Deep: Una herramienta para medir dependencias Java

Martín Agüero
Tesista de Maestría en Ingeniería de Software - UNLP
aguero.martin@gmail.com

Luciana Ballejos
CIDISI – Centro de I+D en Ingeniería en Sistemas de Información – FRSF, UTN
lballejos@santafe-conicet.gov.ar

Claudia Pons
CAETI – Centro de Altos Estudios en Tecnología Informática – UAI
claudia.pons@uai.edu.ar

Abstract

En este trabajo se realiza una revisión general acerca de las soluciones disponibles para la gestión de dependencias. Con el fin de medir el nivel de utilización del software Java, se evaluaron 5 herramientas específicas que fueron descartadas y se explican los motivos. Se desarrolló una nueva herramienta denominada Deep que permite medir la tasa de utilización de dependencias entre 2 archivos Jar y se validó su precisión mediante un análisis comparativo de resultados. Con Deep se midieron 7 productos de software de diversa aplicación. Los resultados del estudio confirman que, al menos en esta muestra, se está utilizando menos del 10% del total de recursos disponibles en sus dependencias. Por último se propone un modelo de distribución de software Java más granular y eficiente.

Palabras clave: dependencia de software, java, biblioteca de software, repositorio, código abierto, métricas de software, apache maven.

1. Introducción

Estabilizar de forma manual una plataforma para desarrollar software puede ser una tarea muy tediosa y, en ciertos casos, hasta frustrante [1]. Por otro lado, es un hecho que la complejidad y tamaño del software se ha incrementado de manera significativa. El hardware de mayores prestaciones promueve que las aplicaciones sean cada vez más sofisticadas e interconectadas entre si [2]. Estos requisitos del mercado empujan a la comunidad e industria a desarrollar software multiprestaciones donde la reutilización a través de librerías[1] es un factor clave de éxito, ya sea por calidad probada o integración inmediata de una prestación. Estas bibliotecas son las dependencias de un proyecto de software, que deben cumplir requisitos tales como disponibilidad directa e indirecta, versión y otros [3]. En el software Java estas dependencias son liberadas empaquetadas en archivos de extensión Jar (Java Archive). Los archivos Jar son un conjunto de clases compiladas (bytecode) y agrupadas en un archivo de tipo Zip pero con extensión Jar [4]. El Class Loader es el encargado de obtener ese bytecode y cargarlo en la Máquina Virtual de Java (JVM) [5].

En la actualidad, tanto la academia como la industria han adoptado el uso de herramientas específicas de soporte a la gestión de dependencias como es el caso de Apache Maven y Apache Ivy o Gradle para el software Java [6]. El modelo que implementan estas herramientas se basa en la descarga de todas las dependencias a un repositorio local. Las tres aplicaciones obtienen las bibliotecas de repositorios públicos como son The Central Repository, ibiblio o MVN Repository. Cada vez que un proyecto requiere de alguna dependencia, estas herramientas primero verifican su disponibilidad en el repositorio local y en caso de no encontrarla, solicitan una copia completa al repositorio remoto. Durante este proceso también se comprueba versión y presencia de dependencias transitivas.

Cada una de estas bibliotecas, también llamadas binarios, proveen un conjunto de funcionalidad, que en la mayoría de los casos, sólo se emplea una mínima parte del total disponible [7]. Esta subutilización de recursos podría indicar que el modelo de descarga local completa de cada Jar es una práctica poco eficiente y desactualizada, contemplando que Maven fue una solución ideada en el año 2002 cuando recién se comenzaba a hablar de Web Services [8] y el cómputo en la nube todavía no era más que un símbolo en diagramas de diseño de redes [9]. Asimismo también puede considerarse que el modelo de distribución por paquetes fue heredado del empleado para la distribución de los sistemas operativos [10].

Este trabajo tiene como objetivo estudiar el grado de utilización de dependencias del software Java. Para ello se evaluarán cinco herramientas idóneas: JDepend [11],

[1] Es una errónea traducción de la palabra 'libraries', la correcta es bibliotecas

Google Codepro Analytix [12], STAN [13], CDA [14] y Jdeps [15]. Se explicarán los motivos por los que se llegó a la conclusión de desarrollar una nueva herramienta que permita medir la tasa de recursos utilizados respecto del total disponible. Con este software se medirán una serie de proyectos open source Java de importante magnitud, con características heterogéneas entre sí y muy difundidos en la comunidad e industria de software Java.

