Congreso: CACIC 2015

Simposio: XI Workshop Ingeniería de Software (WIS)

Título: Resolución más eficiente de dependencias Java

Resolución más eficiente de dependencias Java

Martín Agüero[1], Luciana Ballejos[2]

[1] Tesista de Maestría en Ingeniería de Software – Facultad de Informática, UNLP
aguero.martin@gmail.com
[2] CIDISI – Centro de I+D en Ingeniería en Sistemas de Información – FRSF, UTN
lballejos@santafe-conicet.gov.ar

Resumen. En este trabajo se realiza una revisión general acerca de las características y modos de gestión de dependencias de código abierto para proyectos Java. Asimismo, también se desarrolla un estudio para establecer una tasa de utilización habitual respecto del total de recursos disponibles en cada dependencia. En base a los resultados obtenidos en las mediciones y el análisis de otros trabajos que también abordaron el tema, se propone cambiar la estrategia de sincronización completa de dependencias (repositorio local) por un middleware que interactúe entre el entorno de desarrollo y los repositorios públicos. Se plantea establecer un servicio que resuelva automáticamente los requerimientos de dependencias directas e indirectas y que atienda solicitudes puntuales de bytecode en tiempo de ejecución y de descriptores para la compilación. Para una siguiente etapa, se planea desarrollar el software propuesto a modo de prueba de concepto.

Palabras clave: java, open source, software library, apache maven, software engineering, software dependencies.

1. Introducción

La gestión de dependencias es una actividad central en el desarrollo de software. En la actualidad, el éxito de una plataforma está ligada, en gran parte, a la disponibilidad masiva de librerías[1] publicadas bajo licencia de código abierto [1]. Estos recursos suelen distribuirse desde el website de las comunidades que agrupan y promueven proyectos colaborativos, tales como las fundaciones Apache, Eclipse o Linux. Una de las fortalezas que ofrece el open source es que se trata de un modelo donde el esfuerzo de la comunidad genera un ecosistema que se nutre a sí misma, donde todos los participantes obtienen un beneficio común. A simple vista parece un modelo ideal, no obstante, los artefactos creados por las distintas comunidades padecen de una severa falta de compatibilidad e integración entre ellos [2]. En la mayoría de los casos, los autores ofrecen el código fuente desde un repositorio especializado y público como GitHub, SourceForge o Bitbucket y versiones compiladas desde el propio website de la comunidad. Esto también sucede en repositorios públicos como ibiblio, The Central Repository o MVN Repository.

Para el caso del software compilado con Java, el procedimiento estándar es distribuir los binarios (bytecode) comprimidos en archivos de formato ZIP, pero con

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

extensión JAR[2]. El empaquetado en archivos de extensión JAR no establece como obligatorio la creación de un manifiesto. Su presencia es optativa y puede contener meta-datos como una firma electrónica, control de versiones, declaración de dependencias o el punto de entrada (Main-Class). Para el caso de los bundles OSGi [3], que también son empaquetados en archivos JAR, el manifiesto es obligatorio. Allí se declara el nombre del bundle, identificador, versión, dependencias e interfaces. Esta tecnología es superadora de la especificación Java. No obstante, aún no consiguió posicionarse masivamente entre la comunidad y la industria, posiblemente por no cubrir aspectos como calidad, usabilidad y reusabilidad [4].

A fin de facilitar la gestión y acceso a dependencias para la compilación de proyectos Java, han surgido herramientas como Apache Maven que en la actualidad es el estándar de facto para todo desarrollo de software a gran escala [5]. Maven propone un modelo de descripción genérica del proyecto a través de un archivo POM (Project Object Model) donde se especifican las dependencias, jerarquías, ciclo de vida de la construcción, parámetros de configuración, casos de prueba y otros [6]. Para optimizar la gestión de dependencias, Maven crea un repositorio local donde se copian todos los archivos JAR requeridos por el proyecto. De este modo, todos los proyectos apuntan sus dependencias a la copia única, pudiendo establecer políticas a nivel particular o departamental, donde un grupo de desarrolladores comparte un único repositorio [7].

Esta estrategia de replicar las librerías –inclusive las transitivas– a un repositorio local ha probado ser exitosa, sin embargo, es tecnología que fue pensada 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].

La especificación de la Máquina Virtual de Java [10] define para el tiempo de ejecución al Class Loader como el responsable de cargar el bytecode en la máquina virtual (JVM) [11].

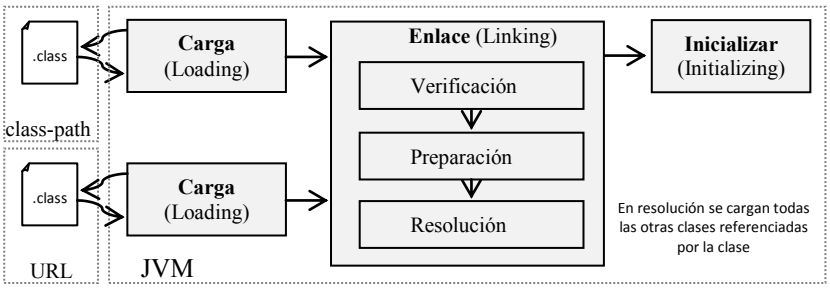


Fig. 1. Proceso de carga de clases en la JVM [12].

Durante este proceso, una instancia de la clase ClassLoader busca en el class-path la clase requerida y la copia a la JVM. Existen implementaciones de ClassLoader de lectura local y a través de la red, lo más habitual es que sea desde una fuente local (Ver Figura 1).

A simple vista este modelo parece ideal, no obstante tanto la industria como la academia han detectado problemas. Uno de ellos es el denominado "Jar Hell" que se da cuando en un mismo class-path conviven clases que comparten un mismo nombre o cuando distintas versiones de una clase deben coexistir en un mismo class-path [4].

[2] https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html

Otro problema que se da con frecuencia es la necesidad de contar también con copias locales de las dependencias transitivas (las dependencias de las dependencias) y con la versión apropiada [13]. Es cierto que Maven (y similares) ofrecen soluciones para los casos citados, mediante perfiles de compilación o automatizando la descarga de las dependencias transitivas [14]. Sin embargo, esta solución implica definir una serie de especificaciones no inherentes al proyecto en sí, además de tener que recuperar del repositorio central tanto la totalidad de las dependencias directas como así también las indirectas.

Este trabajo tiene como objetivo analizar el grado de utilización promedio de estas librerías por parte del software Java y proponer una alternativa al modelo de repositorio local. Por medio de un programa creado específicamente para medir la proporción entre recursos disponibles y empleados, se intentará demostrar que el nivel de referencia a recursos externos, por lo general, no supera la décima parte del total disponible en los archivos JAR. De forma análoga, se puede representar la situación como la de precisar disponer de un libro y por ello solicitar toda una fila de la estantería de la biblioteca. Asimismo en este paper, también se definirá un modelo conceptual que propondrá un servicio intermediario cuya función será la de atender en tiempo real solicitudes de bytecode por parte de la JVM y el compilador. Este sistema ubicará en los repositorios las librerías correspondientes y entregará al cliente sólo las clases solicitadas y sus transitivas. El modelo no está pensado para sustituir la disponibilidad local de dependencias en ambientes de producción, pero sí para ser una alternativa ágil y actual a emplear en instancias de evaluación de tecnología, desarrollo o pruebas.

A continuación, la sección 2 describe la situación actual y otras alternativas propuestas por la academia y la industria en el área. Luego, la sección 3 describe la herramienta desarrollada para el análisis de las dependencias y analiza las características de otras similares que fueron descartadas, además de explicar los motivos. En la sección 4 se presentan mediciones de software libre Java y la sección 5 describe un modelo conceptual de la propuesta. Finalmente, la sección 6 presenta las conclusiones y trabajos futuros propuestos.

2. Antecedentes

Desde la versión 1.2 de la plataforma estándar de Java (J2SE) está disponible la clase URLClassLoader que permite cargar clases Java desde fuentes remotas. Ya en la versión 1.1 de Java, el Class Loader podía obtener clases desde cualquier origen. No obstante, con esta implementación existían problemas de performance, seguridad y permisos de acceso [15].

Aprovechando la característica de carga desde fuentes remotas, Parker y Cleary [15] evaluaron la posibilidad de emplear el protocolo P2P para la carga remota de clases entre JVM. Dos implementaciones de referencia confirman que este paradigma es un enfoque válido para aplicar a sistemas distribuidos. En el trabajo de Ryan y Newmarch [16] estudian distintas técnicas de class loading distribuido y proponen una estructura de descubrimiento y publicación remoto de clases. Las pruebas posicionan a esta técnica como una alternativa viable incluso para entornos wireless.

En el mismo sentido, para ambientes de ejecución de recursos limitados, el trabajo de Petrea y Grigoras [17] presenta una implementación de máquina virtual con

capacidad de carga remota de clases. Mediciones de desempeño verifican que el impacto en la performance es mínimo al comparar el tiempo de respuesta con una máquina virtual CLDC (Connected Limited Device Configuration).

Ossher y otros [1] proponen a Sourcerer para resolver automáticamente las dependencias del software open source a través de un algoritmo de referencia cruzada. Los autores llegan a la conclusión que la resolución automática de dependencias es una opción viable, pero es necesario mejorar la indexación de los meta-datos disponibles en los repositorios.

Por otro lado, el trabajo de Frénot y otros [18] presenta un framework orientado a dispositivos de recursos restringidos, donde se ejecuta OSGi. La solución se basa en servidores de caché remotos que entregan el bytecode de una clase a demanda. En el ambiente local sólo se instalan representaciones de los componentes. Los resultados de las pruebas confirman que, en especial para bundles[3] de mayor tamaño, el tiempo de instalación se reduce considerablemente.

García y otros [2] al estudiar la fragmentación que existe entre los repositorios de open source, proponen un metamodelo para describir los componentes OSGi y un sistema de federación de repositorios. Este trabajo apunta a ofrecer resolución de dependencias por facetas. Wang y otros [19] identifican dos casos puntuales de malas dependencias: las subutilizadas y las inconsistentes. Mediante software ad hoc, los autores detectaron utilización por debajo del 20% de dependencias de terceras partes y dependencias externas en módulos base en un proyecto open source (C++).

El trabajo de Jezek y otros [13] realiza un detallado análisis de los problemas que se dan a consecuencia del alto nivel de reutilización que existe en el software open source. Se identifican los casos denominados como: mediación, reempaquetado, compatibilidad y redundancia de dependencias. Propone un análisis estático de dependencias para verificar todas las interfaces de los componentes a través de un algoritmo de prueba de compatibilidad de tipo. Un estudio empírico dimensiona el problema y advierte sobre la utilización de Maven cuando se renombran las librerías o existen clases duplicadas.

En base a los aportes de los trabajos citados y cuestiones vinculadas para las cuales está pendiente encontrar una mejor solución, a continuación se desarrollará un estudio que intentará establecer un grado de utilización medio de las dependencias en el software open source.

3. Análisis de dependencias

El objetivo de esta tarea consistió en determinar el nivel de acoplamiento entre las librerías JAR. Para ello, en una primera instancia se evaluó utilizar una herramienta de terceros. Se probó con Google Codepro Analytix[4] y también con JDepend[5]. Si bien ambas están muy difundidas tanto en la industria como la academia e implementan las métricas propuestas por Robert Martin [20], se llegó a la conclusión que las siguientes características no eran apropiadas para este estudio:

[3] Conjunto de clases Java empaquetadas como archivo JAR y que posee un archivo manifiesto donde se detallan características como nombre, versión, dependencias, interfaces y otros.

[4] https://developers.google.com/java-dev-tools/codepro/

[5] http://clarkware.com/software/JDepend.html

JDepend:

- El análisis que realiza no es relativo a una dependencia en particular, sino que evalúa a todas las dependencias del proyecto.
- Los resultados son por paquete, no hay un cálculo general.

Google Codepro Analytix:

- La cantidad de referencias a una dependencia no son únicas, es decir, si una clase tiene 2 referencias a una misma clase externa (librería), son contadas como 2 referencias.
- En el informe detallado, los resultados son globales, en relación a todas las dependencias y no una en particular.

Por estos motivos se decidió desarrollar una herramienta específica. El proyecto está publicado como open source, se denomina Deep y ejecuta un análisis similar al de Wang y otros [19] y una métrica basada en el trabajo de Martin [20]. En una misma sesión analiza dos archivos JAR y establece una tasa de dependencia entre sí.

Internamente ejecuta las siguientes tareas:

1. Identifica las clases públicas (incluyendo las abstractas y las interfaces), miembros (variables y métodos) del JAR objetivo (la librería).
2. Busca referencias a esas clases/miembros en el JAR origen y muestra por consola un resultado preliminar.
3. Genera una visualización jerárquica de las dependencias a través de un árbol de dependencias.
4. Por último, calcula la Tasa de Dependencia y muestra los resultados.

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. 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

$$\text{Tasa de dependencia} = \frac{\frac{Rc}{Tc} + \frac{Ra}{Ta} + \frac{Ri}{Ti} + \frac{Rm}{Tm}}{4} \qquad (1)$$

En resumen, es un promedio de las proporciones entre los recursos referenciados y los disponibles. En el repositorio del proyecto[6] se puede bajar la última versión compilada, se explica el modo de uso y se muestran pantallas con salidas por consola.

4. Mediciones

Empleando la herramienta presentada en la sección anterior, se seleccionó un conjunto de productos de software de importante magnitud, con características heterogéneas entre sí[7] y muy difundidos en la comunidad open source Java. También se sumó al análisis el mismo proyecto Deep (sin incluir las dependencias en el mismo JAR). En la Tabla 1 puede observarse el resultado de la medición individual de cada JAR de origen (S) con cuatro dependencias (T) elegidas aleatoriamente.

Tabla 1 – 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-com-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	

[6] https://github.com/martinaguero/deep

[7] Se los considera heterogéneos dado que fueron desarrollados por diferentes empresas / comunidades y poseen objetivos diversos.

Si bien se observan casos particulares como Protocol Buffers (protobuf) para Drools y la API de JPA para Hibernate donde se da una tasa de dependencia significativamente por encima de los demás, 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 las capacidades disponibles en las librerías.

Otro caso que también se midió es la librería ANTLR Runtime 3.5 para Drools Compiler 6.2 donde el análisis entrega un resultado de 0.316 lo cual es esperable, dado que el Parsing es una función central del módulo compilador de Drools.

5. Propuesta

Como respuesta a la situación planteada y con intención de la simplificar y optimizar la gestión de dependencias, se propone el modelo conceptual representado en la Figura 2.

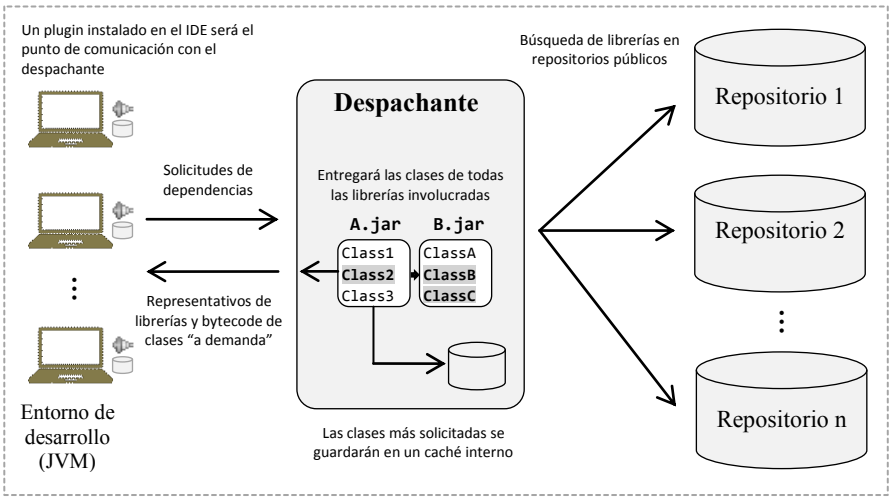


Fig. 2. Intermediario entre entornos de desarrollo (IDE) y repositorios de librerías.

En la Figura 2 las entidades vinculadas por el despachante son: el compilador, la máquina virtual de Java (entorno de desarrollo) y los repositorios de bytecode empaquetado en archivos JAR (librerías).

5.1. Tiempo de compilación

En esta instancia, el intermediario entregará representativos de los JAR requeridos por el cliente, de manera similar a la solución propuesta por Frénot y otros para el framework ROCS [18]. Un representativo es una "sombra" o "cáscara" de las librerías que contiene únicamente los descriptores y firmas de las clases. Todas las dependencias transitivas también serán resueltas automáticamente por el despachante y se enviarán al cliente en forma de representativos.

5.2. Tiempo de ejecución

Durante la ejecución del programa, la JVM obtendrá el bytecode mediante la carga remota de las clases. El despachante será el encargado de entregar al Class Loader el bytecode requerido. Un caché interno guardará en el cliente una copia del bytecode solicitado para que en futuras ejecuciones la carga se realice desde una fuente local.

