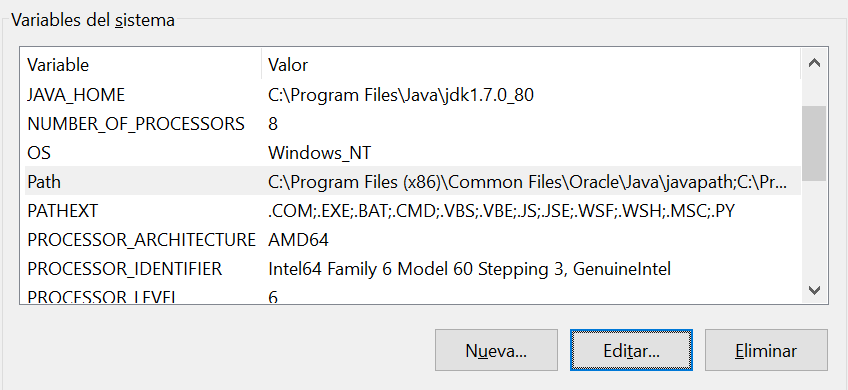
Para hacer esto hay que:

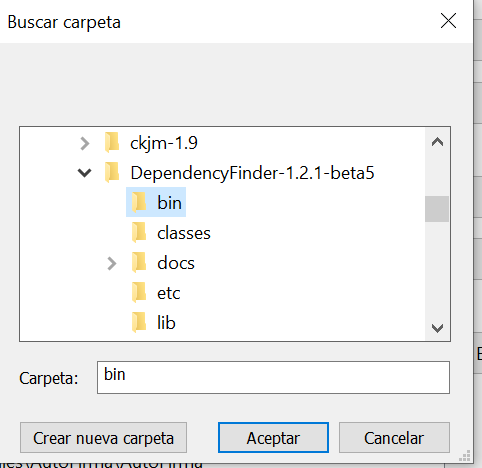
- Pulsar en el menú Inicio de Windows Inicio > Panel de Control > Sistema > Acerca de > Información del Sistema > Configuración avanzada del sistema > Variables de Entorno.

- La ventana Variables de Entorno permite ver las variables de usuario y de sistema existentes, así como sus valores. Podemos modificar, eliminar o añadir variables. En nuestro caso

añadiremos la variable JAVA\_HOME, y modificaremos la variable

PATH, todo ello desde la parte Variables del sistema, pulsando editar. En Path hay que añadir la ubicación del archivo





### 1) Herramienta DependencyFinder

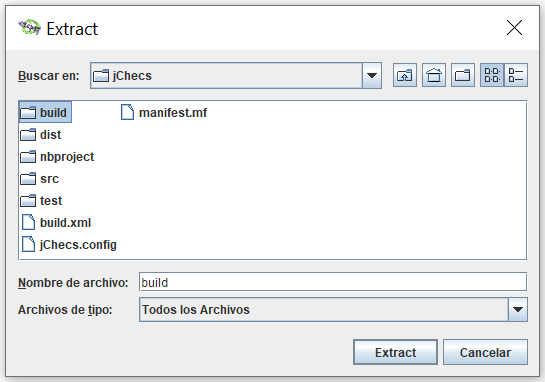
Dependency Finder incluye muchas herramientas el directorio bin, que. Una de ellas es DependencyFinder.bat. Se ejecuta desde el administrador de archivos de Windows o mediante el siguiente comando:

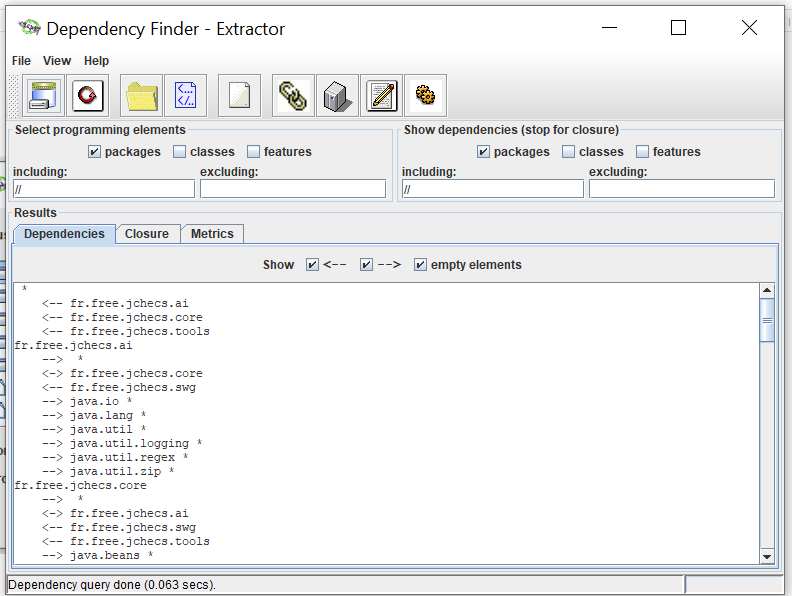
|  |
| --- |
| C:\DependencyFinder-1.2.1-beta5\bin\DependencyFinder.bat |

Como la ruta estaba en la variable Path del sistema, el comando se puede simplificar de esta forma:

|  |
| --- |
| DependencyFinder.bat |

Para seleccionar un proyecto utilizamos File > Extract. Para obtener el gráfico de dependencias File > Dependency.