A continuación, en la sección 2 se evalúan cinco herramientas para el análisis de dependencias y explica por qué fueron descartadas. Luego, la sección 3 expone las características principales de la herramienta desarrollada para este estudio. La sección 4 desarrolla una prueba comparativa. En la sección 5 se presentan mediciones de dependencias de software Java. Finalmente, la sección 6 presenta las conclusiones y trabajos futuros.

2. Herramientas para análisis de dependencias

El principal objetivo de este trabajo consiste en obtener una medida del nivel acoplamiento más frecuente entre el software Java y sus dependencias. Para ello, en una primera instancia se evaluó utilizar una herramienta de terceros. Se probaron Google Codepro Analytix, STAN, JDepend, Jdeps y CDA. A continuación un resumen de los resultados obtenidos.

CodePro Analytix: es un conjunto de herramientas de soporte a la calidad del software integradas a Eclipse. Cuenta con una importante variedad de características entre las cuales se destacan: métricas de software, análisis de código muerto, generación de casos de prueba unitarios y otros.

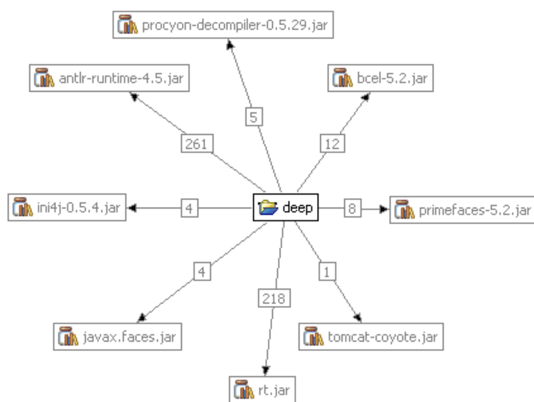
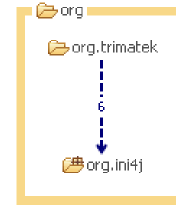


Figura 1 Vista Dependencies de CodePro Analytix

Para este estudio se evaluó la funcionalidad Análisis de dependencias, llegando a las siguientes conclusiones: Cada referencia es contada individualmente, no son únicas. Es decir, si una clase tiene 2 referencias a una misma clase externa (biblioteca), son contadas como 2 referencias. En el informe detallado, los resultados son globales, en relación a todas las dependencias y no una en particular.

STAN: Esta herramienta, que también está basada en Eclipse, ofrece una serie de vistas con información relacionada a las dependencias de un proyecto Java, como por ejemplo: distancia [16], métricas, composición, acoplamiento y otros. Desde la vista Dependencias, STAN (Structure Analysis for Java) muestra un gráfico donde un valor numérico mide el peso de la fortaleza de la dependencia (ver Figura 2).



Source	-->	Target
.trimatek.deep.Deep	references	.ini4j.BasicProfile
.trimatek.deep.Deep	references	.ini4j.InvalidFileFormat
.trimatek.deep.Deep	contains	.ini4j.Wini
.trimatek.deep.LoadTargetProfile	references	.ini4j.BasicProfile
.trimatek.deep.LoadTargetProfile	references	.ini4j.Wini
.trimatek.deep.TestTree	references	.ini4j.Wini

Figura 2 Medición de "peso" con STAN

No obstante, y al igual que CodePro Analytix, cada referencia es contada en forma individual, el peso que le asigna varía si un recurso es invocado más de una vez, en otras palabras, el resultado no es único (unique). Otra desventaja es que no es software open source.

JDepend: Es otra herramienta para analizar las dependencias de software Java, se puede ejecutar como aplicación standalone o con un wrapper desde Eclipse.