5.3. Despachante

El despachante se encargará de resolver la clausura de dependencias en función de los datos provistos por el class-path definido en el IDE del cliente. En tiempo de compilación, entregará los representativos de las dependencias directas e indirectas. La resolución de todas las dependencias estará a su cargo y lo realizará de forma automática en base a los datos disponibles en los repositorios respecto a las dependencias de cada librería. La comunicación con el cliente será a través de un plugin específico y estará instalado en el IDE. En tiempo de ejecución responderá a las solicitudes de clases y enviará por red únicamente aquellas involucradas en la ejecución. Un caché interno en el despachante y el plugin permitirá optimizar el tiempo de respuesta para las solicitudes de carga más frecuentes. El contacto del cliente con el despachante será siempre a través de un único punto de entrada y una dirección permanente, más allá de la naturaleza del proyecto en desarrollo y los requisitos de dependencias, será responsabilidad del plugin establecer comunicación con el despachante y lo mismo para el Class Loader.

5.4. Dependencias transitivas

Como se explicó en las secciones anteriores, será responsabilidad del despachante resolver las dependencias de primer nivel y subsiguientes. En la Figura 3 se presenta un caso de resolución encadenada de dependencias.

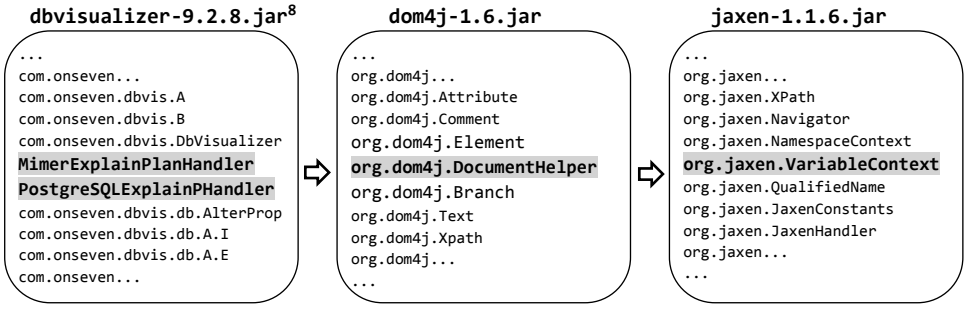


Fig. 3. Resolución de dependencia transitiva.

Como se ve en la Tabla 1, DBVisualizer 9.2 sólo requiere del 0,04 de los recursos públicos de Dom4J que a su vez éste sólo requiere el 0.10 de los recursos públicos de

[8] Los nombres completos de estas clases son `com.onseven.dbvis.db.mimer.MimerExplainPlanHandler` y `com.onseven.dbvis.db.postgresql.PostgreSQLExplainPlanHandler`

Jaxen[9]. El software visualizador de bases de datos referencia a sólo 5 clases de Dom4J de las cuales 3 son interfaces y 2 son concretas, entre ellas está `org.dom4j.DocumentHelper`. A su vez Dom4J hace referencia a 17 clases de Jaxen de las cuales 6 son interfaces, 1 es abstracta y 10 son concretas, una de ellas es `org.jaxen.VariableContext`.

El despachante entregará al cliente los representativos de cada librería para conseguir la compilación y en tiempo de ejecución, sólo las clases requeridas, no el JAR completo, como hubiese ocurrido si no se utilizara la propuesta de este trabajo.

6. Concusiones y trabajos futuros

Inicialmente este trabajo se enfocó en conocer las características de las librerías de programas compilados de dominio público. Se estudiaron las características técnicas y el modo en que son puestas a disposición de la comunidad de desarrolladores open source. También se presentaron las herramientas de software habitualmente empleadas para gestionar las dependencias, haciendo foco en las características que hoy, por diversos motivos, son obsoletas. Asimismo también se hizo un breve repaso por trabajos que han abordado el tema y las alternativas que proponen la academia y la industria.

A fin de obtener datos cuantitativos de la relación entre el software compilado Java, se desarrolló un estudio para establecer una tasa de dependencia habitual entre el software open source y sus dependencias. En una primera instancia se evaluó realizar el análisis utilizando software de terceros pero finalmente se decidió desarrollar una herramienta ad hoc. Los resultados de las mediciones arrojaron una proporción inferior al 0.10 en una muestra de 7 proyectos, llegando a la primera conclusión empírica de este trabajo: se utilizan menos del 10% de los recursos disponibles (clases y miembros públicos) en los archivos JAR.

En base a la situación presentada, se propone reemplazar el vuelco total de dependencias por un servicio intermediario ubicado entre los repositorios de binario open source y los entornos de desarrollo. Se plantea establecer un middleware despachante de representativos de librerías que también atienda solicitudes puntuales de bytecode de clases Java en tiempo de ejecución.

Por último, se presentó un caso de resolución completa de dependencias de primer y segundo nivel, explicando cómo la propuesta obtendría y enviaría al cliente únicamente las clases referenciadas.

La próxima etapa del proyecto tiene planeado desarrollar la especificación completa y detallada de la solución y una implementación de referencia. El objetivo será desarrollar y publicar un servicio en la nube que, en conjunto con un plugin para Eclipse, permita resolver dependencias, compilar y ejecutar software Java. También se planea medir la diferencia de tiempo entre ejecución con dependencias locales y remotas con despachante.

[9] 0,0958 medido con Deep 1.01

7. Referencias

1. Ossher, J., Bajracharya, S., Lopes, C.: Automated Dependency Resolution for Open Source Software. Working Conference on Mining Software Repositories. IEEE (2010)
2. García-Carmona, R., Cuadrado, F., Dueñas, J., Navas, A.: A repository for integration of software artifacts with dependency resolution and federation support. Software and Data Technologies. Springer (2013)
3. OSGi Core Release 6, The OSGi Alliance (2014)
4. Zhou, J., Zhao, D., Ji, Y., Liu, J.: Examining OSGi from an Ideal Enterprise Software Component Model. Software Engineering and Service Sciences. IEEE (2010)
5. InfoQ, http://www.infoq.com/news/2015/03/maven-polyglot
6. McIntosh, S., Adams, B., Hassan, A.: The evolution of Java build systems. Springer Science+Business Media (2011)
7. Massol, V., Van Zyl, J.: Better Builds with Maven. Mergere Library Press (2006)
8. XML, http://www.xml.com/pub/a/ws/2001/04/04/soap.html
9. Schmidt, E., Rosenberg, J.: How Google Works. Grand Central Publishing (2014)
10. The Java Virtual Machine Specification, http://docs.oracle.com/javase/specs/jvms/se8/html/ (2015)
11. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. ACM SIGPLAN Conference (1998)
12. Shankar, L., Burns, S.: Demystifying class loading problems, Part 1: An introduction to class loading and debugging tools. DeveloperWorks. IBM (2005)
13. Jezek, K., Dietrich, J.: On the Use of Static Analysis to Safeward Recursive Dependency Resolution. 40th Euromicro Conference on Software Engineering and Advanced Applications (2014)
14. Maven Transitive Dependencies, https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html
15. Parker, D., Cleary, D.: A P2P Approach to ClassLoading in Java. Agents an Peer-to-Peer Computing. Lecture Notes in Computer Science. Springer-Verlag (2003)
16. Ryan, A., Newmarch, J.: A Dynamic, Discovery Based, Remote Class Loading Structure. Software Engineering and Applications (2003)
17. Petrea, L., Grigoras, D.: Remote Class Loading for Mobile Devices. International Symposium on Parallel and Distributed Computing. IEEE (2007)
18. Frénot, S., Ibrahim, N., Le Mouël, F., Ben Hamida, A.: ROCS: a remotely provisioned OSGi framework for ambient systems. Network Operations and Management Symposium. IEEE (2010)
19. Wang, P., Yang, J., Tan, L., Kroeger, R., Morgenthaler, J.: Generating Precise Dependencies for Large Software. 35th. International Conference on Software Engineering. IEEE (2013)
20. Martin, R.: OO Design Quality Metrics An Analisis of Dependencies. Object Mentor (1994)

Congreso: CONAIISI 2015

Simposio/Área: Ingeniería de Sistemas y de Software

Título: Deep: Una herramienta para medir dependencias Java

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
CIC - Comisión de Investigaciones Científicas
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, en promedio, 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 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 de los productos de software se han incrementado de manera significativa. El hardware de mayores prestaciones promueve que las aplicaciones sean cada vez más sofisticadas e interconectadas entre sí [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 nueva 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, tales como 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 de la 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 también 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

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

se evaluarán cinco herramientas idóneas: JDepend [11], 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 se 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 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.

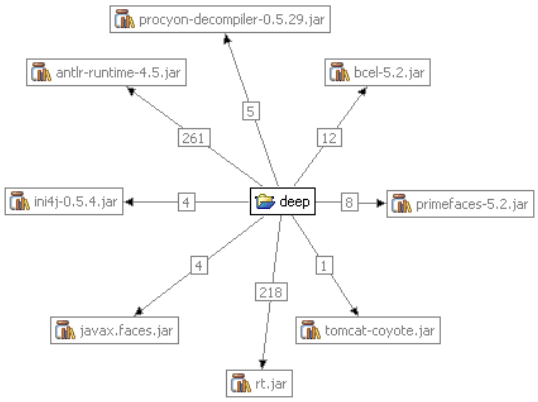
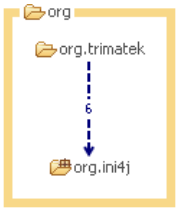


Figura 1. Vista "Dependencies" de CodePro Analytix.

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 a 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. Tal como se ve en la Figura 5, es una herramienta de visualización de dependencias.

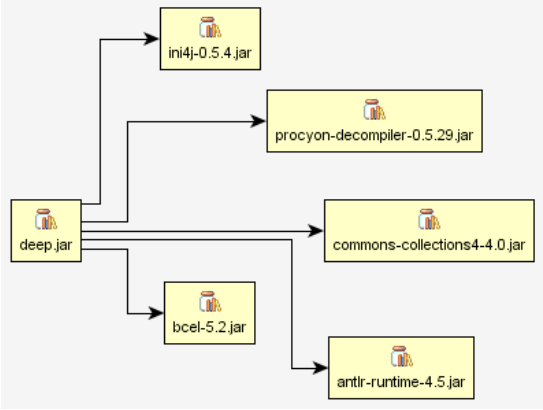


Figura 5. Gráfico de dependencias de CDA.

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 software de código abierto, se denomina Deep [17] y realiza 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).

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

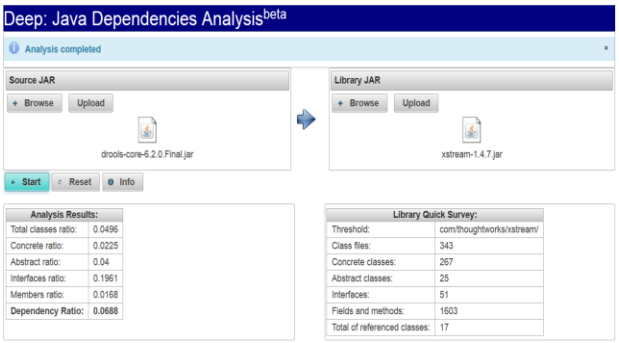


Figura 6. Aplicación web Deep.

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 para conocer la tasa de utilización de recursos disponibles en la biblioteca objetivo (T). Tomando como referencia la métrica "Inestabilidad" propuesta por Martín [16] (Ver Figura 7), la herramienta Deep cuenta la cantidad de referencias a entidades externas (el Jar objetivo) y calcula una proporción.

Ca: Acoplamiento Aferente: El número de clases externas que dependen de clases locales.
Ce: Acoplamiento Eferente: El número de clases locales que dependen de clases externas.
I: Inestabilidad: (Ce ÷ (Ca+Ce)): Métrica de rango [0,1]. Donde I=0 indica máxima estabilidad e I=1 indica máxima inestabilidad.

Figura 7. Métrica Inestabilidad

Una vez relevados los recursos públicos disponibles en la biblioteca y las referencias a esos recursos en el Jar origen (S), se calcula la tasa de dependencia:

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

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.

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

3.2. Árbol de dependencias

Deep también genera una visualización en forma de árbol, donde se grafican las relaciones entre las clases de los binarios 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 8, la clase Message$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.

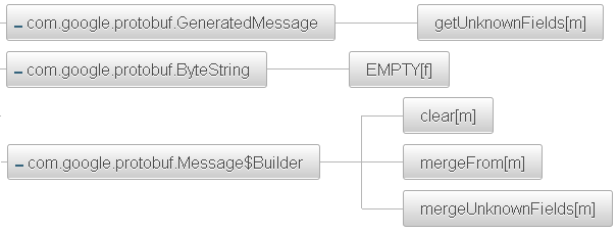


Figura 8. Recursos de Protobuf referenciados por Drools core.

Para el caso del análisis entre drools-core-6.2.0.Final.jar y slf4j-api-1.7.2.jar, la Figura 9 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 de Slf4J referenciados.

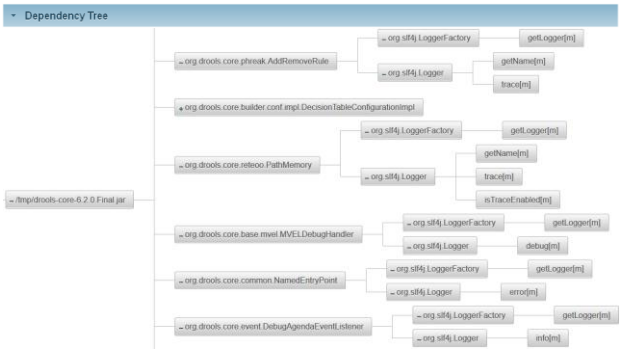


Figura 9. Vista general del árbol de dependencias entre Drools core y Simple Logging Facade for Java.

Junto con la vista jerárquica, la herramienta muestra un cuadro resumen del árbol, un relevamiento de la biblioteca (Library Quick Survey) y el resultado del análisis entre drools-core-6.2.0.Final.jar y slf4j-api-1.7.2.jar (ver Figura 10).

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 10. Cuadros de relevamiento de biblioteca y resultado del análisis.

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

Tabla 1. Principales dependencias de Deep.

Dependencia	Uso
Ini4J [18]	Se emplea para leer la configuración desde un archivo de texto (consola)
Apache Bcel [19]	Relevar los recursos públicos disponibles en el Jar objetivo (T)
Procyon Decompiler [20]	Descompilar las clases del archivo Jar origen (S)
Common Collections [21]	Organizar resultados parciales en un mapa de clave múltple.
ANTLR [22]	Separar en tokens las clases descompiladas para identificar los recursos del objetivo (T) en el fuente del origen (S)
PrimeFaces [23]	Conjunto de componentes visuales 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, además de realizar 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 11)

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: Lo mismo 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 sí 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 sí 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 11 (fuente de datos: Tabla 2) se destacan los resultados de Deep y la revisión manual del código fuente.

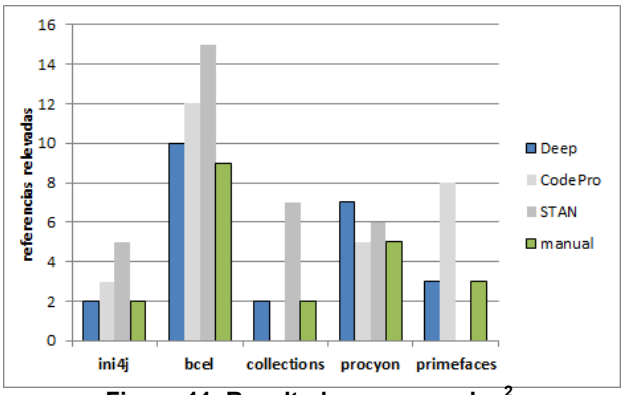


Figura 11. Resultados comparados[2].

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 vs. 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 [24]: Es un framework orientado a simplificar el desarrollo de aplicaciones de uso empresarial. Abarca desde interfaces gráficas, seguridad, comunicaciones, servicios web hasta mensajería.
Drools [25]: 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.

[2] En el gráfico de la Figura 11 se omitieron los resultados de ANTLR-rt dado que alteraba demasiado la escala del eje vertical.

Hibernate [26]: 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 [27]: Es una solución de sincronización automática de bases de datos que también realiza transformaciones en ambientes heterogéneos.
DbVisualizer [28]: 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 software de probado prestigio en la industria 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 y accesibilidad a 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 el requisito de contar referencias no repetidas.

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 llevó a cabo 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 variada 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, debe considerarse 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] Agüero, M., "Deep", 2015 [En línea]. Disponible: https://github.com/martinaguero/deep. [Accedido: 15/10/2015].

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

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

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

[21] Apache Software Foundation, "Common Collections", 2015. [En línea]. Disponible: https://commons.apache.org/proper/commons-collections/. [Accedido: 15/10/2015].