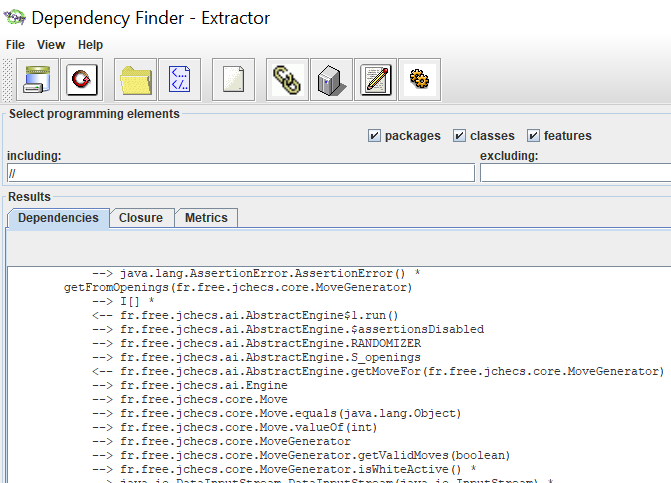


Se utiliza \* para distinguir identificadores que han sido referenciados en el código analizado, pero que no se analizan ya que generalmente son elementos de alguna librería java.

El informe mostrado está compuesto por nodos y links. Los nodos representan la estructura de nuestro proyecto (paquetes, clases, métodos), los links son las dependencias. Podemos filtrar nuestra búsqueda utilizando las secciones “*Select programming elements”* y

*“Show dependencies”*. En ellas encontramos *checkboxes* para buscar sólo paquetes, clases, métodos, variables (*features* se refiere a las características internas de las clases) o combinaciones de ellos.

Si marcamos todos los *checkboxes* obtendríamos información detallada, apareciendo el uso que dentro de una clase se hace de los métodos de otras clases, por ejemplo:

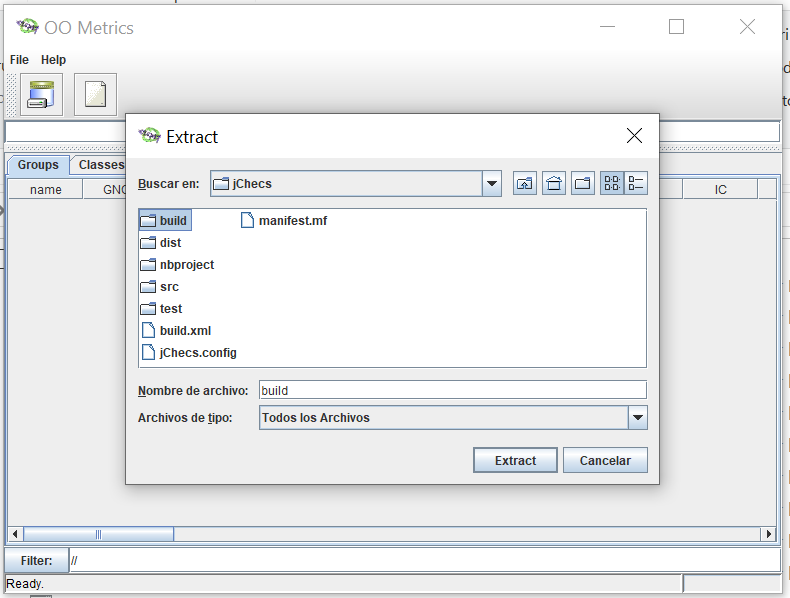


En esta imagen se puede observar que la clase AbstractEngine tiene una dependencia con la clase MoveGenerator del paquete fr.free.jchecs.core, ya que utiliza varios de sus métodos, como isWhiteActive().

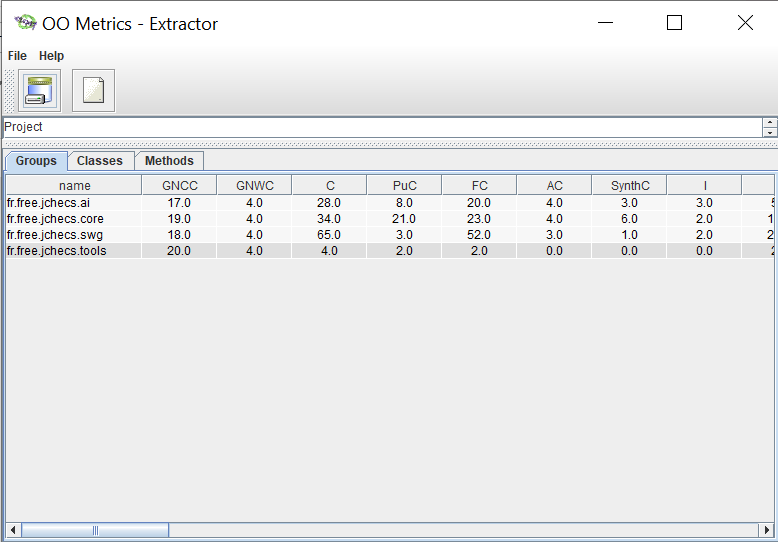
### 2) Herramientas OOMetricsGUI y OOMetrics