Depends upon - efferent dependencies

Package	CC(concr.cl.)	AC(abstr.cl.)	Ca(aff.)
com.strobel.assembler.metadata	0	0	1
com.strobel.decompiler	0	0	1
org.antlr.v4.runtime	0	0	3
org.antlr.v4.runtime.atn	0	0	1
org.antlr.v4.runtime.dfa	0	0	1
org.antlr.v4.runtime.tree	0	0	2
org.apache.bcel.classfile	0	0	2
org.apache.commons.collections4	0	0	1
org.apache.commons.collections4.map	0	0	1
org.apache.tomcat.util.http.fileupload	0	0	1
org.ini4j	0	0	1
org.primefaces.model	0	0	3
org.trimatek.deep.lexer	105	1	2
org.trimatek.deep.model	9	0	4
org.trimatek.deep.service	7	0	4

Figura 3 Análisis con JDepend

De las pruebas realizadas (ver Figura 3), se llegó a las siguientes conclusiones: El análisis que realiza no es relativo a una dependencia en particular, sino que evalúa

a todas las dependencias del proyecto. Además, los resultados son por paquete, no muestra un cálculo general.

Jdeps: Es una herramienta incluida en el SDK de Java que permite visualizar desde la consola la relación de un Jar con todas sus dependencias.

```
digraph "deep.jar" {
   "org.trimatek.deep.lexer"   -> "org.antlr.v4.runtime";
   "org.trimatek.deep.lexer"   -> "org.antlr.v4.runtime.atn";
   "org.trimatek.deep.lexer"   -> "org.antlr.v4.runtime.dfa";
   "org.trimatek.deep.lexer"   -> "org.antlr.v4.runtime.tree";
   "org.trimatek.deep.model"   -> "java.lang";
   "org.trimatek.deep.model"   -> "java.text";
   "org.trimatek.deep.model"   -> "java.util";
   "org.trimatek.deep.service"   -> "com.strobel.decompiler";
   "org.trimatek.deep.service"   -> "java.io";
```

Figura 4 Salida de Jdeps

Como se ve en el ejemplo de la Figura 4, esta herramienta no proporciona resultados cuantitativos por lo que también fue descartada para este estudio.

CDA: De forma similar a Jdeps, Class Dependecy Analysis (CDA), visualiza la relación entre un Jar y otros. Tampoco ofrece resultados cuantitativos, como se ve en la Figura 5, es una herramienta de visualización de dependencias.

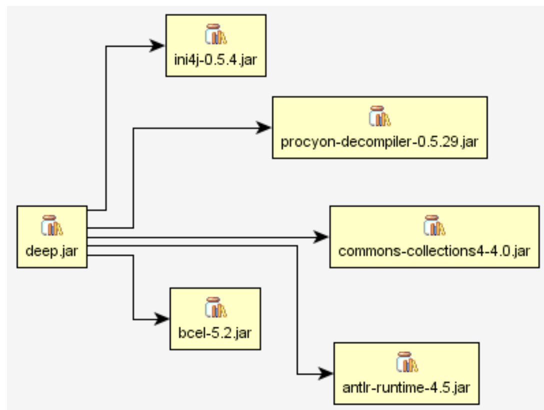


Figura 5 Gráfico de dependencias de CDA

También muestra un listado por clases o paquetes pero, como Jdeps, no genera resultados cuantitativos.

Dado que ninguna de las aplicaciones relevadas cumplía con el requisito de medir referencias únicas, se decidió desarrollar una herramienta específica. El proyecto está publicado como open source, se denomina Deep y ejecuta un análisis similar al del trabajo de Wang y otros [7] (que analiza dependencias a nivel módulo y sólo analiza código fuente en C++) para calcular una métrica basada en el trabajo de Martin [16]. En una misma sesión analiza dos archivos Jar y establece una tasa de dependencia entre sí.

3. Deep

En una primera versión se liberó como Jar para ejecutar por consola de comandos, actualmente hay disponible una interfaz web en la dirección: http://trimatek.org/deep

Internamente, la aplicación realiza las siguientes acciones:

1. Identifica las clases públicas (incluyendo las abstractas y las interfaces), miembros (variables y métodos) del Jar objetivo (la biblioteca).
2. Busca referencias a esas clases/miembros en el Jar origen y muestra un resultado preliminar (Library Quick Survey).
3. Luego genera una visualización jerárquica de las dependencias a través de un árbol de dependencias (Dependency Tree).
4. Por último, calcula la tasa de dependencia y muestra los resultados (Analysis Results).