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

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

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

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

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

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

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

Congreso: CONAIISI 2016

Simposio/Área: Ingeniería de Sistemas y de Software

Título: Resolución de dependencias Java a demanda

Resolución de Dependencias Java a Demanda

Martín Agüero
DISI – Departamento de Ingeniería en Sistemas de Información – FRBA, UTN
maguero@frba.utn.edu.ar

Luciana Ballejos
CIDISI – Centro de I+D en Ingeniería en Sistemas de Información – FRSF, UTN
lballejo@frsf.utn.edu.ar

Claudia Pons
CIC - Comisión de Investigaciones Científicas
Facultad de Informática – UNLP
CAETI – Universidad Abierta Iteramericana – UAI
claudia.pons@uai.edu.ar

Abstract

En este trabajo se analizan las características de las herramientas de soporte a la gestión de dependencias de software Java. También se estudia el modo como se referencian las clases entre sí a nivel binario y las particularidades del enlace dinámico de la máquina virtual de Java. Se propone una alternativa al modelo de vuelco completo de dependencias, por un servicio intermediario entre el ambiente de desarrollo y los repositorios de bibliotecas. Con un prototipo se valida la factibilidad de resolver requerimientos de dependencias a demanda, suministrando únicamente las clases referenciadas por el programa de usuario. Esto permite refinar el nivel de granularidad de bibliotecas Java, dando lugar a una resolución más actualizada y eficiente de dependencias.

Palabras Clave: java, biblioteca de software, gestor de paquetes, microservicios, osgi, cómputo en la nube.

1. Introducción

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, Ivy o Gradle para el software Java [1]. El modelo que implementan estas herramientas está basado en la descarga de todas las dependencias a un repositorio local. Estas tres soluciones obtienen las bibliotecas de repositorios públicos, tales como The Central Repository o ibiblio. Cada vez que un proyecto requiere de alguna dependencia, todas estas herramientas primero verifican su disponibilidad en el repositorio local y, en caso de no encontrarla, solicitan una copia completa al repositorio remoto. Durante el proceso también se comprueba versión y presencia de dependencias transitivas.

Cada una de estas bibliotecas, también llamadas *binarios*, proveen un conjunto de funcionalidades de la que, en la mayoría de los casos, sólo se emplea una mínima porción del total disponible [2]. Esta subutilización de recursos podría indicar que el modelo de descarga local completa de cada biblioteca es una práctica poco eficiente y desactualizada. Aún más, Maven es una solución diseñada en el año 2002, cuando recién se empezaba a hablar de Web Services y el cómputo en la nube todavía no era más que un símbolo en diagramas de redes. También puede considerarse que el modelo de distribución por paquetes, que fue heredado de la organización interna de los sistemas operativos [3], posiblemente ya sea obsoleto.

Una de las especificaciones que definen a la tecnología Java establece las características de la implementación del patrón Máquina Virtual [4]. En ese documento se describe el modo en que una Máquina Virtual de Java debe interpretar y ejecutar las instrucciones de código intermedio (bytecode) generado durante la compilación. También especifica que el enlazamiento entre el programa del usuario y los recursos externos (las clases Java) sea a través de referencias simbólicas. Esas referencias luego son traducidas en tiempo de ejecución, mediante la información registrada en el área de constantes (Constants pool) del código intermedio. Esta indirección permitiría postergar la necesidad de contar con la totalidad de esos recursos hasta el momento de ejecución.

Este trabajo presenta y describe el prototipo de un sistema de software que gestiona de manera eficiente y a demanda recursos de bibliotecas Java (Jar). En función del análisis del bytecode, resuelve y solicita a un servicio web, únicamente las clases Java requeridas. Asimismo, el servicio también analiza, resuelve y provee de manera automática las dependencias transitivas requeridas para la ejecución del programa de usuario.

En la sección 2 se presenta una breve descripción del contexto tecnológico desde donde emerge el proyecto. Luego, en la sección 3 se explican las características de las herramientas actuales para gestionar dependencias. La sección 4 introduce los fundamentos conceptuales de la propuesta. En la sección 5 se describen las características

tecnológicas de un prototipo. En la sección 6 se describe un caso de uso con el prototipo y finalmente, la sección 7 presenta las conclusiones.

2. Contexto Tecnológico

El resultado de compilar código fuente Java (extensión java) genera como salida un archivo de extensión *class*[1] por cada clase Java. Un archivo *class* o binario principalmente contiene instrucciones (o bytecode) y una tabla de símbolos (Constant pool) que serán interpretadas por la Máquina Virtual de Java (JVM) en tiempo de ejecución [5]. El Constant pool contiene diferentes tipos de constantes que pueden ser desde literales numéricos hasta referencias a métodos o campos que deberán ser resueltos en tiempo de ejecución.

2.1. Máquina Virtual de Java

La JVM es una máquina de computación abstracta, similar a una máquina de computación real. Posee un conjunto de instrucciones y gestiona varias áreas de memoria en tiempo de ejecución. Su función es la de ejecutar el código Java compilado desde cualquier plataforma de hardware o sistema operativo.

2.2. Binarios Java

Dentro de una biblioteca Java[2], la mínima unidad contenedora de información para la JVM son los archivos binarios. En la tabla de símbolos de un archivo *class* también se definen referencias simbólicas a recursos externos, que deberán estar disponibles en el classpath[3] en tiempo de ejecución. Esos recursos pueden ser clases pertenecientes a otros paquetes del mismo programa de usuario o bibliotecas de terceros.

2.3. Carga de Clases (Class Loading)

Es el mecanismo que provee gran parte de las innovaciones de la plataforma Java, principalmente, la capacidad de instalar componentes de software en tiempo de ejecución. El objetivo de los class loaders es dar soporte a la carga dinámica de clases en la plataforma Java. Son instancias de subclases de la clase java.lang.ClassLoader. Existen dos tipos de class loaders: el bootstrap que es provisto por la JVM y los definidos por el usuario.

A fin de resolver una referencia simbólica a una clase, la JVM primero debe cargar el archivo binario, para finalmente crear el objeto de esa clase [6].

[1] Según la especificación de la JVM, también podría tener otra extensión.
[2] Archivo ZIP con extensión Jar.
[3] Es la variable del sistema operativo que indica a la JVM dónde ubicar clases que no son parte del programa de usuario en ejecución.

2.4. Enlace Dinámico (Dynamic Linking)

Otra característica particular de esta tecnología es que, en lugar de requerir enlazar un programa por completo antes de su ejecución, las clases e interfaces son cargadas a demanda por la JVM. En la mayoría de los ambientes de programación, el enlace se crea en el momento de compilación, y en tiempo de ejecución el sistema carga el recurso completo. En Java, el compilador sólo define referencias simbólicas y es la JVM la que emplea esa información para ubicar y cargar individualmente las clases e interfaces "a demanda". Esto hace el proceso de carga más complejo, pero tiene sus ventajas:

- Inicio más rápido, porque debe cargar menos código.
- En tiempo de ejecución, el programa puede enlazar a la última versión del binario, por más que la versión no estuviera disponible al momento de la compilación [7].

Durante este proceso la JVM realiza una sucesión de actividades encadenadas: verificación, preparación y resolución, para finalmente dar paso a la inicialización de la clase, tal como muestra la Figura 1 [8].

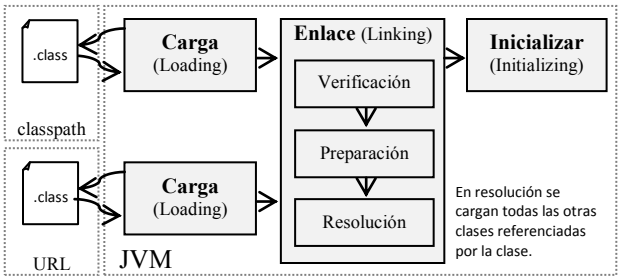


Figura 1. Carga, enlace e inicialización de clases.

Generalmente el enlace de clases Java se realiza de manera implícita y cuando todo "marcha bien" no se ve afectada la ejecución normal de los programas. En esta tecnología el enlace dinámico es transparente, tanto para programadores como para usuarios.

2.5. Resolución de Referencias Simbólicas

Las referencias simbólicas son entradas en la sección Constant pool, donde se define el enlace entre clases. Es un puntero identificado por un nombre de clase completo (fully qualified name) que establece las dependencias a nivel clase. En tiempo de ejecución, la JVM deriva estas referencias simbólicas en clases o interfaces a través del enlace dinámico. Cada implementación de JVM puede elegir resolver cada referencia simbólica a una clase o interfaz individualmente cuando se utiliza (lazy resolution), o bien, resolver todas en simultáneo cuando la clase es verificada (eager resolution).

En ambas estrategias la resolución es a demanda y en tiempo de ejecución, por lo tanto, disponer del recurso

completo (binario de clase) en tiempo de compilación es un requerimiento que se puede postergar hasta el momento en que el programa es ejecutado.

3. Antecedentes

El éxito de una plataforma de software está, en gran parte, ligada a la disponibilidad masiva de bibliotecas publicadas bajo licencia de código abierto [9]. Para el caso del software Java, el procedimiento estándar consiste en distribuir los binarios (bytecode) comprimidos en archivos de formato ZIP, pero con extensión Jar. Este empaquetado no establece como obligatoria la presencia de un manifiesto. Es optativo, y puede contener meta-datos tales como: firma electrónica, control de versiones, declaración de dependencias o el punto de entrada (Main-Class). Para el caso de los bundles OSGi [10], que también son empaquetados en archivos Jar, el manifiesto es obligatorio. Allí se declara el nombre del bundle, identificador, versión, dependencias e interfaces.

A fin de facilitar la gestión y acceso a dependencias para la compilación de proyectos Java, han surgido herramientas como Apache Maven, actual estándar de facto para todo desarrollo de software a gran escala [1]. Maven propone un modelo de descripción genérica del proyecto, a través de un archivo POM (Project Object Model) donde se especifican las dependencias, jerarquías, ciclo de vida de la construcción, parámetros de configuración, casos de prueba y otros [11]. Para optimizar la gestión de dependencias, esta herramienta crea un repositorio local en el cual se copian todos los archivos Jar requeridos por el proyecto. De este modo, los proyectos apuntan sus dependencias a la copia única, pudiendo establecer políticas a nivel particular o departamental, donde un grupo de desarrolladores comparte un repositorio único.

A simple vista este modelo parece ideal. No obstante, tanto la industria como la academia han detectado problemas. Uno de ellos es el denominado "Jar Hell", que se da cuando en un mismo classpath conviven clases que comparten un mismo nombre, o bien, cuando distintas versiones de una clase deben coexistir en un mismo classpath. Otro problema frecuente es la necesidad de contar también con dependencias transitivas (las dependencias de las dependencias) y con la versión apropiada [12].

Si bien Maven y otras herramientas similares proponen perfiles de compilación o bien, buscan automatizar la descarga de las dependencias transitivas, esta solución implica definir una serie de especificaciones no inherentes al proyecto, además de requerir recuperar del repositorio central la totalidad de las dependencias directas e indirectas [13].

3.1. Caso de ejemplo

Para visualizar las características de Maven, a modo de ejemplo, la Figura 2 muestra el uso de la herramienta para gestionar las dependencias de un ejemplo básico del software GeoTools[4], una vista parcial del listado de bibliotecas derivadas de las 2 directas requeridas (gt-shapefile y gt-swing).

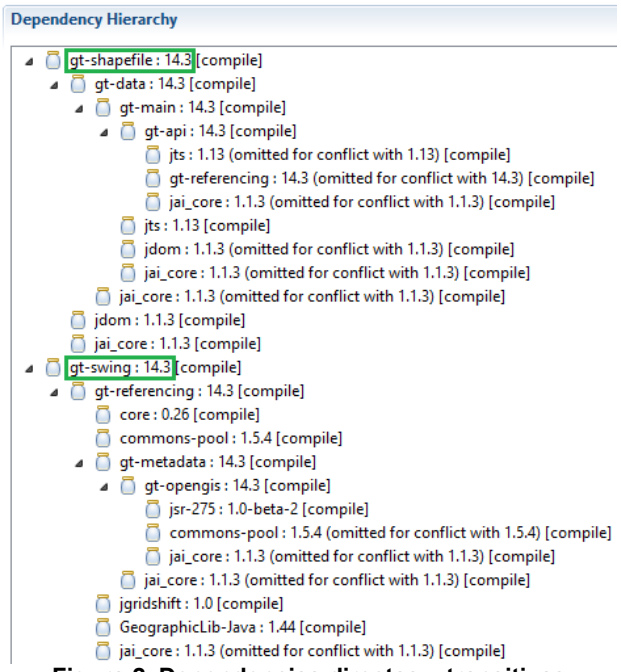


Figura 2. Dependencias directas y transitivas requeridas por el programa Quickstart de GeoTools.

Al definir en el archivo POM, las 2 dependencias directas del proyecto donde está el programa Quickstart, Maven resuelve los requerimientos de dependencias transitivas, resultando en la descarga local de 57 bibliotecas adicionales para poder compilar y ejecutar el programa de 16 líneas de código que se muestra en la Figura 3.

```
public class Quickstart {
  public static void main(String[] args) throws Exception {
     File file = JFileDataStoreChooser.showOpenFile("shp", null);
     if (file == null) {
         return;
     }
     FileDataStore store = FileDataStoreFinder.getDataStore(file);
     SimpleFeatureSource featureSource = store.getFeatureSource();
     MapContent map = new MapContent();
     map.setTitle("Quickstart");
     Style style = SLD.createSimpleStyle(featureSource.getSchema());
     Layer layer = new FeatureLayer(featureSource, style);
     map.addLayer(layer);
      JMapFrame.showMap(map);
  }
}
```

Figura 3. Código fuente del programa Quickstart.

[4] http://docs.geotools.org/latest/userguide/tutorial/quickstart/maven.html

Para cuantificar la tasa de uso (referencias) entre el programa Quickstart y sus dependencias, se emplea una herramienta desarrollada específicamente para este proyecto denominada Deep[5] [2]. Este programa analiza el bytecode de todas las clases de ambos archivos Jar y mide, sobre el total de recursos públicos disponibles por la biblioteca (Library Jar), qué proporción es referenciada por el Jar de origen (Source Jar). En la Figura 4 se muestra un ejemplo de la interfaz de usuario web.

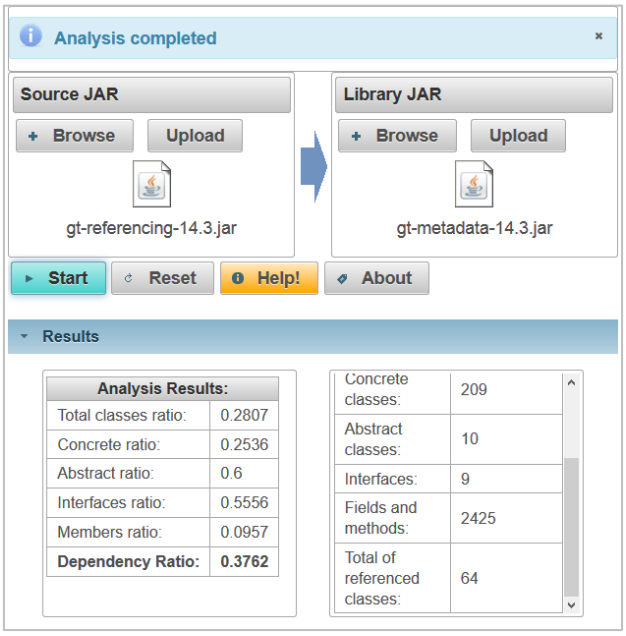


Figura 4. Interfaz gráfica web de la herramienta Deep.

La Tabla 1 muestra los resultados de medir con Deep la tasa de utilización entre el programa Quickstart, sus dependencias directas (Nivel 0), algunas dependencias transitivas (Nivel 1) y el segundo nivel de algunas dependencias transitivas (Nivel 2).

Tabla 1. Tasa de utilización de recursos del programa Quickstart y sus dependencias.