Otra herramienta muy interesante es OOMetricsGUI (situada también en el directorio bin), la cual permite obtener una serie de métricas a nivel de paquete, clase y de método. El archivo a ejecutar es OOMetricsGUI.bat.

|  |
| --- |
| C:\DependencyFinder-1.2.1-beta5\bin\OOMetricsGUI.bat |



En la primera ventana “Groups” aparecen métricas de los paquetes, como “C” que es el número de clases que contiene el paquete.



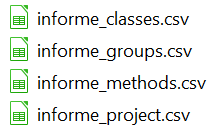
Es interesante fijarse en aquellos valores que están marcados en rojo, pues son los que, en opinión del autor de la herramienta, serían demasiado altos para la correspondiente métrica.

La herramienta no permite grabar los resultados en un archivo, pero hay una versión para usar en línea de comandos que si lo permite (OOMetrics.bat). Ejecutando el siguiente comando se guardan los resultados en archivos cvs que después se pueden importar en Excel.

|  |
| --- |
| C:\DependencyFinder-1.2.1-beta5\bin\OOMetrics.bat -csv –out informe C:\jChecs\dist\jChecs.jar |

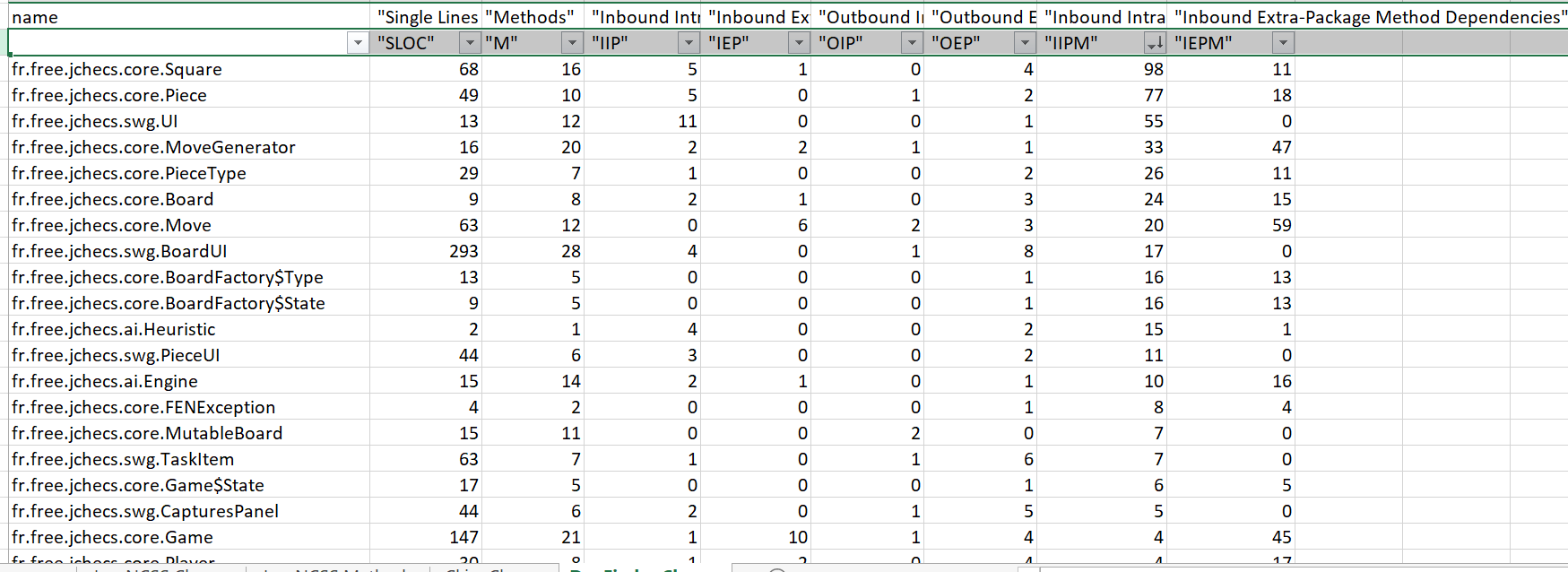
Este comando busca las clases en el archivo jar de la aplicación. Si no se había generado al ejecutarla, hay que volver a NetBeans y elegir la opción generar jar dentro de las propiedades del proyecto.

Se generan 4 archivos csv.



Se pueden importar con Excel. Para ello hay que abrir Excel y crear una nueva hoja de cálculo vacía, y elegir Datos > Desde texto > Se elige uno de los archivos > Importar > en paso 1 marcar “Delimitados” > en paso 2 marcar “Coma” > en paso 3 elegir “Avanzadas” > separador decimal = “.” > en ventana Importar datos elegir “Hoja de cálculo existente”.