En la Figura 6 se muestra el resultado del análisis entre drools-core-6.2.0.Final.jar y xstream-1.4.7.jar

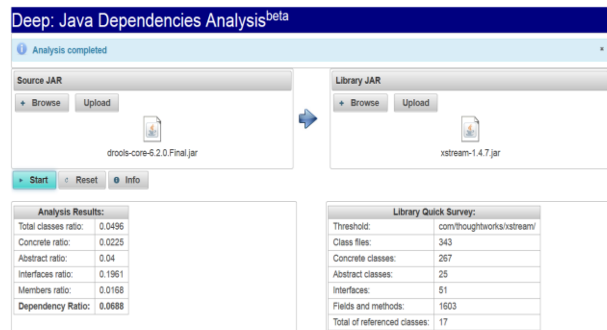


Figura 6 Aplicación web Deep

Para usar la herramienta el usuario debe subir ambos Jars, luego definir un umbral de análisis (Threshold) para el contenido del Jar objetivo y por último presionar el botón Start. Una vez finalizado el proceso, mostrará una serie de resultados que se explican a continuación.

3.1. Tasa de dependencia

A fin de cuantificar el grado de dependencia de un Jar hacia otro, se definió una métrica cuyo resultado se obtiene de los siguientes resultados parciales. De forma similar a la métrica "Inestabilidad" propuesta por Robert Martín en su trabajo titulado "OO Design Quality Metrics, An Analysis of Dependencies" [16], la herramienta Deep cuenta la cantidad de referencias a entidades externas (el Jar objetivo) y calcula una proporción. Siendo:

S: el JAR origen.

T: el JAR objetivo.

Rc: Clases concretas referenciadas en S.
Tc: Total de clases concretas disponibles en T.
Ra: Clases abstractas referenciadas en S.
Ta: Total de clases abstractas disponibles en T.
Ri: Interfaces referenciadas en S.
Ti: Total de interfaces disponibles en T.
Rm: Miembros referenciados en S.
Tm: Total de miembros disponibles en T.

Por último, la herramienta calcula la tasa de dependencia de la siguiente forma:

$$\text{Tasa de dependencia} = \frac{\frac{Rc}{Tc} + \frac{Ra}{Ta} + \frac{Ri}{Ti} + \frac{Rm}{Tm}}{4}$$

Con un resultado cercano a 1 implica una muy alta utilización del total de recursos públicos disponibles y cercado a 0 significa mínima presencia de referencias al Jar objetivo. En resumen, es un promedio de las proporciones entre los recursos referenciados y los disponibles.

3.2. Árbol de dependencias

Deep también genera una visualización en forma de árbol, donde se grafican las relaciones entre las clases del proyecto origen (S) y objetivo (T). Esta representación permite explorar las clases que están relacionadas y en el extremo derecho muestra los métodos o campos referenciados. Como se ve en la Figura 7, la clase Builder del Jar objetivo es referenciada por el Jar origen e invoca los métodos clear, mergeFrom y mergeUnknownFields. También se aprecia que Drools accede al campo EMPTY de la clase ByteString de Protocol buffers.

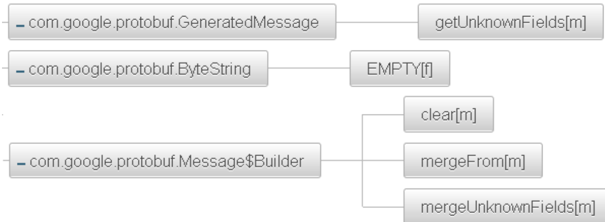


Figura 7 Árbol de dependencias Drools -> Protobuf

Para el caso del análisis entre drools-core-6.2.0.Final.jar y slf4j-api-1.7.2.jar, la Figura 8 muestra una vista parcial de las relaciones, donde en el extremo izquierdo está el Jar de Drools, la siguiente capa corresponde a las clases de Drools que hacen referencia a clases de Slf4J y en el lado derecho los métodos y campos referenciados.