<table>
<tr><th colspan="3">Nivel 0</th><th></th></tr>
<tr><td rowspan="8">quickstart</td><th colspan="2">Nivel 1</th><td></td></tr>
<tr><td rowspan="6">gt-swing (→ 0.0125)</td><th colspan="2">Nivel 2</th></tr>
<tr><td rowspan="5">gt-referencing (→ 0.0166)</td><td>core (→ 0.1325)</td></tr>
<tr><td>commons-pool (→ 0.2602)</td></tr>
<tr><td>gt-metadata (→ 0.3762)</td></tr>
<tr><td>jgridshift (→ 0.3125)</td></tr>
<tr><td>geographicLib (→ 0.2288)</td></tr>
<tr><td>gt-render (→0.1817)</td><td rowspan="2">-sin medir -</td></tr>
<tr><td>miglayout (→0.0714)</td></tr>
<tr><td>gt-shapefile (→ 0.0000)</td><td colspan="2">-sin medir-</td></tr>
</table>

[5] http://trimatek.org/deep

En los resultados de la Tabla 1 se observa que Quickstart referencia menos de un 2% del total de recursos públicos disponibles en la biblioteca gt-swing, porcentaje que tampoco es superado entre esa biblioteca y su dependencia gt-referencing. Sí, hay un incremento en el 2do nivel, entre gt-referencing y sus dependencias, mostrando un porcentaje próximo al 40% de referencias a los recursos públicos de gt-metadata. A modo de resumen, se puede decir que entre el programa de usuario y las dependencias de Nivel 0, la tasa de utilización en promedio no supera al 2% del total de recursos públicos. Luego, las referencias entre gt-swing y sus dependencias no alcanza a promediar el 10%. Por último, entre gt-referencing y sus dependencias hay un nivel de utilización mayor, aproximándose a un promedio del 30%.

En base a lo presentado en las secciones 2 y 3, a modo de conclusiones preliminares se puede afirmar:

1. Durante la compilación sería prescindible contar con las bibliotecas transitivas, dado que las referencias simbólicas del programa de usuario sólo apuntan a recursos de dependencias directas.
2. Recién al momento de ejecutar el programa, la JVM resolverá las dependencias simbólicas.
3. Analizando la información disponible en el Constant pool del binario, es posible conocer puntualmente qué clases son referenciadas por el programa de usuario y las bibliotecas.

Requerir contar con la totalidad de las dependencias, inclusive las transitivas, simplemente para compilar un programa que emplea menos del 2% de los recursos públicos de sus dependencias, es una solución ineficiente y desactualizada. Por otro lado es una realidad que la tecnología Java, en gran parte por la masividad de Android , y también por el importante impulso que le dio Oracle con la versión 8, sigue liderando como lenguaje de programación de propósito general [14]. Esto justificaría proponer un sistema de gestión de dependencias acorde a la tecnología actual y más eficiente.

Tomando como referencia el trabajo de Petrea y Grigoraș para plataformas móviles de recursos limitados [15], este proyecto propone migrar del vuelco total de bibliotecas a un suministro a demanda de clases para los ambientes de desarrollo.

4. Descripción Conceptual

A continuación se describirán las características generales de la propuesta.

4.1. Proxy de bibliotecas y clases a demanda

Con el objetivo de centralizar y optimizar la gestión de bibliotecas, se propone reemplazar el sistema actual por uno que suministre las clases requeridas, en función de las relaciones establecidas en el binario Java.

Como se explicó en la sección anterior, recién en tiempo de ejecución es necesario contar con el recurso real. Por lo tanto, en tiempo de compilación sería suficiente conocer la firma de los recursos públicos, sin necesidad de tener el recurso real. Además, dado que el compilador conoce los recursos y características de las clases a partir del Constant pool, es posible compilar código fuente Java y establecer relaciones entre una clase y otra, sin necesidad de contar con el bytecode de las instrucciones. En relación a esto, la Figura 5 muestra una extracción del Constant pool de la clase Quickstart obtenida con el programa javap[6] donde se ve en la entrada #80 una referencia simbólica a la clase org.geotools.data.FileDataStore, que es parte de la biblioteca gt-swing.

```
public class prueba.Quickstart
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Class              #2             // prueba/Quickstart
   #2 = Utf8               prueba/Quickstart
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
  [...]
  #79 = Utf8               store
  #80 = Utf8               Lorg/geotools/data/FileDataStore;
  #81 = Utf8               featureSource
```

Figura 5. Extracto del Constant pool de la clase Quickstart.

Por lo tanto, se propone compilar a partir de un proxy de la biblioteca, un representativo que contenga únicamente los descriptores y firmas de las clases [16]. Luego, un servicio web analiza y entrega a demanda sólo las clases que son referenciadas por el programa del usuario en las dependencias directas y transitivas, tal como se ilustra en la Figura 6.

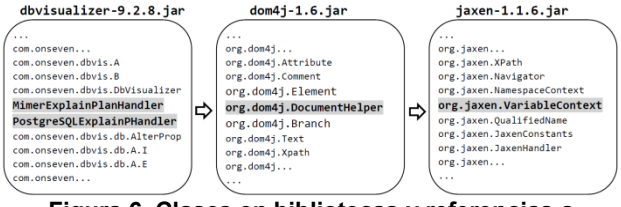


Figura 6. Clases en bibliotecas y referencias a dependencias.

Esta solución contempla la presencia de un intermediario entre los repositorios de bibliotecas y el entorno de desarrollo: un servicio que también sea despachante de representativos, es decir, proveedor de versiones simplificadas de las bibliotecas, en cuyas clases sólo estén las firmas de los recursos públicos. Esos proxies de bibliotecas permitirán la compilación del programa del usuario.

Previamente a la ejecución, el cliente solicita las clases de las dependencias directas que son referenciadas por el programa del usuario y es el despachante quien obtiene esas clases, junto con todas las clases de las bibliotecas referenciadas por esas clases, sus superclases, y lo mismo para las clases de las dependencias transitivas. Así, recursivamente el servicio despachante analiza las clases "apuntadas" por el programa de usuario y entrega al ambiente de desarrollo sólo los binarios necesarios.

La solución propuesta se puede representar con el diagrama que presenta la Figura 7.

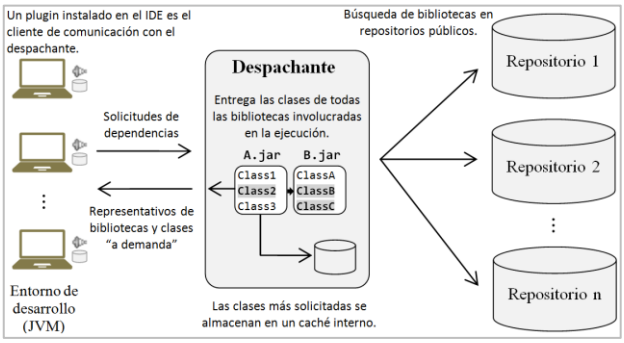


Figura 7. Servicio despachante de clases a demanda.

Además, el servicio despachante se basará en la información definida en los archivos POM (Project Object Model) de las bibliotecas almacenadas en los repositorios. De esta manera, conocerá en qué Jar están las clases referenciadas.

4.2. Características particulares

El fundamento principal de esta propuesta es que en la mayoría de los proyectos de software, sólo se emplea una mínima porción de los recursos públicos de una biblioteca. Como se estudió en una fase previa de este proyecto, habitualmente la tasa de utilización de bibliotecas Java no supera el 10% del total de recursos públicos (clases, interfaces, métodos y variables de clase) [2]. Esta subutilización de recursos justificaría que, en fase de desarrollo, se trabaje con una granularidad de dependencias a nivel de clase Java en lugar de hacerlo con la biblioteca completa. Además, se propone que el despachante analice el camino definido entre referencias a clases además de seleccionar y entregar al ambiente de desarrollo sólo las clases requeridas para la ejecución.

De este modo, y a diferencia de Maven, la clausura de bibliotecas se produce en un servidor que, en función de las dependencias directas, adapta a demanda el despacho de clases de todos los niveles involucrados. Además, la

[6] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html

solución propuesta abstrae a los clientes de los repositorios de bibliotecas, delegando en el servidor la lógica de búsqueda de recursos externos al programa de usuario.

A nivel local, el proyecto también deberá contar con una descripción de identificador de artefacto y versión de biblioteca, pero la recuperación no se realizará a través de solicitudes directas del ambiente de desarrollo al repositorio. En este modelo, se propone que sea el despachante quien determine dinámicamente de cuál repositorio obtener la biblioteca, evitando así la rigidez que implica trabajar con direcciones a fuentes estáticas. También se contempla la posibilidad de contar con un caché de bibliotecas, con el propósito de agilizar el tiempo de respuesta, además de ser utilizado como alternativa de contingencia.

5. Prototipo

A modo de prueba de conceptos, se diseñó e implementó un prototipo del sistema. El software se divide en tres componentes principales: 1. Interfaz con el usuario (UI), 2. *Despachante* y 3. Repositorio de bibliotecas. 1 y 2 fueron desarrollados para este proyecto, 3 es un servicio de terceros[7] [17]. Tanto para UI como para el servicio despachante, la tecnología marco es OSGi [18].

5.1. Arquitectura

El diseño general del prototipo se puede representar con el diagrama de la Figura 8.

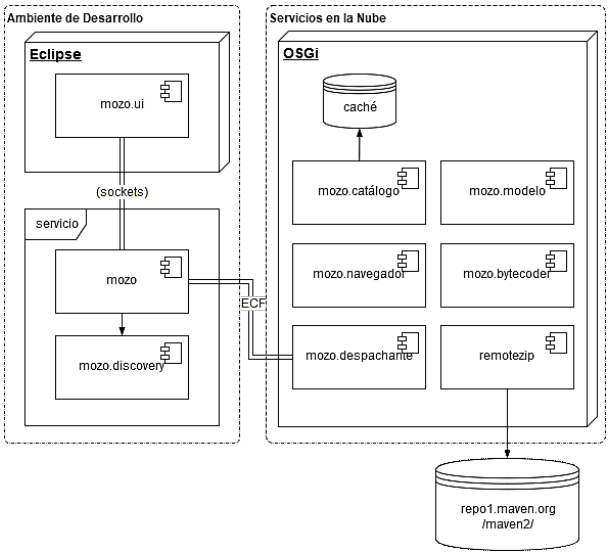


Figura 8. Arquitectura general del prototipo.

El proyecto se denomina Mozo y en relación al diagrama de la Figura 7, la solución se plantea como un intermediario o middleware entre el ambiente de desarrollo y los repositorios de bibliotecas. Son 9 módulos que, a través de servicios OSGi, asisten a la compilación y ejecución de las solicitudes de un programa en desarrollo. A continuación se presenta una breve descripción de cada componente.

5.1.1. Ambiente de desarrollo

Del lado del usuario, el sistema se despliega como un plugin para Eclipse (IDE) y un proceso en segundo plano (background process). El plugin es la interfaz con el usuario y el proceso es el componente que interactúa con los servicios del módulo *Despachante*, mediante Eclipse Communication Framework (ECF)[8]. El módulo Discovery se encarga de localizar los servicios en la nube y guiar al módulo Mozo hacia *Despachante*.

La comunicación entre los módulos mozo.ui y el servicio mozo es a través de sockets asíncronos, de modo tal de no bloquear la instancia del IDE durante las solicitudes del usuario.

A partir de una contribución al menú contextual de la vista Package Explorer, el plugin permite seleccionar entre dos opciones: Solicitar proxies y Obtener clases. La primera analiza las dependencias del archivo POM del proyecto Java y genera una solicitud de dependencias de primer nivel al *Despachante* para poder compilar. La segunda releva todas las clases referenciadas en el proyecto y solicita al *Despachante* obtener las clases necesarias para ejecutar.

5.1.2. Servicios en la Nube

El núcleo del sistema es un conjunto de microservicios donde el punto de entrada son los dos servicios que expone el módulo *Despachante* (Ver Figura 9). El servicio Mozo (ambiente de desarrollo) invoca los métodos remotos loadJarProxy y fetchDependencies de mediante ECF. Todo el proceso de serialización/deserialización es administrado de forma transparente por el framework y, a nivel OSGi, el módulo Mozo interactúa con *Despachante* como si se tratara de un servicio local.

[7] https://repo1.maven.org/maven2/

[8] http://www.eclipse.org/ecf/

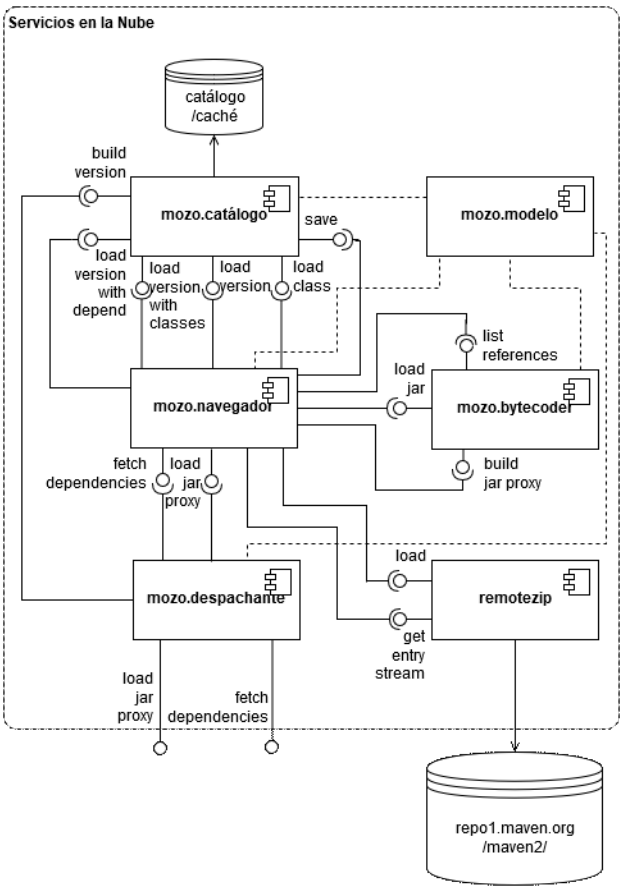


Figura 9. Conjunto de microservicios que conforman el intermediario entre el IDE y los repositorios.

El sistema almacena en un repositorio local copias de las bibliotecas despachadas al IDE para optimizar el tiempo de respuesta en futuras solicitudes. Como se explicaba conceptualmente en la sección anterior, *Despachante* ofrece dos funcionalidades principales: 1. Generar y entregar proxies de las bibliotecas y 2. Seleccionar y proveer las clases requeridas para ejecutar. A continuación se explicarán brevemente las características y responsabilidades de cada componente.

Despachante: Es el punto de entrada al conjunto de microservicios. Interactúa con el plugin Eclipse a través del proceso de segundo plano Mozo. Recibe solicitudes de proxies o clases. Este módulo verifica la consistencia de las solicitudes/respuestas y delega en el módulo *Navegador* las decisiones de cada caso.

Navegador: Es el componente que centraliza la organización y gestión de recursos. En función de las características de la solicitud, activa los servicios de los demás módulos del sistema. Por ejemplo, si recibe una solicitud de bibliotecas de primer nivel, deberá retornar proxies de éstas. Por lo tanto, en primera instancia hace una solicitud al *catálogo*, en caso de no contar con alguna, solicita al módulo *Remotezip* la obtención del Jar desde el repositorio. Luego, activa el servicio buildJarProxy del módulo Bytecoder y el resultado es persistido como una cadena de bytes en el *catálogo*. Por último retorna el proxy al Despachante y es éste el que serializa la colección del resultado. Cuando recibe una solicitud de clases, activa el servicio listReferences del módulo Bytecoder. De no contar con las clases referenciadas en el *catálogo*, genera una solicitud a *Remotezip*, el cual recuperará puntualmente la clase requerida desde el Jar en el repositorio.

Remotezip: Este módulo es un subproyecto de Mozo basado en el trabajo de Emanuele Ruffaldi para la plataforma .NET [19]. Este módulo permite obtener fracciones de archivos ZIP, sin necesidad de copiar el archivo localmente. En primer lugar, el algoritmo busca el encabezado del directorio central del Zip para poder conocer el índice y ubicación de cada archivo contenido. Luego, con ese dato y si el servidor soporta solicitudes de rango de bytes (RFC2616: 206 - Partial Content), el módulo copiará únicamente la fracción que corresponde al archivo objetivo.

En este prototipo, el módulo *Navegador* solicita a *Remotezip* obtener clases contenidas en archivos Jar a través del servicio getEntryStream, luego de invocar a load (que retorna un listado con el contenido del directorio central). De esta manera se recuperan recursos ubicados en el repositorio a nivel de clase y no a nivel de biblioteca, como es habitual. Así, se consigue un tiempo de respuesta notablemente inferior al logrado teniendo que copiar el Jar completo, dado que los archivos de clases poseen un tamaño que difícilmente supere los 40 Kilobytes.

Este proyecto también está disponible bajo licencia de código abierto en un repositorio de acceso público: https://github.com/martinaguero/remotezip.

Bytecoder: En este módulo se realizan todas las operaciones de manipulación de bytecode Java. Para construir un proxy de Jar, en primer lugar se quitan todas las clases no públicas. Luego, por cada clase pública restante se crean versiones livianas de cada una, a partir de las siguientes acciones:

a. Por cada método público, se genera otro público donde sólo está la firma del original, es decir, sin instrucciones (Ver Tabla 2).
b. Por cada atributo público, se crea otro con el mismo tipo y denominación.
c. Se mantienen las referencias a otras clases del Constant pool original.

De este modo, otras clases podrán ser compiladas a partir de las referencias a los recursos públicos del proxy.