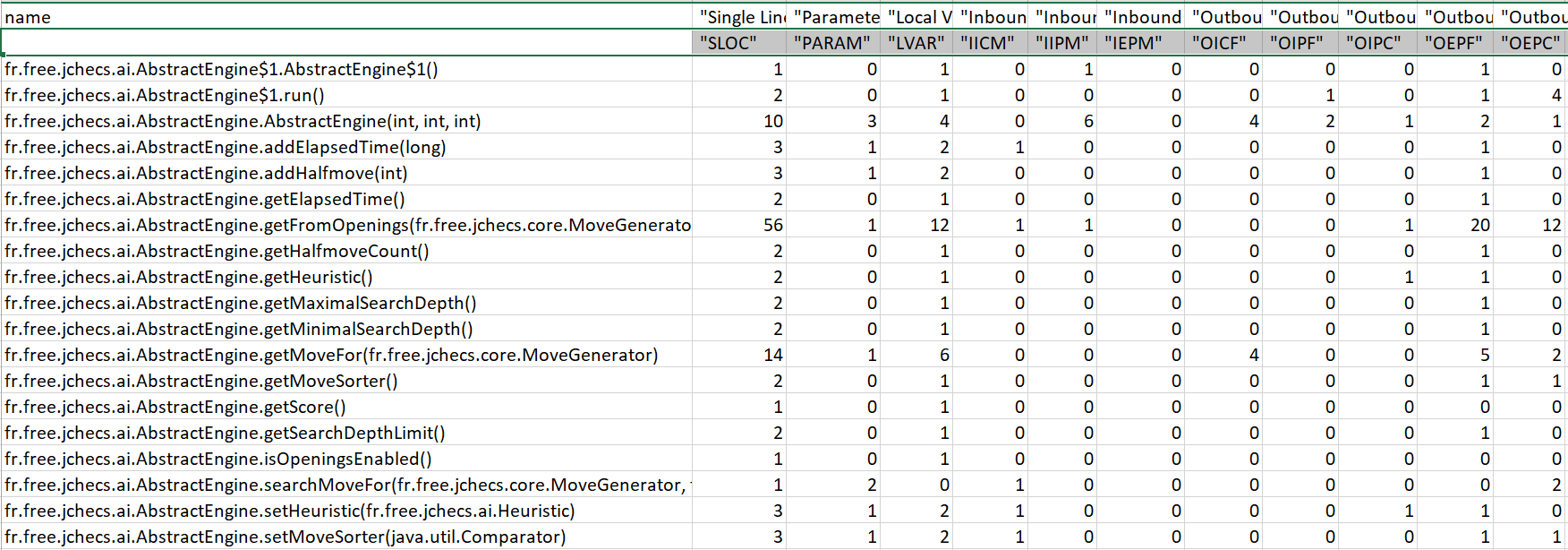
En la hoja resultante se pueden borrar las columnas de métricas poco importantes. Se recomienda ver el Excel adjunto a esta práctica, donde se han eliminado las métricas menos importantes. Como esta imagen de las métricas sobre clases.



Las métricas que se han importado son las siguientes (Ver manual de [DF](https://depfind.sourceforge.io/Manual.html)):

* Single Line of Code (SLOC): Number of code lines in the method. Comments and empty lines are not counted. Large methods are usually a sign of poor design.
* Methods (M): Number of methods in this class
* Inbound Intra-Class Method Dependencies (IICM): Other methods within the same class, that depend on this method.
* Outbound Intra-Class Feature Dependencies (OICF): Methods and fields within the same class that this method depends on.
* Inbound Intra-Package Method Dependencies (IIPM): Methods in other classes of the same package that depend on this method.
* Outbound Intra-Package Feature Dependencies (OIPF): Methods and fields within the classes of the same package that this method depends on.
* Inbound Extra-Package Method Dependencies (IEPM): Methods in other packages that depend on this method.
* Outbound Extra-Package Feature Dependencies (OEPF): Methods and fields in other packages that this method depends on.
* Outbound Intra-Package Class Dependencies (OIPC): Classes within the same package that this method depends on.
* Outbound Extra-Package Class Dependencies (OEPC): Classes in other packages that this method depends on.

En el caso de los métodos, las métricas destacables son:

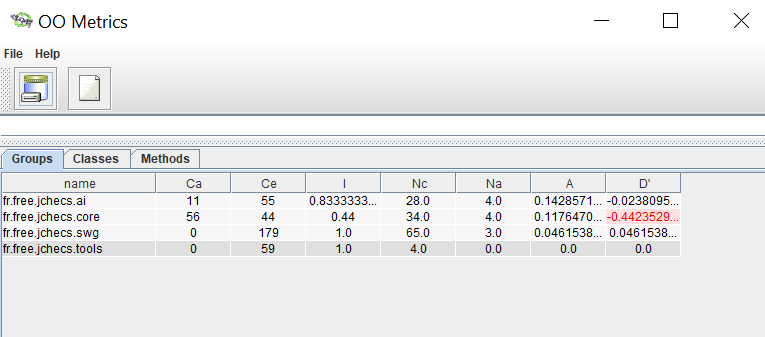


* Single Line of Code (SLOC): Number of code lines in the method. Comments and empty lines are not counted. Large methods are usually a sign of poor design.
* Parameters (PARAM): Number of parameters for the method. Very long signatures are often seen as a sign of poor design.
* Local Variables (LVAR): Number of local variables use by the method.
* Inbound Intra-Class Method Dependencies (IICM): Other methods within the same class, that depend on this method.
* Outbound Intra-Class Feature Dependencies (OICF): Methods and fields within the same class that this method depends on.
* Inbound Intra-Package Method Dependencies (IIPM): Methods in other classes of the same package that depend on this method.
* Outbound Intra-Package Feature Dependencies (OIPF): Methods and fields within the classes of the same package that this method depends on.
* Inbound Extra-Package Method Dependencies (IEPM): Methods in other packages that depend on this method.
* Outbound Extra-Package Feature Dependencies (OEPF): Methods and fields in other packages that this method depends on.
* Outbound Intra-Package Class Dependencies (OIPC): Classes within the same package that this method depends on.
* Outbound Extra-Package Class Dependencies (OEPC): Classes in other packages that this method depends on.

En general esta herramienta aporta información interesante sobre métricas relacionadas con dependencia a nivel de métodos, clases y paquetes.

A nivel de paquetes, tanto la versión visual como la de comando, permiten calcular el Índice de Inestabilidad de un paquete creada por Robert Martin, de esta forma:

|  |
| --- |
| C:\DependencyFinder-1.2.1-beta5\bin\OOMetricsGUI.bat –configuration C:\DependencyFinder-1.2.1-beta5\etc\MartinConfig.xml |



Para poderlo salvar en un archivo con nombre “informeMartin.cvs”:

|  |
| --- |
| C:\DependencyFinder-1.2.1-beta5\bin\OOMetrics.bat –configuration C:\DependencyFinder-1.2.1-beta5\etc\MartinConfig.xml –groups -csv -out informeMartin C:\jChecs\dist\jChecs.jar |

Estas son las métricas que se obtienen:

* Ca: Afferent Coupling. Número de clases de fuera de este paquete que dependen de clases de este paquete.
* Ce: Efferent Coupling. Número de clases de fuera de este paquete de las que dependen clases de este paquete.
* I: Inestability. I = Ce / (Ca + Ce).
* Nc: Número de clases del paquete.
* Na: Número de clases abstractas e interfaces en el paquete.
* A: Abstractness. A = Na / Nc.
* D’: Distancia normalizada. D' = A + I – 1.

Se puede observar que el paquete fr.free.jchecs.core tiene un número elevado para Ca, de manera que si se realizan cambios en el mismo, éste afectará a numerosas clases, las cuales (muy probablemente) a su vez tendrán que sufrir alguna modificación.

La Inestabilidad puede tomar valores en el rango comprendido entre 0 y 1. Valores cercanos a 1 indican máxima estabilidad, pero poca flexibilidad para modificar el software. Sin embargo valores muy bajos indican demasiada volatilidad. Sin embargo, para extraer conclusiones más certeras se suelen tener en cuenta tanto la Inestabilidad como la Abstracción, de manera que no es malo tener valores grandes de I si se tienen valores pequeños de A y no es malo tener valores pequeños de I si se tienen valores grandes de A. En otras palabras, lo mejor es un valor D’ cercano al cero.

## Quinto paso. Analizar los resultados

Una vez obtenidas todas las métricas, se debe realizar un análisis sobre los paquetes, clases y métodos que tendrán una peor “mantenibilidad”.

Se trata de revisar los valores de las métricas que son significativas en cada caso, y decidir qué elementos son los que pueden dar más problemas a la hora de hacer modificaciones en el software. Incluso si los valores de las métricas están en un rango recomendado por lo expertos.

Esto último es lo más difícil de decidir, pues no todos los autores se ponen de acuerdo en los valores más adecuados, suelen ser recomendaciones en base a la experiencia en muchos proyectos.

Lo que siempre hay que hacer en un informe en el que se indique si una elemento del software puede ser problemático de mantener por el valor de una métrica asociada, es citar la fuente en la que nos hemos basado para afirmarlo, que será el documento en el que algún experto justifica un rango de valores adecuados.

Algunas fuentes en las que se pueden encontrar recomendaciones sobre valores son las siguientes:

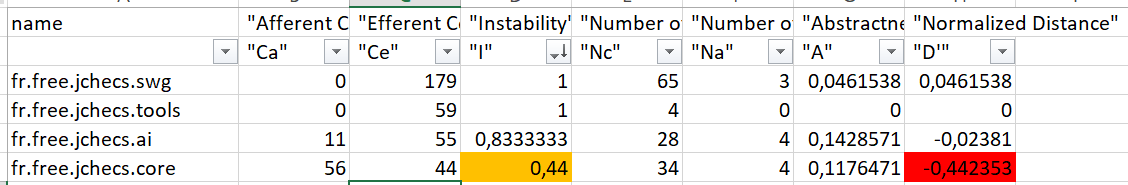
* Medición en la Orientación a Objetos: <http://www.cc.uah.es/drg/b/RodHarRama00.pdf>
* Métricas orientadas a objetos: <https://gredos.usal.es/jspui/bitstream/10366/21758/1/DPTOIA-IT-2001-002.pdf>
* Chidamber & Kemerer object-oriented metrics suite: <https://www.aivosto.com/project/help/pm-oo-ck.html>
* A metrics suite for object oriented design: <http://www.eso.org/~tcsmgr/oowg-forum/TechMeetings/Articles/OOMetrics.pdf>
* Umbrales para métricas orientadas a objetos: <https://www.researchgate.net/publication/228880050_UMBRALES_PARA_METRICAS_ORIENTADAS_A_OBJETOS>
* Un caso de estudio sobre la identificación de valores umbrales para medidas de código: <http://lbd.udc.es/jornadas2011/actas/JISBD/JISBD/S3/Regulares/jisbd2011_submission_68.pdf>
* RFC y el acoplamiento entre clases: <https://jummp.wordpress.com/2010/05/28/rfc-y-el-acoplamiento-entre-clases/>
* Niveles de riesgo según rango de Complejidad Ciclomática: <https://es.wikipedia.org/wiki/Complejidad_ciclom%C3%A1tica#Riesgo>
* La herramienta Dependency Finder: porque marca con rojo los valores que estima que no son adecuados según criterio de su autor: <https://depfind.sourceforge.io/Manual.html>

### Análisis de paquetes

En el caso de los paquetes, algunas métricas que se pueden analizar son:

* Número de clases (Classes en JavaNCSS, Nc en Dependency Finder)
* Número de métodos (Functions en JavaNCSS)
* Número de líneas de código (NCSS en JavaNCSS)
* Número de comentarios: (Javadocs en JavaNCSS). Cuanto mayor sea la relación entre comentarios y líneas de código, se supone que más fácil de mantener
* Ca: Afferent Coupling (Dependency Finder)
* Ce: Efferent Coupling (Dependency Finder)
* I: Inestability (Dependency Finder)
* Na: Número de clases abstractas e interfaces en el paquete (Dependency Finder)
* A: Abstractness (Dependency Finder)
* D’: Distancia normalizada (Dependency Finder)

Deben razonarse los valores recogidos en el archivo Excel. Se pueden marcar con un color los paquetes con los valores peores (por ejemplo, naranja). Y si hay recomendaciones de valores, con otro color los que no lo cumplen (por ejemplo, rojo). En este caso se marca en esta tabla que el paquete “core” puede ser problemático al tener una distancia normalizada negativa y leja a cero. La herramienta de Robert Martin de Dependency Finder lo señala en rojo. Además tiene la peor estabilidad (I=0,44).

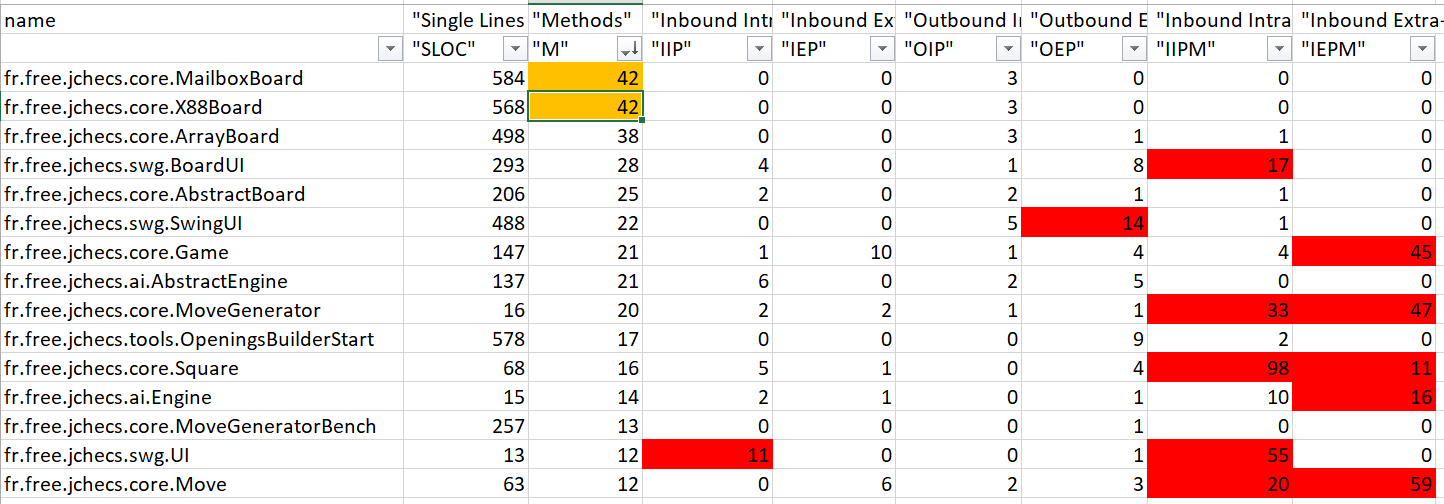


### Análisis de clases

En el caso de las clases, algunas métricas que se pueden analizar son:

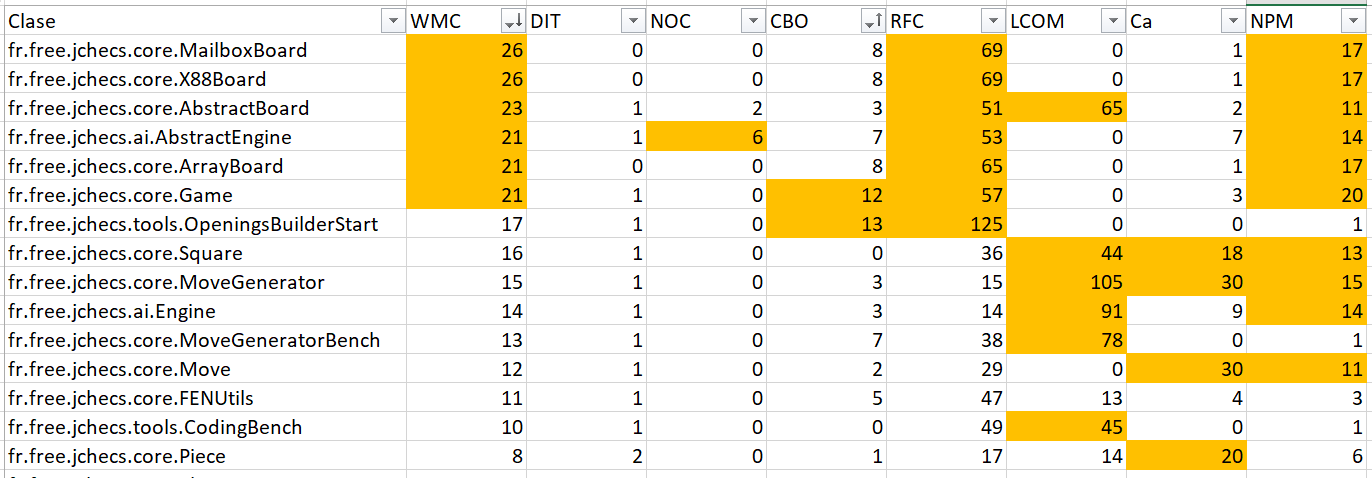
* Número de métodos (Functions en JavaNCSS, WMC en ckjm, Methods en DF)
* Número de líneas de código (NCSS en JavaNCSS, SLOC en DF)
* Número de comentarios: (Javadocs en JavaNCSS).
* Profundidad de herencia (DIT en ckjm)
* Número de hijos (NOC en ckjm)
* Acoplamiento entre clases (CBO en ckjm)
* Respuesta (RFC en ckjm)
* Falta de cohesion en métodos (LCOM enckjm)
* Afferent coupling (Ca en ckjm)
* Número de métodos públicos (NPM en ckjm)
* Inbound Intra-Class Method Dependencies (IICM en DF)
* Outbound Intra-Class Feature Dependencies (OICF en DF)
* Inbound Intra-Package Method Dependencies (IIPM en DF)
* Outbound Intra-Package Feature Dependencies (OIPF en DF)
* Inbound Extra-Package Method Dependencies (IEPM en DF)
* Outbound Extra-Package Feature Dependencies (OEPF en DF)
* Outbound Intra-Package Class Dependencies (OIPC en DF)
* Outbound Extra-Package Class Dependencies (OEPC en DF)

En el caso de las métricas de DF(Dependency Finder), la propia herramienta nos señala en rojo las clases problemáticas, que habría que marcar también en el Excel.



Habría que fijarse sobre todo en las clases que tienen varias métricas señaladas en rojo. Pero también consultar el resto de métricas de ckjm para decidir cuáles serían las clases más difíciles de mantener.

Las métricas de ckjm serían:



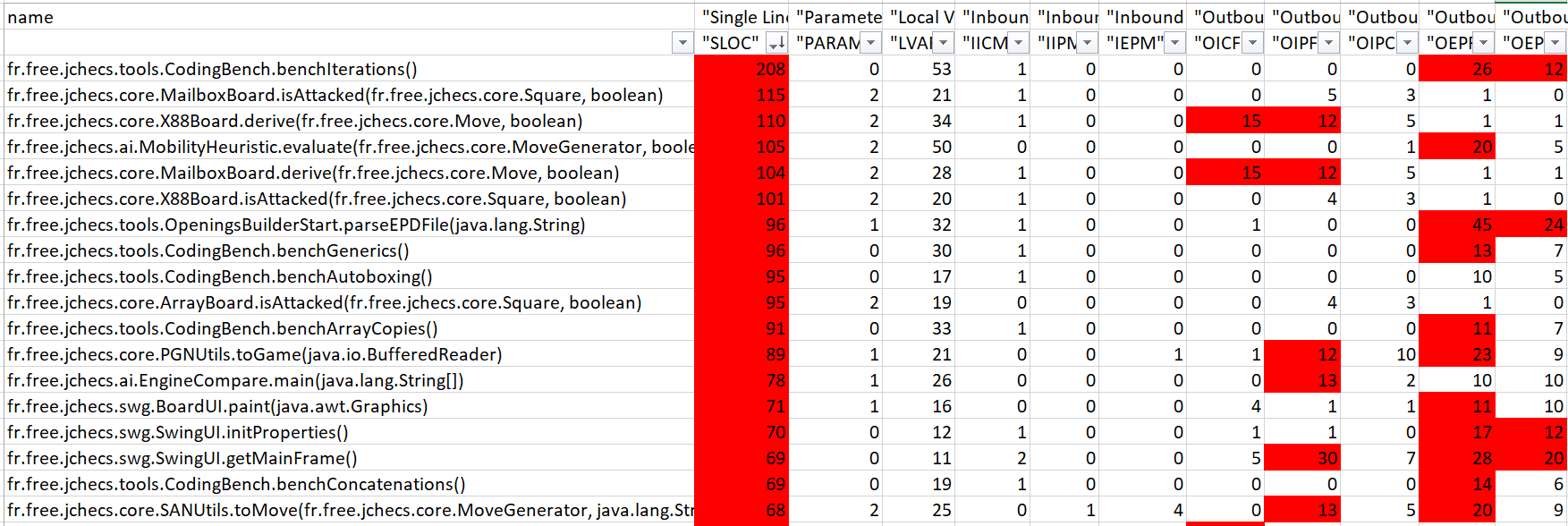
A la vista de los valores, hay dos clases con un número de métodos bastante mayor que el resto, pero las métricas de dependencias no son muy altas. Quizá alguna clase con menos métodos pero más dependencia de otras clases más difícil de controlar. El problema con la herramienta ckjm es que no se pudo analizar el paquete swg, y no se dispone de sus métricas.

### Análisis de métodos

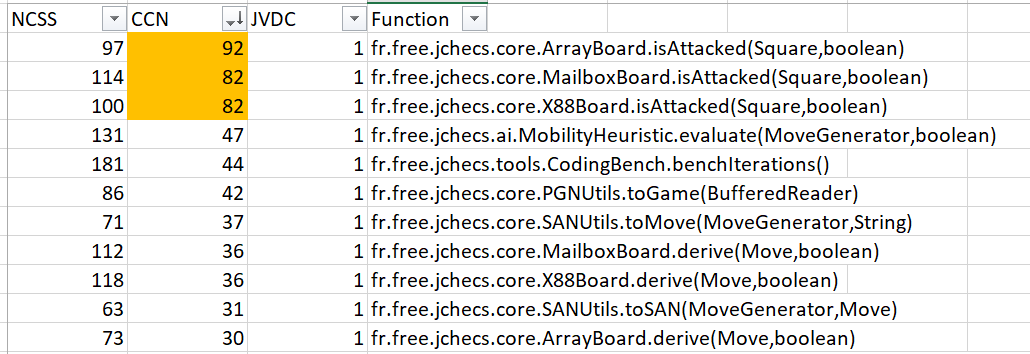
En el caso de los métodos, algunas métricas que se pueden analizar son:

* Número de líneas de código (NCSS en JavaNCSS, SLOC en DF)
* Número de comentarios: (JVDC en JavaNCSS).
* Complejidad ciclomática (CCN en JavaNCSS)
* Número de variables locales (LVAR en DF)
* Número de parámetros (PARAM en DF)
* Inbound Intra-Class Method Dependencies (IICM en DF)
* Outbound Intra-Class Feature Dependencies (OICF en DF)
* Inbound Intra-Package Method Dependencies (IIPM en DF)
* Outbound Intra-Package Feature Dependencies (OIPF en DF)
* Inbound Extra-Package Method Dependencies (IEPM en DF)
* Outbound Extra-Package Feature Dependencies (OEPF en DF)
* Outbound Intra-Package Class Dependencies (OIPC en DF)
* Outbound Extra-Package Class Dependencies (OEPC en DF)

En el caso de las métricas de DF (Dependency Finder), la propia herramienta nos señala en rojo las clases problemáticas, que habría que marcar también en el Excel.



A partir de estos valores y revisando y justificando los valores del resto de métricas de ckjm hay que decidir cuáles serían los métodos más difíciles de mantener.



En este caso, si nos fijamos en la complejidad ciclomática, hay claramente tres métodos que destacan sobre el resto, con una complejidad muy alta y ademán no incluyen comentarios. Sin embargo en las métricas de dependencias no tienen valores altos.

El problema con la herramienta ckjm es que no se pudo analizar el paquete swg, y no se dispone de sus métricas a nivel de métodos.

Como conclusión, parece que las clases MailboxBoard, X88Board y ArrayBoard del paquete core pueden ser las más difíciles de mantener, pues además de ser las que más métodos tienen, entre dichos métodos se encuentran los de mayor complejidad ciclomática. Pero también la clase BoardUI tiene muchos métodos y posibles problemas de dependencia, no se dispone de su información completa, pues no pudo ser analizada por ckjm. A nivel de paquete, está claro que el que se prevé más complejo para mantener es “core” pues, además de tener clases complicadas, tiene gran dependencia, como lo demostraba las métricas de inestabilidad y distancia normalizada.