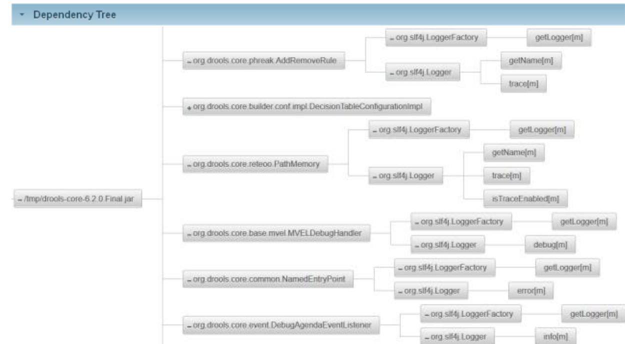


Figura 8 Vista parcial de Drools core -> Slf4J

Junto con la vista jerárquica, la herramienta muestra un cuadro resumen del árbol, un relevamiento de la biblioteca y el resultado del análisis entre drools-core-6.2.0.Final.jar y slf4j-api-1.7.2.jar (ver Figura 9).

Library Quick Survey:	
Threshold:	org/slf4j/
Class files:	22
Concrete classes:	13
Abstract classes:	1
Interfaces:	8
Fields and methods:	200
Total of referenced classes:	2

Analysis Results:	
Total classes ratio:	0.0909
Concrete ratio:	0.0769
Abstract ratio:	0
Interfaces ratio:	0.125
Members ratio:	0.2
Dependency Ratio:	**0.1005**

Figura 9 Cuadros de relevamiento de biblioteca y resultado del análisis

Por último en la Tabla 1 se explica brevemente la función de cada dependencia de Deep.

Tabla 1 Principales dependencias de Deep

Dependencia	Uso
Ini4J [17]	Leer configuración desde archivo de texto (consola)
Apache Bcel [18]	Relevar los recursos públicos disponibles en el Jar objetivo (T)
Procyon Decompiler [19]	Decompilar las clases del archivo Jar origen (S)
ANTLR [20]	Separar en tokens las clases decompiladas para identificar los recursos del objetivo (T) en el fuente del origen (S)
PrimeFaces [21]	Conjunto de componentes para implementar la Interfaz gráfica web.

4. Pruebas

Para validar la precisión de Deep, se analizaron las dependencias de la misma herramienta. Comparando los resultados con CodePro Analytix, STAN y un relevamiento manual del código fuente. Los resultados de

la Tabla 2 corresponden a la cantidad de clases (del Jar objetivo) que son referenciadas por las clases de Deep (Jar origen):

Tabla 2 Mediciones comparadas

	Deep	CodePro	STAN	manual
Ini4j	2	3	5	2
BCEL	10	12	15	9
Collections	2	0	7	2
Procyon	7	5	6	5
ANTLR-rt	32	261	271	21
PrimeFaces	3	8	no	3

(Ver resultados graficados en la Figura 10)

Ini4j: STAN mide 5 referencias porque no son únicas, es decir, si está repetida en varias clases del software de origen, con esta herramienta son contadas nuevamente y además incluye la clase padre de org.ini4j.Winini.

BCEL: Hay diferencia de una clase con la revisión manual porque Deep detectó la clase org.apache.bcel.classfile.Constant que es parte del bytecode pero no del fuente.

Collections: Al igual que ocurre con la medición de Ini4J, STAN vuelve a contar las repeticiones. Por otro lado CodePro no detectó dependencia, lo cual podría indicar un error en la precisión de esta herramienta.

Procyon: Hay diferencia de 2 respecto de la revisión manual porque Deep releva a las clases com.strobel.Decompiler y com.strobel.assembler.metadata.ITypeLoader que no son parte del fuente pero si del bytecode.

ANTLR-rt: La diferencia de 11 respecto del conteo manual se da porque Deep detecta clases anidadas que no son parte del fuente pero si del bytecode.