Tabla 2. Comparación entre método original y luego de procesado por Bytecoder.

Bytecode del método setPool() **original**	Bytecode del método setPool() **"ahuecado"**
public void setPool(org.apache.commons.pool.ObjectPool) throws java.lang.IllegalStateException, java.lang.NullPointerException; **descriptor: (Lorg/apache/commons/pool/ObjectPool;)V** **flags: ACC_PUBLIC** **Code:** **stack=3, locals=2, args_size=2** **0: aconst_null** **1: aload_0** **2: getfield** **5: if_acmpeq 18** **[15 instrucciones no mostradas]** **35: putfield** **38: return** **LineNumberTable:** **line 59: 0** **[4 líneas no mostradas]** line 66: 38 LocalVariableTable: **Start Length Slot Name Signature** **0 39 0 this Lorg/apache/commons/dbcp/PoolingDataSource;** **0 39 1 pool Lorg/apache/commons/pool/ObjectPool;** **StackMapTable: number_of_entries = 2** **frame_type = 18** **frame_type = 14** **Exceptions:** **throws java.lang.IllegalStateException, java.lang.NullPointerException**	**public void setPool(org.apache.commons.pool.ObjectPool) throws java.lang.IllegalStateException, java.lang.NullPointerException;** **descriptor: (Lorg/apache/commons/pool/ObjectPool;)V** **flags: ACC_PUBLIC** **Code:** **stack=0, locals=2, args_size=2** **0: return** **LocalVariableTable:** **Start Length Slot Name Signature** **0 0 0 this Lorg/apache/commons/dbcp/PoolingDataSource;** **0 0 1 arg0 Lorg/apache/commons/pool/ObjectPool;** **Exceptions:** **throws java.lang.IllegalStateException, java.lang.NullPointerException**
Método de la clase PoolingDataSource de la biblioteca Commons DBCP 1.4 mostrados con el programa javap. **Nota:** Por cuestiones de espacio, se omitieron 19 líneas del original.	

Como el compilador sólo necesita contar con las dependencias de primer nivel, el programa de usuario se compila a partir de versiones de bibliotecas reducidas entre un 70% y 80% del tamaño original (Ver Figura 10).

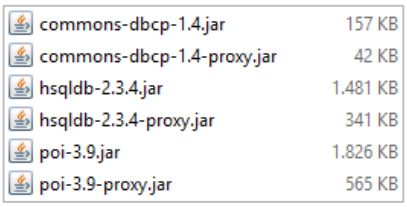


Figura 10. Diferencia de tamaño entre bibliotecas originales y representativos (proxy).

Esta técnica posee ciertas limitaciones derivadas de la herramienta empleada para manipular el bytecode: Apache BCEL 5.4[9], no soporta genéricos ni anotaciones. Por lo tanto, este módulo no podrá crear representativos totalmente correctos si la clase original emplea alguno de esos recursos. Está pendiente investigar si es posible solucionar esta limitación con otras bibliotecas de manipulación de bytecode, como por ejemplo ASM o Soot, en lugar de BCEL.

Este módulo también se encarga de listar todas las referencias a otras clases a partir del servicio listReferences.

[9] https://commons.apache.org/proper/commons-bcel

Catálogo: Es la entidad que administra la persistencia a un medio relacional. El módulo atiende solicitudes de guardar/recuperar proxies y clases por parte de *Navegador*. Organiza las clases en una simple jerarquía de *Repositorio* → *Grupo* (o fabricante) → *Producto* → *Version* (equivalente a Artefacto de Maven) → *Class*. En la base están las clases (*Class*) que son los archivos class de Java. La Figura 11 muestra la representación de las tablas del esquema.

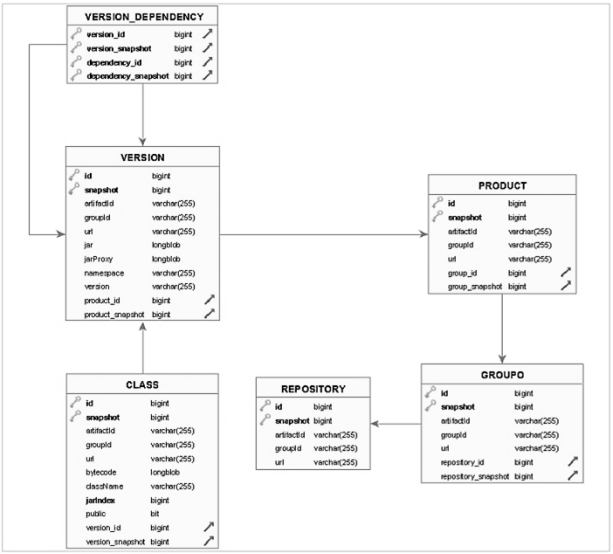


Figura 11. Esquema de persistencia relacional simple de catálogo / caché.

El soporte Objeto-Relacional está diseñado según la especificación JPA 2.1[10], con la implementación de Hibernate 5 [20]. Este módulo abstrae a *Navegador* de cualquier complejidad vinculada con la persistencia. También actúa como caché de clases y proxies, dado que tanto la tabla Version como Class persisten el binario (longblob) de los recursos recuperados del repositorio.

Modelo: Por último está el módulo Modelo que es referenciado por todos los otros. Contiene la definición de las interfaces más importantes, pero no expone ningún servicio.

A continuación en ejemplo de uso comparado con la herramienta Maven.

6. Caso de Uso

A modo de ejemplo se presenta el caso donde el objetivo es probar la herramienta intérprete de fuentes RSS *Informa*[11]. Esta biblioteca está publicada en el repositorio Maven y, según el archivo POM, posee 18

[10] https://jcp.org/aboutJava/communityprocess/final/jsr338/index.html
[11] http://informa.sourceforge.net/

dependencias de primer y segundo nivel. Si, como es habitual, se emplea Maven para la gestión de dependencias, éste descargará todas esas bibliotecas a una ubicación local y las referenciará como parte del classpath del proyecto (Ver Figura 12).

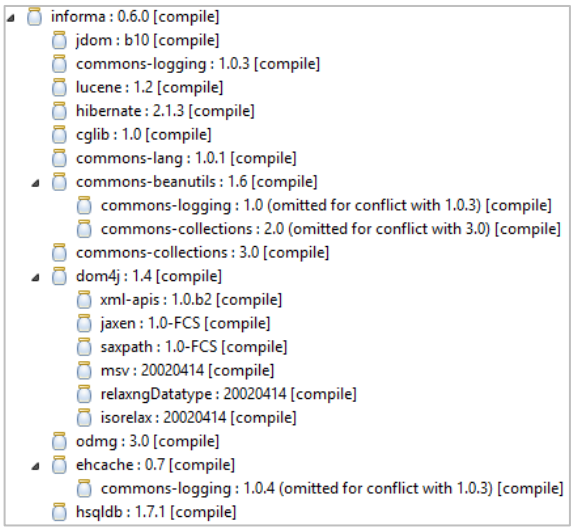


Figura 12. Dependencias de *Informa* 0.6.0 y sus transitivas.

Como se ve en la Figura 13, el programa de ejemplo son 13 líneas de código donde obtiene una fuente en formato XML y muestra algunos atributos por pantalla. Posee 2 referencias a clases que son parte de la API de Java y 4 referencias a clases del proyecto *Informa*. Con la palabra reservada *import* se establecen las dependencias de Nivel 0. A su vez, esas 4 clases pueden establecer referencias a otras clases que no son parte de la biblioteca *Informa*, esas son las dependencias de Nivel 1, y así sucesivamente.

```
import java.io.File;
import java.io.IOException;

import de.nava.informa.core.ChannelIF;
import de.nava.informa.core.ParseException;
import de.nava.informa.impl.basic.ChannelBuilder;
import de.nava.informa.parsers.FeedParser;

public class PruebaInforma {
    public static void main(String[] args)
            throws IOException, ParseException {
        File inpFile = new File("rss-feed.xml");
        ChannelIF channel = FeedParser.parse(
                new ChannelBuilder(), inpFile);
        System.out.println(channel.getCreator() +
                " - " + channel.getTitle());
        for (Object i : channel.getItems()) {
            System.out.println("Item: " + i);
        }
    }
}
```

Figura 13. Código fuente de ejemplo con *Informa*.

En la Tabla 3 se listan las dependencias de la biblioteca *Informa* y las dependencias transitivas medidas con Deep. En los casos donde se obtiene una medición 0, es posible que estén disponibles para ser referenciadas por el programa de usuario.

Tabla 3. Medición de tasa de referencias a dependencias obtenidas con Maven.

Nivel 0	Nivel 1	Nivel 2	
prueba-informa	informa (→ 0.0451)	jdom-b10 (→ 0.1881)	
		commons-logging (→0.6888)	
		lucene (→0.1259)	
		hibernate (→0.0260)	
		cglib (→0.0000)	
		commons-lang (→0.0000)	
		commons-beanutils (→0.0000)	commons-logging (→0.6814)
			commons-collections (→0.0063)
		commons-collections (→0.0191)	
		dom4j (→0.0000)	xml-apis (→0.5694)
			jaxen (→0.7939)
			saxpath (→0.9233)
			msv (→0.0000)
			relaxngDataty (→0.2183)
			isorelax (→0.0000)
		odmg (→0.0000)	
		ehcache (→0.0000)	commons-logging (→0.6869)
		hsqldb (→0.0000)	

En el Nivel 0, la tasa de referencias a la biblioteca informa.jar es de un 4%, en el Nivel 1 no supera al 10% y en Nivel 2 alcanza, en promedio, un 43%, por el alto acoplamiento entre dom4j y ehcache con sus dependencias. No obstante, lo curioso es que entre dom4j e *informa* no se relevaron referencias directas.

6.1. Gestión de dependencias con Mozo

El prototipo del sistema denominado Mozo está conformado por 3 partes: integración con el entorno de desarrollo a través de un plugin Eclipse, cliente de comunicación con el middleware y una colección de servicios en la nube (Ver diagrama de Figura 8).

Modo de Uso: Desde un proyecto Java de Eclipse el usuario define las dependencias de Nivel 0, a través de un archivo POM. Luego, ubica el cursor en el nodo raíz del proyecto y, desde el menú contextual, selecciona la opción *Solicitar proxies,* que es parte de la sección *Mozo* (Ver Figura 14). En ese momento Eclipse invoca el

método execute() de una subclase de AbstractHandler, que lee las dependencias definidas en el archivo POM. Una vez relevadas las dependencias directas, transmite por sockets una instancia de comando al servicio local que traducirá esa solicitud en una invocación al servicio loadJarProxy() del componente *Despachante*. La comunicación entre el componente Mozo y *Despachante* es sincrónica, no así entre Mozo y Eclipse, por lo que no se bloqueará la interfaz de usuario en ningún momento. Cuando *Despachante* haya finalizado de procesar la solicitud, entregará a Mozo los proxies de bibliotecas solicitados y éste transmitirá la respuesta por sockets al plugin Eclipse. El plugin procesará el resultado y agregará al classpath del proyecto los archivos Jar recuperados. De este modo, el programa de usuario se compilará a partir de versiones livianas de las bibliotecas. Es decir, quedará provisoriamente enlazado a representativos de las clases originales. Una vez que el programa fue compilado, antes de ejecutarlo, el usuario deberá solicitar recibir las clases referenciadas en forma directa e indirecta (Ver Figura 14), también a través de una opción en el menú contextual de Eclipse[12].

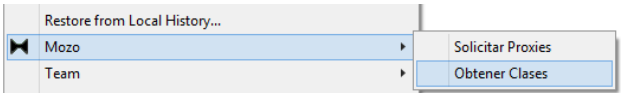


Figura 14. Menú contextual de Eclipse con plugin Mozo.

El IDE invocará el método execute() de otra subclase de AbstractHandler, donde se relevan los *imports* del código fuente mediante la iteración de elementos que implementan la interfaz ICompilationUnit y se envía el listado empaquetado como una clase comando hacia el servidor de sockets que publica el servicio Mozo. Éste reenvía la solicitud al servicio fetchDependencies() publicado en la nube por el módulo *Despachante*. Internamente, se obtienen las clases referenciadas en todos los niveles y retorna al cliente (Mozo) el conjunto de clases indexadas por biblioteca de origen. Por último, las clases "ahuecadas" se reemplazan por las originales dentro del Jar de Nivel 0 y las clases de dependencias transitivas (Nivel 1, 2, n) son reempaquetadas como un Jar, y también sumadas al classpath.

El programa se ejecuta normalmente a partir de un suministro "personalizado" de clases. Para el caso presentado, en tiempo de compilación, sólo es necesario contar con el representativo de biblioteca de nivel 0 (informa.jar). Luego, previamente a la ejecución y en base al análisis de las referencias simbólicas definidas en el Constant pool de las clases referencias (Nivel 0, 1 y 2), se suman al classpath jdom-b10, commons-logging y commons-collections, sólo integrados por las clases "a demanda" de los requerimientos del programa de usuario. En la Tabla 4 se muestran las bibliotecas seleccionadas por Mozo para que el programa PruebaInforma pueda ser ejecutado.

Tabla 4. Medición de tasa de referencias a dependencias obtenidas con Mozo.

Nivel 0	Nivel 1	Nivel 2
prueba-informa	informa (→ 0.1423)	jdom-b10 (→ 0.2618)
		commons-logging (→0.7089)
		common-collections (→0.0170)

A simple vista se puede observar que sólo fueron transferidas (desde el repositorio al entorno de desarrollo) las bibliotecas con las clases que utiliza el programa de usuario. Midiendo las referencias con Deep, ahora se obtiene que a Nivel 0 la tasa de referencias ha aumentado de un 4% a un 14% y a Nivel 1 la tasa es de un 33%, contra el casi 10% de la Tabla 3. Además, para ejecutar este programa no fue necesario contar con las dependencias de Nivel 3. Es apropiado aclarar que muchas clases internas pueden tener visibilidad pública de forma tal que puedan ser accedidas por clases de la misma biblioteca, pero ubicadas en diferentes paquetes.

Todo el software del prototipo está disponible como código abierto en este repositorio de acceso público: https://github.com/martinaguero/mozo.

7. Conclusión y Trabajos Futuros

Las herramientas de soporte a la gestión de dependencias Java como Maven cubren los requerimientos de los programas de usuario mediante el vuelco total de dependencias directas y transitivas a un repositorio local. Ese modelo de distribución de binarios por paquetes o bibliotecas que ha sido legado de la organización interna de los sistemas operativos ya es obsoleto.

Por otro lado, la tecnología Java establece que durante la compilación sólo deben definirse las referencias simbólicas (o punteros) a otras clases, postergando al momento de ejecución el enlace real con esos recursos. En cuanto a las bibliotecas disponibles en los repositorios, todas cuentan con las referencias a sus dependencias. Este trabajo ha estudiado cómo éstas características permiten analizar el binario Java y, en función de las referencias simbólicas, localizar dinámicamente las clases de sus dependencias. Asimismo, también se explica de qué modo es posible

[12] En una versión posterior se prevé reemplazar estas acciones manuales por otras automáticas inmediatas al guardado de archivos POM y fuentes.

extraer únicamente archivos puntuales de un Jar/Zip remoto.

A partir de la medición de referencias entre clases, se dimensionó la tasa de utilización de recursos de bibliotecas, demostrando que en varios casos es mínima o nula según la funcionalidad requerida. Esto estaría indicando que el vuelco completo de bibliotecas es una solución ineficiente y desactualizada.

Mediante un prototipo funcional se validaron estos conceptos, proponiendo la presencia de un intermediario entre el entorno de desarrollo (IDE) y los repositorios de bibliotecas. De este modo se desacopla al cliente de los repositorios mediante un servicio web que suministra representativos o proxies para compilación y las clases referenciadas por las bibliotecas y sus dependencias para ejecución.

A modo de conclusión general, se puede afirmar que es factible migrar de un modelo de vuelco total de bibliotecas a un suministro a demanda de clases, cambiando el nivel de granularidad de la resolución de dependencias Java. Pasando así de proveer a nivel biblioteca, a analizar los archivos de clases y transferir sólo el binario requerido por el programa de usuario.

Para una fase posterior se planea incorporar al prototipo compatibilidad con genéricos, anotaciones, y reflexión del lenguaje Java. También se proyecta reemplazar el uso del menú contextual del IDE por eventos asociados al guardado de archivos.

8. Referencias

[1] Varanasi, B., Belida, S., "Introducing Maven", *Springer Science+Business Media.* Apress, 2014.

[2] Agüero, M., Ballejos, L., Pons, C., "Deep: Una Herramienta para Medir Dependencias Java", en *3er Congreso Nacional de Ingeniería Informática / Sistemas de Información (CoNaIISI)*, 2015.

[3] Abate, P., Boender, J., "Strong Dependencies between Software Components", en *Proceedings of the IEEE 3er International Symposium on Empirical Software Engineering and Measurement*, 2009.

[4] Powel, B., Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems, Addison-Wesley, 2002.