PrimeFaces: Para el caso del análisis con STAN, no hay resultados porque la versión de uso público posee un límite de hasta 500 clases, y el Jar de PrimeFaces 5.2 supera ese número. CodePro midió 8 porque al igual que en los casos anteriores, vuelve a contar referencias ya contabilizadas.

En el gráfico de la Figura 10 (fuente de datos: Tabla 2) se destacan los resultados de Deep y la revisión manual del código fuente.

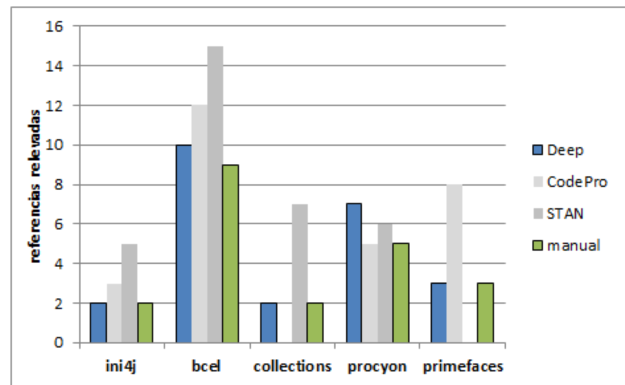


Figura 10 Resultados comparados[2]

En los casos donde no coincide el número de referencias (Bcel y Procyon), se comprobó que la diferencia se da porque el bytecode incluye las clases padre del Jar objetivo. La diferencia entre el conteo manual, CodePro y STAN es mayor dado que estas 2 herramientas no cuentan como únicas las referencias, es decir, si existe más de una clase que invoca a un mismo recurso de la biblioteca, se vuelve a contar esa referencia. En cambio Deep sólo cuenta las referencias únicas.

5. Mediciones con Deep

Se seleccionó un conjunto de productos de software Java muy difundidos y con características heterogéneas entre sí. A continuación una breve descripción de cada uno:

Spring [22]: Es un framework orientado a simplificar el desarrollo de aplicaciones de uso empresarial. Abarca desde interfaces gráficas, seguridad, comunicaciones, servicios web hasta mensajería.

Drools [23]: Es un sistema de gestión de reglas de negocio (BRMS) que provee un núcleo denominado motor de reglas de negocio (BRE) y un módulo compilador de reglas.

Hibernate [24]: Es conjunto de herramientas que facilita la persistencia de objetos en bases de datos relaciones. A su vez también implementa la especificación Java Persistence API (API).

SymmetricDS [25]: Es una solución de sincronización automática de bases de datos que también realiza transformaciones en ambientes heterogéneos.

DbVisualizer [26]: Es un cliente de visualización gráfica de bases de datos. Permite navegar entre las tablas, crear nuevas entidades o realizar consultas a través de SQL. Se ejecuta en ambientes Linux, Windows o Mac OS.

También se sumó al análisis el mismo proyecto Deep. Cabe destacar que fueron elegidos por tratarse de

[2] En el gráfico de la Figura 10 se omitieron los resultados de ANTLR-rt dado que alteraba demasiado la escala del eje vertical.

software de probado prestigio en la industria de software y de variada aplicación.

En la Tabla 3 se observa el resultado de la medición individual de cada Jar de origen (S) con cuatro dependencias (T) elegidas aleatoriamente.

Tabla 3 Medición de tasa de dependencia

JAR Origen (S)	JAR Objetivo (T)	Tasa de dependencia	Promedio
spring-2.0.7.jar	log4j-1.2.14.jar	0,0122	0,03172
	standard-1.1.2.jar	0,0375	
	cglib-2.1.jar	0,0714	
	jstl-1.5.0.jar	0,0375	
drools-core-6.2.0.jar	protobuf-2.5.0.jar	**0,3292**	0,12760
	xstream-1.4.7.jar	0,0688	
	slf4j-api-1.7.2.jar	0,1005	
	comm-codec1.4.jar	0,0119	
hibernate-core4.3.jar	javassist-3.18.jar	0,1196	0,18902
	dom4j-1.6.1.jar	0,1603	
	antlr-2.7.7.jar	0,1548	
	jpa-2.1-api.jar	**0,3214**	
symmetric-3.7.19.jar	comm-codec1.3.jar	0,0177	0,051175
	comm-coll-3.2.jar	0,0015	
	comm-io-2.4.jar	0,1671	
	log4j-1.2.17.jar	0,0184	
dbvisualizer-9.2.8.jar	synthetica.jar	0,0013	0,056275
	jdom-2.0.jar	0,1610	
	icepdf-core-4.3.jar	0,0240	
	dom4j-1.6.jar	0,0388	
drools-compiler-6.2.0.jar	antlr-runt-3.5.jar	**0,3160**	0,096975
	xstream-1.4.7.jar	0,0336	
	slf4j-api-1.7.2.jar	0,0842	
	mvel2-2.2.4.jar	0,0719	
deep-nodep-1.01.jar	antl-rt-4.5.1.jar	0,2243	0,067775
	bcel-5.2.jar	0,0222	
	procyon-dec0.5.jar	0,0111	
	ini4j-0.5.4.jar	0,0135	

Si bien se observan casos particulares como Protocol Buffers (protobuf) para Drools core y la API de JPA para Hibernate core donde también se da una tasa de dependencia significativamente superior, el promedio general (0,08864) no llega a alcanzar la décima parte. A priori, se puede afirmar que en esta muestra, en promedio, se está utilizando menos del 10% de los recursos disponibles en las bibliotecas.

Otro caso que también se midió es la biblioteca ANTLR Runtime 3.5 para Drools Compiler 6.2 donde el análisis entrega un resultado de 0.3160 lo cual es esperable, dado que el Parsing es una función central del módulo compilador de Drools.

6. Conclusiones y trabajos futuros

Integrar software de terceros, ya sea por disponibilidad inmediata de prestaciones o calidad probada, es una práctica cada vez más habitual en el desarrollo de software. En la comunidad open source esto se potencia dada la masividad de bibliotecas liberadas bajo licencia de uso sin restricciones. A consecuencia, ocurre que estos proyectos sólo emplean una mínima porción de funcionalidad disponible en las bibliotecas. Para cuantificar esta situación, en principio, se probó evaluando una serie de herramientas de análisis de dependencias sin obtener resultados positivos dado que no cumplieron con los requisitos.

A raíz de esto, se desarrolló una herramienta específica para medir la tasa de dependencia entre 2 archivos Jar. Con el fin de evaluar su precisión, se realizó un estudio comparativo que confirmó la validez de los resultados. Finalmente se realizó un análisis cuantitativo para obtener una medida de tasa de dependencia entre un conjunto de productos de software Java y sus dependencias. Este análisis arrojó una proporción inferior al 0.1 en una muestra de 7 productos de diversa aplicación.

En base al resultado obtenido, es válido afirmar que la tasa de utilización de dependencias para el software creado con Java, por lo general, es habitualmente baja. Se podría suponer que el modelo de distribución por bibliotecas no sería el apropiado, dado que es muy ineficiente y que, sin contar con una herramienta de gestión, se vuelve una actividad muy tediosa para el desarrollador. Además que la solución de replicación de dependencias a un repositorio local fue ideada cuando el modelo de organización por bibliotecas era muy aceptado y todavía gran parte del software se distribuía en medios físicos como el CD-ROM. Por lo tanto, se considera conveniente evolucionar el modo de acceso a dependencias Java por uno más preciso, donde una entidad intermedia seleccione la biblioteca y entregue el bytecode puntual que solicita el Class Loader.

La siguiente etapa de este proyecto tiene como objetivo diseñar e implementar un servicio intermediario entre el ambiente de desarrollo y los repositorios de bibliotecas. Se planea crear un artefacto de software que atienda a solicitudes de dependencias de forma centralizada y eficiente.