[5] Lindholm, T., Yellin, F., Bracha, G., Buckey, A., The Java Virtual Machine Specification, 8th Edition, Oracle America, 2015.

[6] Liang, S., Bracha, G., "Dynamic Class Loading in the Java Virtual Machine", en *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1998.

[7] Eisenbach, S., Sadler, C., Shaikh, S., "Evolution of Distributed Java Programs", en *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, Springer-Verlag, 2002.

[8] Agüero, M., Ballejos, L., Pons, C., "Resolución más eficiente de dependencias Java", en *XXI Congreso Argentino de Ciencias de la Computación*, CACIC, 2015.

[9] Ossher, J., Bajracharya, S., Lopes, C., "Automated Dependency Resolution for Open Source Software", en *7th IEEE Working Conference on Mining Software Repositories*, 2010.

[10] OSGi Core Release 6, OSGi Alliance, 2014.

[11] McIntosh, S., Adams, B., Hassan, A., "The evolution of Java build systems", *Empirical Software Engineering*, Springer, 2012.

[12] Jezek, K., Dietrich, J., "On the use of static analysis to safeward recursive dependency resolution", en *40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2014.

[13] Apache Maven, "Introduction to the Dependency Mechanism". En línea: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html. Consultado: 20/07/2016.

[14] TIOBE, "TIOBE Index for July 2016". En línea: http://www.tiobe.com/tiobe-index/. Consultado: 02/08/2016.

[15] Petrea, L., Grigoraș, D., "Remote Class Loading for Mobile Devices", en *IEEE 6th International Symposium on Parallel and Distributed Computing*, 2007.

[16] Frénot, S., Ibrahim, N., Le Mouël, F., Ben Hamida, A., "ROCS: A Remotely Provisioned OSGi Framework for Ambient Systems", en *IEEE Network Operations and Management Symposium*, 2010.

[17] Karakoidas, V., Mitropoulos, D., Louridas, P., Gousios, G., Spinellis, D., "Generating the Blueprints of the Java Ecosystem", en *IEEE 12th Working Conference on Mining Software Repositories*, 2015.

[18] Knoernschild, K., *Java Application Architecture: Modularity Patterns with Examples Using OSGi*, Prentice-Hall, 2012.

[19] Ruffaldi, E., "Extracting files from a remote ZIP archive". En línea: http://www.codeproject.com/Articles/8688/Extracting-files-from-a-remote-ZIP-archive. Consultado: 02/08/2016.

[20] Keith, M., Schincariol, M., *Pro JPA 2*, Apress, 2013.

Congreso: Open Source Systems 2017

Simposio/Área: FLOSS technologies and applications

Título: Cloud Dependency Management: An efficient proposal for Java

Cloud Dependency Management: An efficient proposal for Java

Martín Agüero[1] and Luciana Ballejos[2]

[1] Universidad Tecnológica Nacional, FRBA, Argentina
[2] Universidad Tecnológica Nacional, FRSF, Argentina

Abstract. Based on Package Managers, the software industry developed Dependency Managers. These tools interact with library repositories supporting software development environments in the process of obtaining all required resources for compiling the new software. In Java technology, this adaptation suffers from certain drawbacks: dependency underutilization, performance issues, high coupling with repositories, and conflicts between libraries. The first two would be solved by the new Java release, but the others will still require improvements in the dependencies management technology. The tool proposed in this article aims to update the way Java software dependencies are resolved, shifting the current desktop-centric tools like Maven or Gradle, to a specialized and efficient middleware in the cloud.

Keywords: Dependency management, Java, Library repository, Cloud computing, Software engineering.

1 Introduction

1.1 Dependencies Management

The free software community permanently increases its capabilities through the reuse of components shared as libraries or packages. Nowadays, most software projects tend to be designed as a composition of resources with specific capabilities. This is also seen as a key success factor, either by proven quality or immediate integration of a new feature [1].

Since the beginning of the '90s, different tools have emerged to support the integration and update of operating systems through packages. These tools, called Packet Managers, allow adding and removing, atomically, software packages from external repositories. The software industry also adopted the packet distribution model, defining the library as a set of reusable, indivisible, and highly cohesive software elements. These libraries are usually available in public repositories, like ibiblio or The Central Repository.

In the case of Java technology, libraries are compressed files in Zip format with Jar extension, where the manifest is not mandatory, delegating to third party solutions the formal declaration of their dependencies. Although the adaptation of the Package Manager model to the development environment has been successful, it suffers from

the following drawbacks: **Dependency Underutilization:** The model of total library dump to a local environment was appropriate for the 1990s and early 2000s, when Internet access was limited. However, requiring so many pieces of unused software limits portability to production environments with limited resources, such as IoT devices or cloud computing [2]. **High Coupling:** The most widely used dependency resolution model relays on the client, through the project descriptor, to find the libraries in repositories. Studies reveal that almost 40% of the cases where the construction of a project using a Dependency Manager fails, being the problem related to the dependencies resolution [3]. **Performance:** In limited resources environments (IoT devices or virtual servers), it is expensive to bind classes at runtime, in terms of processor cycles. This is increased if the libraries have a large number of unreferenced classes [4]. **Conflicts between Libraries:** Conflicts between libraries occur frequently during dependencies resolution processes. Dependency Managers must have enough analytical capabilities to avoid these conflicts [5] [6].

1.2 Java Platform Module System (JPMS)

The 9th version of the platform proposes to fragment the monolithic JDK to a module-based platform, which will be the foundation for module-based software development [7]. In the implementation of the specification (Jigsaw project), libraries are divided into modules, and modules are interconnected through a mandatory descriptor that defines: module name, module requirements, packages that the module exports, and the services that it publishes and consumes. These modules, like the Java libraries, are sets of Java classes bundled in Zip files with Jar extension. The module system also fragments the location from which the classes are loaded, proposing a modular class-path, which would speed up the class loading response time. This new version also includes a static modules linker to create customized and optimized execution platforms for the target platform.

Dependency Underutilization and Performance issues, would be addressed by Java 9. The module system fragments the platform in a ratio of 1:20, which would significantly reduce this underutilization of libraries. JDK 9 would also provide a solution for performance problems derived from the monolithic architecture. With a modular class-path and the capability to create custom images, the platform would be solving these problems. The remaining issues: High Coupling and Conflicts between Libraries, are addressed by this project as explained in the following sections: Section 2 describes the proposed tool. Section 3 presents a demonstration example and, finally, Section 4 describes conclusions.

2 Tool

To avoid the high coupling between the development environment and the repositories, and the conflicts between the libraries, this project proposes moving the dependency resolution from the desktop (local environment) to a specialized middleware in the cloud (see Fig. 1).

Considering that in Java 9 each module includes a mandatory descriptor, the 'module-info' file is enough to know the dependencies of each module. To analyze each module descriptor, this prototype, just transfers the bytes that represent the module-info file, loaded from the repositories with Remote Zip open source subsystem[1].

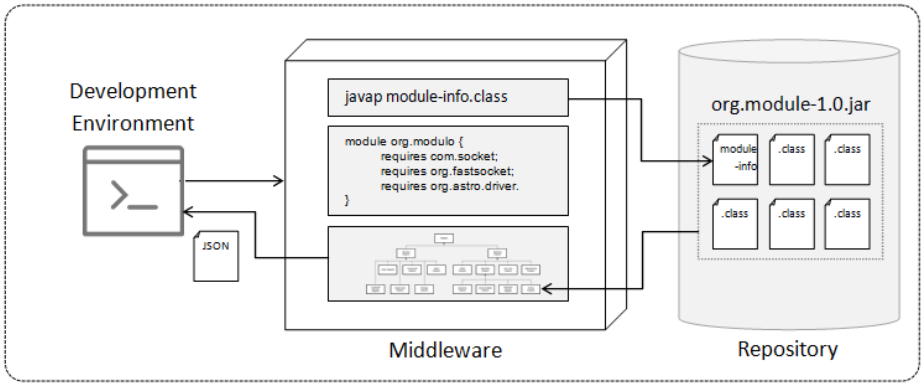


Fig. 1. Middleware between Development Environment and Repositories.

Finally, the middleware builds a dependencies graph with all involved modules, including the repository path to each of them. This tool is called Mozo (waiter in Spanish) and it is also freely available as open source[2].

2.1 Development Environment (Client)

Instead of presenting an old-fashioned locally-installed software, like Maven or Gradle, which are outdated and difficult to maintain, this solution proposes a light client based on a command line interface, that remotely loads the classes involved in the required operations. The client is a tiny Java class that runs from command line and loads specific remote Java classes, with URL Class Loading facility. This design feature enables to transparently keep the client updated.

3 Demonstration

Based on Jigsaws' early access builds and documentation[3], a basic test case was designed. The example is a statistics system, where the modules involved are displayed in the left side of Fig. 2.

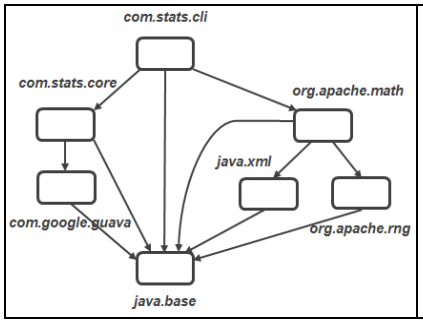
To start using Mozo and load all necessary modules in order to get the statistics system completely available, the following steps are required:

1. Download Mozo.class file from `http://trimatek.org/mozo/Mozo.class`
2. Run it from command line with: `java -cp . Mozo`

[1] https://github.com/martinaguero/remotezip
[2] https://github.com/martinaguero/mozo
[3] https://jdk9.java.net/jigsaw/

3. In console prompt enter: `find-modules com.stats.cli`

In Step 3, the client will load the FindModules class, which will pass the request to the middleware. Once it finishes evaluating the required modules and verifying their location, it replies to the client with the result data, saving it in res0 variable (See Fig. 2, right column).




```
"from": "user-request: /mozo/find?modules=com.stats.cli",
 "requires": [
  {
   "module": "com.stats.cli",
   "path": "http://www.trimatek.org/repository/com.stats.cli.jar",
   "requires": [
    {
     "module": "com.stats.core",
     "path": "http://www.trimatek.org/repository/com.stats.core.jar",
     "requires": [
      {
       "module": "com.google.guava",
       "path": "http://www.trimatek.org/repository/com.google.guava.jar"
      }
                                    - continues -
```


Fig. 2. com.stats.cli modules dependencies (left) and resolution graph.

4. The final step is to download the modules: `download-modules res0`

In Step 4, the client running class (Mozo.class) will load the remote Download class and it will begin copying the files from the sources (repositories) to a local path (java. modules are not downloaded because they are part of the JDK).*

4 Conclusions

This work aims to update the way Java software dependencies are resolved, passing from a model focused on: metadata, desktop and repositories, to a specialized cloud service that can support current and future demands of the software industry.

References

1. Wnuk, K., Regnell, B., Berenbach, B.: Scaling Up Requirements Engineering, Exploring the Challenges of Increasing Size and Complexity in Market-Driven Software Development. REFSQ '11. Springer-Verlag. (2011).
2. Wang, P., Yang, J., Tan, L., Kroeger, R., Morgenthaler, D.: Generating Precise Dependencies for Large Software. MTD '13. (2013).
3. Sulír, M., Porubän, J.: A Quantitative Study of Java Software Buildability. PLATEAU 2016. (2016).
4. Mausolf, J., Austin, Kimberly, Ann.: Classpath Optimization in Java Runtime Environments", US Patent: US 9.069.582 B2. (2015).
5. Jezek, K., Dietrich, J.: On the Use of Static Analysis to Safeward Recursive Dependency Resolution. 40th EUROMICRO. SEAA'14. (2014).
6. Kuniyasu, C., Di Cosmo, R., Treinen, R, Zacchiroli, S.: Why Do Software Packages Conflict?: MSR '12. (2012).
7. Reinhold, M.: JSR 376 – Java Platform Module System. Oracle Corporation. https://www.jcp.org/en/jsr/detail?id=376, last accessed 2017/01/06.

Congreso: Conferencia Latinoamericana de Informática 2017

Simposio/Área: Simposio Latinoamericano de Ingeniería de Software

Título: Dependency Management in the Cloud: An Efficient Proposal for Java

Dependency Management in the Cloud: An Efficient Proposal for Java

Martín Agüero and Luciana Ballejos

Abstract— Since the beginning of the 90s, different tools have emerged to support integration and upgrading of most of the existing operating systems. These tools, called Package Managers, are used to atomically add and remove software pieces coming from external repositories. Based on Package Managers, the software community created Dependency Managers with the aim to give support to the development of wide-scale software. These tools interact with library repositories, aiding development environments in the dependency recovery and closure processes. Although the adaptation of the Package Manager model to software development has been successful, it suffers from certain drawbacks that significantly impact on productivity. This work offers a proposal that updates the way in which Java software dependencies are resolved, moving from a model centered on workstations, descriptor files and repositories to a service specialized in resolving dependencies that can support current and future demands of the software industry.

Index Terms—Software engineering, middleware, dependency management, open source software, software repository.

1 Introducción

La organización y distribución de software en paquetes ha demostrado ser una solución efectiva. Desde principios de los años '90, diferentes herramientas han surgido para dar soporte a la integración y actualización en la mayoría de los sistemas operativos [1]. Se conocen como Gestores de Paquetes (package managers). La industria del software también adoptó el modelo de distribución por paquetes, administrando su disponibilidad a través de Gestores de Dependencias. Al igual que los Gestores de Paquetes, éstos realizan una copia local de los paquetes o bibliotecas, que son las dependencias del software en desarrollo. Cada vez que un proyecto requiere una nueva dependencia, estas herramientas 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 sus dependencias transitivas (las dependencias de las dependencias). Estos componentes de software proveen un conjunto de funcionalidades específicas definidas en una interfaz que describe recursos provistos y requeridos [2]. De este modo se promueve la reutilización, siendo en la actualidad un factor clave de éxito, ya sea por calidad probada o integración inmediata de las nuevas prestaciones [3].

Este modelo de organización y distribución por paquetes, donde se define como servidor al repositorio central, y del lado del cliente los repositorios locales y el software gestor, fue diseñado hace más de 20 años, cuando recién se comenzaba a hablar de 'Web Services', y el término 'Cómputo en la Nube' apenas se consolidaba como metáfora para explicar un concepto emergente [4]. Desde el punto de vista arquitectónico, el modelo de gestión de dependencias se presenta como una relación estática entre el cliente y el repositorio central. En ese modelo, el cliente sólo tiene acceso a las fuentes que son declaradas en su ámbito local, estableciéndose un alto nivel de acoplamiento entre el cliente y el servidor. Por otro lado, también se ha demostrado que en muchos casos existe una subutilización de esas dependencias [5]. Esa forma poco eficiente de uso de recursos puede impactar significativamente en la mantenibilidad y portabilidad del sistema [6].

En el caso de la tecnología Java, cada paquete o biblioteca está integrado por un conjunto de clases o binarios altamente cohesionados entre sí, que brindan cierta funcionalidad específica y que, en muchos casos, también dependen de otras bibliotecas para su compilación y ejecución. Esta tecnología define que una máquina virtual interprete y ejecute las instrucciones definidas en código intermedio (bytecode), generado durante la compilación. Los enlaces entre recursos (las clases) se establecen a través de referencias simbólicas. Esas referencias son enlazadas dinámicamente en tiempo de ejecución por la máquina virtual y, en función de los recursos disponibles en el class-path (el área de almacenamiento en disco o red donde se ubican todas las clases disponibles), son cargadas a memoria [7]. Esta característica de funcionamiento permitiría postergar la necesidad de contar con el recurso real hasta el momento de ejecución.

Por otro lado, ya está confirmado que para la siguiente versión de la plataforma Java (lanzamiento estimado: julio de 2017) está previsto incorporar un sistema de módulos (proyecto Jigsaw) con el propósito de modularizar internamente la plataforma y, a la vez, ofrecer herramientas para construir software modular [8]. Los principales objetivos del proyecto son: 1) Incrementar la seguridad y mantenibilidad de la plataforma. 2) Optimizar el rendimiento. 3) Facilitar la construcción y mantenimiento de bibliotecas de uso estándar o empresarial. Este sistema de módulos redefine tanto la plataforma como los programas creados con ella, pasando de una arquitectura mono-

- *M. Agüero es Docente del Departamento de Ingeniería en Sistemas de Información de UTN FRBA y está desarrollando su trabajo de Tesis de Maestría de UNLP. E-mail: maguero@frba.utn.edu.ar*
- *L. Ballejos es Docente Investigador del CIDISI de UTN FRSF. E-mail: lballejo@frsf.utn.edu.ar*

978-1-5386-3057-0/17/$31.00 ©2017 IEEE

lítica a una integrada por módulos. A diferencia del sistema de bibliotecas Java tradicionales (archivos Jar), en esta nueva versión, los Jars modulares deben incorporar de forma obligatoria un descriptor de módulo, donde se definen recursos provistos y requeridos. Otra nueva funcionalidad, que es parte del proyecto Jigsaw y también se sumará a Java 9, será la posibilidad de crear imágenes modulares de tiempo de ejecución (modular run-time images) [9].