7. Referencias

[1] Ossher, J., Bajracharya, S., Lopes, C., "Automated *Dependency Resolution for Open Source Software". 7th IEEE Working Conference on Mining Software Repositories*. Cape Town, South Africa. 2010. pp. 130-140

[2] Wnuk, K., Regnell, B., Berenbach, B., "Scaling Up Requirements Engineering, Exploring the Challenges of Increasing Size and Complexity in Market-Driven Software Development" *Proceedings of the 17th international working conference on Requirements engineering: foundation for software quality*. Springer-Verlag. Berlin. Germany. 2011

[3] Zimmermann, T., Nagappan, N., Herig, K., Premraj R., Williams, L., "An Empirical Study on the Relation between Dependency Neighborhoods and Failures" 4th IEEE International Conference on Software Testing, Verification and Validation. 2011

[4] The Java Tutorials, "Packaging Programs in JAR Files", 2015 [En línea]. Disponible:

https://docs.oracle.com/javase/tutorial/deployment/jar/. [Accedido: 14/10/2015].

[5] Liang, S., Bracha, G., "Dynamic Class Loading in the Java Virtual Machine". Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. New York. USA. 1998 pp. 36-44

[6] Varanasi, B., Belida, S., “Introducing Maven” Apress. Springer Science+Business Media. New York. USA. 2014

[7] Wang, P., Yang, J., Tan, L., Kroeger, R., Morgenthaler, J. "Generating Precise Dependencies for Large Software". *Proceedings of the IEEE 4th International Workshop on Managing Technical Debt*. Piscataway, USA. 2013. pp. 47-50

[8] XML.com, "A Brief History of SOAP", 2001 [En línea]. Disponible: http://www.xml.com/pub/a/ws/2001/04/04/soap.html. [Accedido: 14/10/2015].

[9] Schmidt, E., Rosenberg, J. "How Google Works". Grand Central Publishing . 2014

[10] Abate, P., Boender, J., "Strong Dependecies betweeen Software Components" *Proceedings of the IEEE 3rd International Symposium on Empirical Software Engineering and Measurement*. Washington, DC, USA. 2009

[11] Clarkware, "JDepend", 2015 [En línea]. Disponible: http://clarkware.com/software/JDepend.html. [Accedido: 14/10/2015].

[12] Google, "CodePro Analytix", 2015 [En línea]. Disponible: https://developers.google.com/java-dev-tools/codepro/. [Accedido: 14/10/2015].

[13] Odysseus Software GmbH, "STAN", 2015 [En línea]. Disponible: http://stan4j.com/. [Accedido: 14/10/2015].

[14] Duchrow, M., "Class Dependency Analyzer", 2015 [En línea]. Disponible: http://www.dependency-analyzer.org/. [Accedido: 14/10/2015].

[15] Oracle, "jdeps", 2015. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html. [Accedido: 14/10/2015].

[16] Martin, R., "OO Design Quality Metrics An Analisis of Dependencies". Object Mentor. 1994

[17] Ini4J, "Ini4J", 2015. [En línea]. Disponible: http://ini4j.sourceforge.net. [Accedido: 15/10/2015].

[18] Apache Commons, "BCEL", 2015. [En línea]. Disponible: https://commons.apache.org/proper/commons-bcel/. [Accedido: 15/10/2015].

[19] Strobel, M., "Procyon Java Decompiler", 2015. [En línea]. Disponible: https://bitbucket.org/mstrobel/procyon/. [Accedido: 15/10/2015].

[20] Parr, T., "ANTLR", 2015. [En línea]. Disponible: http://www.antlr.org/. [Accedido: 15/10/2015].

[21] PrimeTek, "PrimeFaces", 2015. [En línea]. Disponible: http://primefaces.org/. [Accedido: 15/10/2015].

[22] Pivotal Software, "Spring Framework", 2015. [En línea]. Disponible: https://spring.io. [Accedido: 15/10/2015].

[23] JBoss Community, "Drools", 2015. [En línea]. Disponible: http://www.drools.org/. [Accedido: 15/10/2015].

[24] JBoss Developer, "Hibernate ORM", 2015. [En línea]. Disponible: http://hibernate.org/. [Accedido: 15/10/2015].

[25] JumpMind, "SymmetricDS", 2015. [En línea]. Disponible: http://www.symmetricds.org/. [Accedido: 15/10/2015].

[26] DBVis Software AB, "DbVisualizer", 2015. [En línea]. Disponible: https://www.dbvis.com/. [Accedido: 15/10/2015].