En el ámbito de la modularización del software, existe desde hace varios años la tecnología OSGi (Módulos Dinámicos para Java). Se lanzó en el año 1999 como una iniciativa de un consorcio de fabricantes de dispositivos para domótica y que luego fue adaptada para crear desde aplicaciones de escritorio (por ejemplo, el IDE Eclipse), hasta servicios en la nube o soluciones IoT [10]. Esta tecnología también propone subdividir el software Java en módulos y establece la presencia de un ambiente de ejecución controlado que además gestione el ciclo de vida y un registro de servicios para esos módulos. Si bien en un principio OSGi generó gran entusiasmo, problemas de complejidad y dificultades de productividad impidieron que sea masivamente adoptada por la comunidad de software [11].

Actualmente es válido suponer que la tendencia surgida a partir de la gran diversidad de ambientes de ejecución (dispositivos IoT o cómputo en la nube), es fragmentar o modularizar el software. Ya sea por madurez de la tecnología, o por metas de productividad, el foco de la industria está puesto en la personalización de los sistemas productivos. Si bien la organización del software Java en bibliotecas brinda un cierto grado de división por funcionalidad, en muchos casos es posible que no esté ofreciendo el nivel de granularidad que la industria está necesitando.

Por otro lado, y en el ámbito del ambiente donde se construye el software, el modelo de gestión de dependencias, heredado del modelo de gestión de paquetes de sistemas operativos, posiblemente ya sea obsoleto [12]. Está inspirado en un diseño de los años '90, donde el cliente debe concentrar los datos del proyecto, la ubicación de los repositorios, y contar con los algoritmos y estrategias adecuadas para conseguir la resolución de esas dependencias. Además, este modelo también depende de la exactitud y completitud de los metadatos asociados a esas bibliotecas de los repositorios.

Este trabajo se orienta en distinguir y proponer soluciones para los siguientes inconvenientes derivados del modo en el que se están gestionando actualmente las dependencias del software Java: **Alto acoplamiento:** Estudios revelan que casi en el 40% de los casos donde falla la construcción de un proyecto que emplea un Gestor de Dependencias, el problema está relacionado con la resolución de dependencias [12]. **Conflictos entre bibliotecas:** Es habitual que existan incompatibilidades entre bibliotecas, por eso el sistema de gestión de dependencias debe contar con suficiente capacidad analítica para evitar esos conflictos y no depender de la intervención humana [13] [14].

A continuación, en la siguiente sección se describe el contexto tecnológico donde se desarrolla este trabajo. Luego, en la Sección 2 se explican los antecedentes que dan fundamento a este proyecto. En la Sección 3 se presenta una propuesta conceptual. En la Sección 4 se explican las características de diseño e implementación de un prototipo. Luego, en la Sección 5 se desarrolla un caso de estudio. En la Sección 6 se desarrolla un relevamiento de otros gestores de dependencias/paquetes y, finalmente, en la Sección 7 se presentan las conclusiones y trabajos futuros.

2 ANTECEDENTES

El éxito de una plataforma de software está, en gran parte, ligado a la disponibilidad masiva de bibliotecas de acceso libre. Para el caso del software Java, lo habitual es distribuir el binario (bytecode) empaquetado en archivos de bibliotecas de extensión Jar. Esta modalidad de distribución en bibliotecas no establece como obligatoria la presencia de un manifiesto. Esto es optativo, y puede contener metadatos tales como: firma electrónica, control de versiones, declaración de dependencias o el punto de entrada (Main-Class). Para el caso de los bundles OSGi, que también son archivos Jar, el manifiesto sí es obligatorio. Allí se declara el nombre del bundle, identificador, versión, dependencias e interfaces.

Con el propósito de dar soporte a la construcción de proyectos Java de gran escala, han surgido herramientas como Apache Maven, Ivy o Gradle. Maven, la más difundida de la actualidad, libera su primera versión en el año 2004 y propone cubrir todo el ciclo de vida de un proyecto Java [15]. Es una solución que da soporte a nivel proyecto, dependencias y construcción del software. Establece un modelo de descripción genérica del proyecto, a través de un archivo POM (Project Object Model), donde se especifican las dependencias, jerarquías, ciclo de vida de la construcción, parámetros de configuración, casos de prueba, repositorios [16]. A medida que se obtienen bibliotecas de un repositorio remoto, esta herramienta crea un repositorio local donde se copian todos los archivos Jar requeridos por el proyecto. De este modo, los proyectos apuntan sus dependencias a la copia única, pudiendo establecer políticas a nivel particular o departamental, donde un grupo de desarrolladores comparte un repositorio único.

A simple vista, este modelo parece ideal. No obstante, tanto la industria como la academia han detectado problemas en él. Uno de ellos es el denominado "Jar Hell" (o infierno de Jars), que tiene lugar cuando en un mismo class-path conviven clases que comparten un mismo nombre, o bien, cuando distintas versiones de una clase deben coexistir en un mismo class-path. Otro problema frecuente es la necesidad de contar también con dependencias transitivas y con de la versión apropiada [13]. Si bien Maven y otras herramientas similares proponen perfiles de compilación o bien, buscan automatizar la descarga de las dependencias transitivas, esta solución implica definir una serie de especificaciones no inherentes al proyecto, además de requerir recuperar del repositorio central la totalidad de las dependencias directas e indirec-

tas [15]. Otro inconveniente que presenta este modelo es la subutilización de recursos de bibliotecas [5] [6].

2.1 Gestión de Bibliotecas con Maven

Maven provee y gestiona dependencias mediante la copia local de las bibliotecas directas e indirectas. En este caso (Fig. 1), se muestra una vista jerárquica de las bibliotecas (o artefactos) requeridas para compilar y ejecutar un ejemplo introductorio de GeoTools.

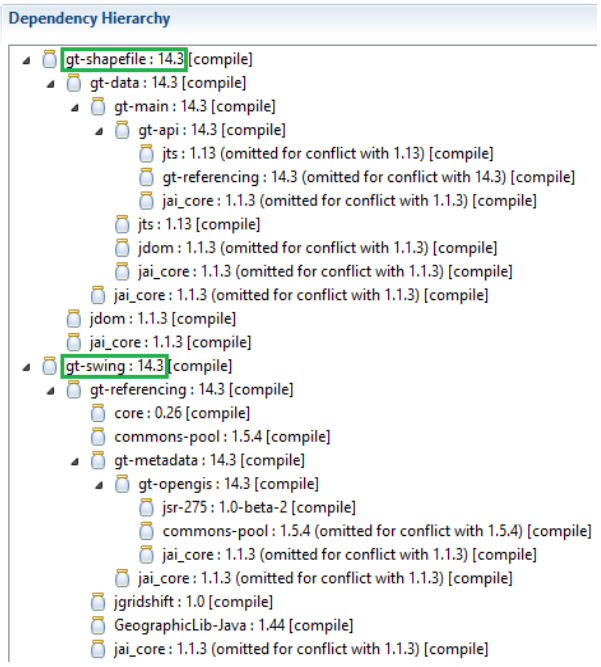


Fig. 1. Dependencias directas y transitivas requeridas por la clase Quickstart de Geotools.

Esta vista parcial muestra el listado de bibliotecas directas (gt-shapefile y gt-swing) y las indirectas (las transitivas).

Al definir en el archivo pom.xml las 2 dependencias directas, Maven incorpora a través del análisis de los metadatos de cada artefacto, sus dependencias transitivas. Esto resulta en la descarga local de 57 bibliotecas adicionales para poder compilar y ejecutar el programa de 16 líneas de código que se muestra en la Fig. 2.

```
public class Quickstart {
  public static void main(String[] args) throws Exception {
     File file = JFileDataStoreChooser.showOpenFile("shp", null);
     if (file == null) {
         return;
     }
     FileDataStore store = FileDataStoreFinder.getDataStore(file);
     SimpleFeatureSource featureSource = store.getFeatureSource();
     MapContent map = new MapContent();
     map.setTitle("Quickstart");
     Style style = SLD.createSimpleStyle(featureSource.getSchema());
     Layer layer = new FeatureLayer(featureSource, style);
     map.addLayer(layer);
      JMapFrame.showMap(map);
  }
}
```

Fig. 2. Código fuente de la clase Quickstart.

Para conseguir la jerarquía presentada en la Fig. 1, primero se debe configurar el archivo descriptor (pom.xml) especificando los repositorios y las dependencias directas, como muestra la Fig. 3.

Esta forma de establecer la fuente de dependencias trae

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>demo</groupId>
    <artifactId>Quickstart</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <repositories>
        <repository>
            <id>maven2-repository.dev.java.net</id>
            <name>Java.net repository</name>
            <url>http://download.java.net/maven/2</url>
        </repository>
        <repository>
            <id>osgeo</id>
            <name>Open Source Geospatial Foundation Repository</name>
            <url>http://download.osgeo.org/webdav/geotools/</url>
        </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>org.geotools</groupId>
            <artifactId>gt-shapefile</artifactId>
            <version>14.3</version>
        </dependency>
        <dependency>
            <groupId>org.geotools</groupId>
            <artifactId>gt-swing</artifactId>
            <version>14.3</version>
        </dependency>
    </dependencies>
</project>
```

Fig. 3. Archivo pom.xml descriptor de proyecto Quickstart con Maven.

como consecuencia un **Alto acoplamiento:** Este modelo establece que sea el cliente, a través del descriptor de proyecto, quien localice las bibliotecas en las fuentes declaradas (los repositorios), delegando en el ambiente de programación la necesidad de contar con esas direcciones. Sulir [12] revela que casi el 40% de los casos donde falla la construcción de un proyecto Maven, el problema está relacionado con la resolución de dependencias. También genera **Conflictos entre bibliotecas:** Es habitual que existan problemas durante la resolución de dependencias [13]. El sistema de gestión de dependencias debe contar con suficiente capacidad analítica para evitar estos conflictos [14]. Además, es imprescindible que los metadatos estén completos, incluyendo casos especiales de incompatibilidad que pueden surgir a partir del enlace dinámico de Java [26].

3 Propuesta

Según la version preliminar de la JLS (Especificación del Lenguaje Java) para la versión 9, cada módulo incluirá un descriptor: un archivo de nombre fijo 'module-info.java' que será compilado junto con todo el código fuente de módulo [17]. En ese descriptor se debe indicar: Nombre del módulo, módulos que requiere, paquetes que exporta, servicios que provee, servicios que consume y otros. El Sistema de Módulos para la Plataforma Java (JPMS) establece que por cada módulo se deben definir los módulos que requiere en forma directa. Al igual que con las bibliotecas, también existe la transitividad de dependencias en Java 9. Éstas pueden no estar presentes al momento de compilación, pero sí durante la ejecución.

Tanto Maven como Gradle, dan soporte a la gestión de dependencias mediante una arquitectura centrada en los datos, donde varios clientes acceden a uno o más repositorios. Ambas soluciones se basan en la instalación local de software, a través de la cual se accede a los repositorios definidos en descriptores propietarios de cada proyecto. En el caso de Maven, el archivo pom.xml y para Gradle, el archivo build.gradle, escrito con un DSL propietario. Según la información difundida a través de conferencias[12], para Java 9 está previsto que las dos herramientas

[1] Maven: http://www.slideshare.net/RobertScholte/java-9-and-the-impact-on-maven-projects

mantengan esta arquitectura e integración directa entre el ambiente de desarrollo y los repositorios de bibliotecas/módulos (Fig. 4).

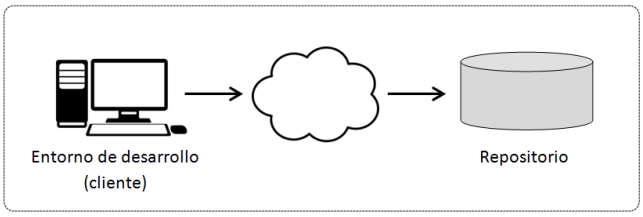


Fig. 4. Arquitectura General de los Gestores de Dependencias.

Con el propósito de actualizar este modo de resolver dependencias, se considera diseñar una solución basada en una arquitectura orientada a servicios integrada por un cliente liviano, que sólo se encargue de transmitir los nombres de los módulos directos del proyecto en desarrollo, y un servicio en la nube que concentre el conocimiento para resolver la clausura y ubicación esas dependencias (Fig. 5).

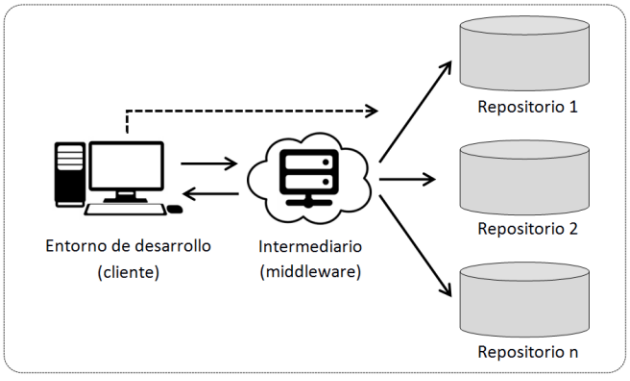


Fig. 5. Arquitectura General de esta nueva propuesta.

De este modo, se desacopla el ambiente de desarrollo de los repositorios, siendo el intermediario o middleware la entidad que determina la ubicación de los módulos. Esta actualización propone un cliente liviano, cuya única función sea interactuar con el usuario y enviar las solicitudes de módulos al servicio en la nube.

Como en Java 9 cada módulo estará integrado por un descriptor obligatorio, de esta manera se evita agregar un descriptor externo, tal como contemplan Maven, Ivy y Gradle. Con el archivo 'module-info' es suficiente para conocer las dependencias del módulo, sólo con analizar el campo 'requires'.

Clausura: Con la información disponible en ese descriptor de módulo, se conocen las referencias a nivel módulo. De este modo, no es necesario evaluar en el bytecode las referencias simbólicas entre clases. Como se representa en la Fig. 6, se propone que un algoritmo itere las referencias a módulos definidas en el campo "requires", hasta alcanzar la clausura o el nivel de profundidad definido en configuración. Para conocer las dependencias de cada módulo, se descompila el descriptor con el programa javap (que es parte del JDK).

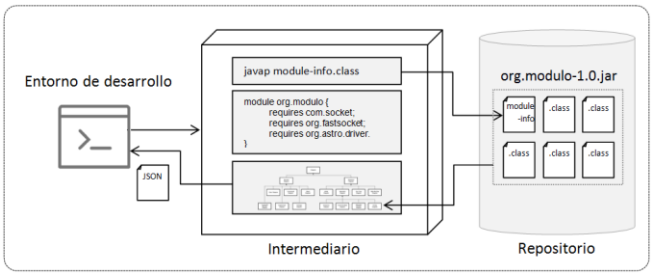


Fig. 6. Interacción Cliente – Intermediario – Repositorio.

En relación al uso del programa javap, se aclara que jdeps sería la opción estándar del JDK para analizar dependencias de un módulo. No obstante, este jdeps requiere la presencia del archivo Jar en una ubicación local y, copiarlo desde el repositorio al intermediario, penalizaría significativamente el tiempo de respuesta. Otra alternativa es usar el argumento print-module-descriptor para el programa jar.

Versiones: Si bien OSGi maneja versiones de módulos, el grupo de expertos responsable de Java 9 prefirió omitir dar soporte a esta característica [18]. Una de las premisas de esta nueva versión, es la de mantener la simplicidad. Por ese motivo, y para evitar una transición más compleja, queda bajo la responsabilidad de las herramientas de construcción y contenedores de aplicaciones [19].

Repositorios: Del mismo modo que en la versión anterior del prototipo, para localizar y copiar archivos de un Jar remoto, se emplea el subproyecto Remote Zip[3], que permite obtener únicamente el descriptor de módulo antes de efectuar la copia completa. Así, sólo transfiere el archivo module-info desde el repositorio al intermediario. Para simplificar la selección dinámica de repositorios, esta solución siempre evalúa primero el nombre de archivo (de donde obtiene nombre del módulo y versión) y el descriptor module-info de cada módulo, no los archivos de metadatos.

Interfaz con el usuario: Una solución basada en un cliente pesado, tal como se distribuyen las dos herramientas de gestión de proyectos Java más difundidas, es desactualizada y difícil de mantener. Para esta evolución del prototipo se propone, en una primera fase, ofrecer una interfaz hombre-máquina a través de la línea de comandos. Esto permite tener el mayor grado de control posible sobre la aplicación cliente, conformada por una clase Java muy pequeña, cuya única misión es la de cargar la clase remota de la aplicación (Fig. 7).

[2] Gradle: https://gradle.org/uncategorized/java-components-solving-the-puzzle-with-jigsaw-and-gradle/

[3] https://github.com/martinaguero/remotezip

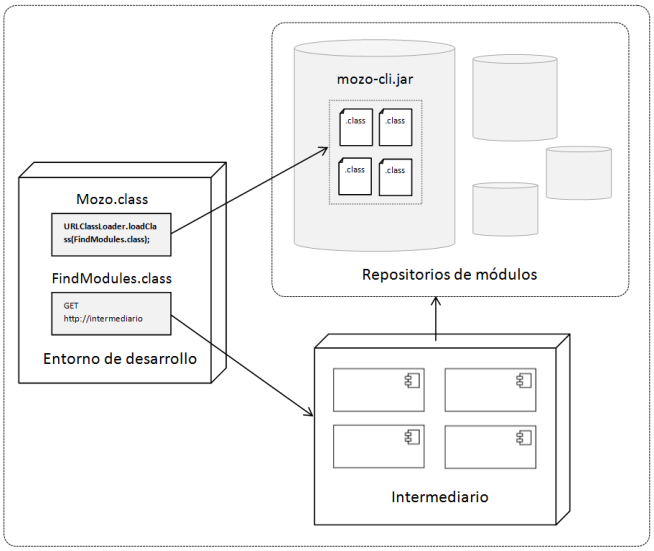


Fig. 7. Carga remota de clases para el cliente.

A medida que el usuario activa diferentes funcionalidades, las clases se cargan a demanda. De este modo, es posible actualizar gran parte de la aplicación cliente sin que el usuario deba realizar ninguna acción.

4 PROTOTIPO

Como el objetivo principal de este proyecto es migrar la resolución de dependencias desde el entorno de desarrollo local a un servicio especializado en la nube, esta versión también centraliza esa funcionalidad en un intermediario entre el cliente y los repositorios de bibliotecas. Para este caso, en lugar de emplear tecnología OSGi, se decidió desarrollar una solución orientada a módulos Java, favoreciendo la simplicidad general. En esta versión, también se empleó el componente que permite extraer archivos de un Zip remoto. La interfaz con los clientes es un componente REST basado en la plataforma Apollo de Spotify [20].

La funcionalidad principal es analizar las dependencias o requerimientos de módulos Java 9, e indicar al cliente dónde obtenerlos. De forma recursiva, analiza el descriptor de cada módulo y localiza los requeridos hasta conseguir la clausura. Para agilizar esta acción, los descriptores son obtenidos remotamente desde los repositorios a través de Remote Zip. De esta manera, sólo se transfiere el archivo module-info.class desde el repositorio al middleware.

Una vez copiado el descriptor, éste se descompila con el programa javap (versión 9) y los módulos que son parte del listado "requires" son sumados a la jerarquía que define la clausura. Este algoritmo se puede representar con el diagrama de la Fig. 8.

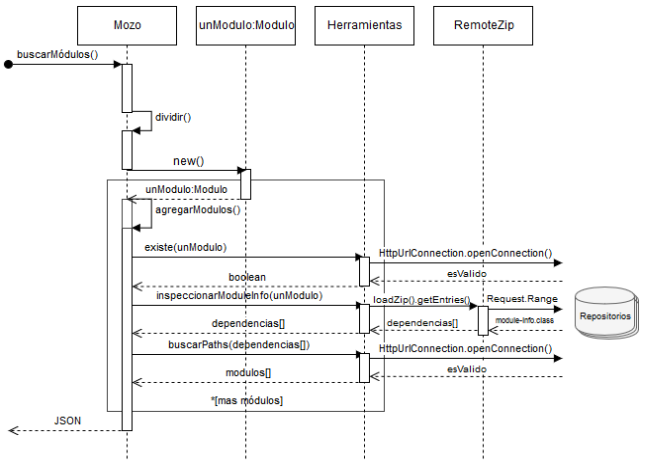


Fig. 8. Secuencia de acciones para localizar módulos.

El resultado para una solicitud de resolución de dependencias es una estructura en formato JSON donde los módulos indicados en la solicitud del usuario integran la raíz, con sus respectivas ubicaciones y los demás módulos requeridos en sus descriptores.

4.1 Implementación

Para desarrollar este prototipo, en una primera instancia se evaluó utilizar la versión de acceso temprano (EA o early access) de Java 9 [21]. Si bien es posible compilar y ejecutar clases y módulos de prueba desde un editor básico como vi, los entornos de desarrollo integrados (IDE) como Eclipse, NetBeans o IntelliJ aún no son compatibles. Según diferentes blogs o las webs de los fabricantes, existen versiones que funcionan con builds anteriores de JDK 9 EA. No obstante, la disponibilidad al momento de hacer este análisis (151) no funcionaba ni para Eclipse ni para IntelliJ. Durante las pruebas funcionó con NetBeans y era posible crear y ejecutar módulos, pero también fue descartado, porque su compatibilidad aún es limitada y, además, es muy inestable [22].

El prototipo desarrollado como propuesta se llama Mozo y está disponible en el repositorio del proyecto: https://github.com/martinaguero/mozo.

4.2 Interfaz con el Usuario

Para usar el prototipo, se debe descargar la clase Mozo.class disponible en **http://trimatek.org/mozo/Mozo.class**

El cliente se ejecuta desde la interfaz de comandos con: **java -cp . Mozo**

Una vez que la consola muestra el prompt mozo>, con Enter, imprime la ayuda en línea. Los comandos disponibles son:

find-modules [lista de nombres de módulos separados por coma]: genera una solicitud GET al servicio intermediario y guarda el resultado en una variable res0, res1, resN.

download-modules [nombre de la variable con el resultado de buscar módulos]: Con el resultado del comando find-modules, se descargan los módulos que son parte

del JSON guardado en una variable de resultados (res0, res1, resN).

print [nombre de la variable con el resultado de buscar módulos]: Imprime por pantalla el contenido de una variable de resultados.

list-modules [nombre de la variable con el resultado de buscar módulos]: Lista sin repeticiones los módulos que son parte de un resultado.

5 Caso de Estudio

Se diseñó un caso donde los módulos que conforman un sistema son com.stats.cli, com.stats.core, com.google.guava, org.apache.math, org.apache.rng (Fig. 9).

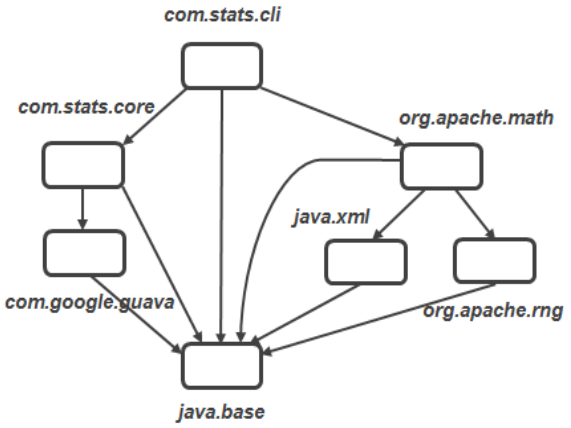


Fig. 9. Módulo com.stats.cli y sus dependencias.

En este caso, el módulo com.google.guava es requerido transitivamente por com.stats.core, también hay transitividad entre com.stats.cli y org.apache.rng, por org.apache.math. A su vez, todos los módulos requieren al módulo java.base y sólo org.apache.math a java.xml, que también es parte de la biblioteca estándar de módulos de Java. El objetivo es contar en el entorno local con el programa Stats, cuya interfaz con el usuario está en el módulo Stats CLI. Para conseguir este módulo junto con sus dependencias, las acciones que el usuario/programador debe ejecutar son:

1. Descargar el archivo Mozo.class de:
http://trimatek.org/mozo/Mozo.class
2. Ejecutarlo desde la interfaz de comandos con:
$>java -cp . Mozo
3. En la consola de comandos de Mozo (prompt), el usuario ingresa:
mozo> find-modules com.stats.cli

Entonces, el cliente cargará la clase FindModules, que pasará la solicitud al intermediario. Una vez que finaliza de evaluar los módulos requeridos y verificar su ubicación, responde al cliente con un JSON de la estructura.

4. El resultado generado por el intermediario se muestra en la Fig. 10.

```
GET request: http://trimatek.org:8080/mozo/find?modules=com.stats.cli
Response code: 200
{
 "from": "user-request: /mozo/find?modules=com.stats.cli",
 "requires": [
  {
   "module": "com.stats.cli",
   "path": "http://www.trimatek.org/repository/com.stats.cli.jar",
   "requires": [
    {
     "module": "com.stats.core",
     "path": "http://www.trimatek.org/repository/com.stats.core.jar",
     "requires": [
      {
       "module": "com.google.guava",
       "path": "http://www.trimatek.org/repository/com.google.guava.jar"
      }
     ]
    },
    {
     "module": "org.apache.math",
     "path": "http://www.trimatek.org/repository/org.apache.math.jar",
     "requires": [
      {
       "module": "org.apache.rng",
       "path": "http://www.trimatek.org/repository/org.apache.rng.jar"
      }
     ]
    }
   ]
  }
 ]
}

result stored in: res0
```

Fig.10. Respuesta del Intermediario[4].

5. Por último, el usuario ingresa el comando:
download-modules res0

Entonces, la clase del cliente en ejecución (Mozo.class) cargará remotamente la clase Download disponible en http://trimatek.org/mozo/org.trimatek.mozo.cli.jar., que iterará por los nodos del árbol donde esté presente el campo path, descargando desde esa fuente el archivo de Jar modular a la ubicación local donde se ejecuta Mozo.class.

6 Herramientas Relacionadas

La gestión de dependencias de software actualmente está soportada por una variedad de soluciones. Según la tecnología o el lenguaje de programación, existen diferentes herramientas que asisten al desarrollador para establecer una línea base de bibliotecas, a partir de la cual se crea nuevo software. Esas bibliotecas, en muchos casos, están configuradas bajo complejas reglas donde se busca cumplir con combinaciones entre versiones de versiones específicas, o evitar conflictos relacionados con incompatibilidades entre sí. Otro caso habitual es cuando se debe definir una fuente particular para ciertas bibliotecas, o cuando una fue eliminada del repositorio estándar.

Los Gestores de Dependencias (también llamados gestores de paquetes a nivel aplicación) están basados en el modelo de Gestor de Paquetes, surgido en el ámbito de la comunidad Linux, cuyas implementaciones más difundidas son RPM para RedHat y APT para Debian. La última implementación de Gestor de Paquetes para Windows es denominado PackageManagement (anteriormente OneGet) e incorpora como novedad ser una agregación de Gestores de Paquetes [23].

[4] Este resultado no incluye a los módulos java.base y java.xml porque son parte del JDK.

A continuación, se describirá una breve reseña de los Gestores de Dependencias más empleados en el ámbito del desarrollo de software.

Bundler (Ruby): Es el Gestor de Dependencias más utilizado para crear software con Ruby. Al igual que otros Gestores de Paquetes, se instala localmente y asiste al programador para administrar y recuperar bibliotecas. Ruby define que las bibliotecas (gemas) deben incluir un descriptor donde se definen, entre otros datos, su ubicación (el repositorio) donde estará la gema y sus dependencias.

Pip (Python): Es la solución de gestión de paquetes de Python. Del mismo modo que otras herramientas similares, se instala en el entorno local y permite descargar bibliotecas de repositorios remotos. Tiene como cualidad particular la presencia de archivos de requerimientos, que son un conjunto de bibliotecas a instalar. Estos archivos de requerimientos son empleados como: 1. Imagen de ambiente como para repetir una plataforma de bibliotecas. 2. Conjunto de reglas para resolver dependencias. 3. Forzar la instalación de una biblioteca alternativa. 4. Reemplazar una dependencia remota por una local. También existen los archivos de restricciones que controlan qué versión de un archivo de requerimientos son instalados.

NuGet (.Net): Es un Gestor de Paquetes orientado al desarrollo con la plataforma .Net de Microsoft. Se ofrece como cliente de línea de comandos, o como plug-in del entorno de desarrollo Visual Studio. Permite instalar paquetes disponibles en el repositorio NuGet Gallery y también agregar repositorios de terceros. Al igual que otros gestores de paquetes, resuelve y descarga dependencias transitivas. En la última versión del cliente, incorpora un archivo de configuración de proyecto donde se declaran las dependencias directas junto con su version.

Composer (PHP): Se define como un Gestor de Dependencias y no un Gestor de Paquetes, diferenciándose porque no instala bibliotecas, sino que las recupera y asocia a proyectos de software. Se instala localmente como una aplicación de línea de comandos con alcance a nivel sistema o nivel proyecto. Permite definir las dependencias a través de un archivo de formato JSON y agregar repositorios públicos y privados. Presenta la capacidad de agregar autocarga de clases (autoload) de bibliotecas. Cuando se activa esta funcionalidad, Composer descargará a demanda las clases referenciadas en el código fuente.

Bower (JavaScript): Surge como un proyecto de Twitter para asistir la gestión de recursos de la presentación (front end) del software. Se instala como una aplicación local y permite descargar paquetes de diferentes fuentes. Bower considera como paquete tanto las bibliotecas, como programas individuales como jQuery o AngularJS. Centraliza la configuración de un proyecto a través de del manifiesto bower.json. Se complementa con otro gestor de paquetes para JavaScript, denominado nmp. Permite agregar otros repositorios a través de Resolvers.

sbt (Scala): El lenguaje de programación Scala funciona en la cima de la máquina virtual de Java, por lo tanto, puede utilizar las bibliotecas Java o Scala indistintamente. Sbt o simple build tool es la herramienta de soporte a la construcción de proyectos Scala de mayor difusión. Delega la gestión de dependencias en Apache Ivy, que junto con Apache Maven y Gradle son las tres herramientas para gestión de dependencias Java más empleadas en la actualidad (1: Ivy sólo es un gestor de dependencias). Sbt configura las dependencias desde un lenguaje específico del dominio (DSL) o desde un archivo POM de Maven. También permite especificar configuraciones a través de un XML de Ivy.

Apache Ivy (Java): Es una herramienta que se especializa en la gestión de dependencias Java. También ofrece resolución automática de dependencias transitivas y, a diferencia de Maven, en algunos casos permite seleccionar qué dependencias transitivas se deberán descargar. Ofrece un mecanismo de resolución de conflictos que permite asociar un gestor de conflictos para cada caso particular.

Maven (Java): Es una solución integrada para la gestión de proyectos Java. Además de dar soporte integral a la gestión del proyecto, es una herramienta de soporte a la construcción de software y un gestor de dependencias. Desde un archivo de configuración de proyecto, organiza las diferentes actividades relacionadas al proyecto, a lo largo de todo su ciclo de vida. La gestión de dependencias es una de sus principales funcionalidades. Su implementación está basada en el modelo de gestores de paquetes, permitiendo la resolución tanto de las dependencias directas, como las transitivas. Si bien es la solución de gestión de proyectos más utilizada por la comunidad Java, últimamente ha perdido una importante cantidad de usuarios, a partir de la aparición de Gradle.

Gradle (Java): Se presenta como una solución superadora a Maven y Ant que ha tomado como base conceptual a esas herramientas. Al igual que Maven, también es un gestor de dependencias, que además de dar soporte a la resolución de dependencias directas y transitivas, asocia las bibliotecas del caché local a su repositorio de origen, favoreciendo la repetividad de la construcción en otro ambiente. Como herramienta de construcción, se diferencia de Maven, principalmente, por saber distinguir puntualmente qué módulo debe ser reconstruido. Gradle considera a cada módulo como un proyecto en sí, definiendo configuraciones particulares para cada uno. Esto deriva en una mayor escalabilidad, permitiendo fragmentar el proyecto en diferentes unidades de compilación. Otra particularidad es que en lugar de usar XML o JSON para configurar el proyecto, lo hace a través de un DSL, ganando en expresividad y simplicidad.

GNU Guix (Linux): La comunidad GNU está desarrollando un Gestor de Paquetes de nueva generación. Se presenta como un gestor de paquetes puramente funcional. Tiene como característica original que permite interactuar con repositorios, programar construcciones y declarar paquetes a través de un DSL funcional. El cliente se instala localmente como un gestor de paquetes tradicional, y se destaca por ofrecer gestión transaccional de paquetes [24].

Spack (HPC): La computación de altas prestaciones (HPC) tiene sus particularidades respecto de la gestión de

dependencias. Como habitualmente son ambientes conformados por gran variedad de arquitecturas y sistemas operativos, conseguir la clausura de las dependencias presenta un desafío adicional. En algunos casos, la solución consiste en unificar plataformas a través de máquinas virtuales. No obstante, es una solución que puede impactar significativamente en el rendimiento. Con el propósito de crear una herramienta a escala de estos requerimientos, el Laboratorio Lawrence Livermore recientemente desarrolló la herramienta Spack, con el fin de unificar y simplificar la gestión de dependencias en la construcción de software [25]. Spack se presenta como una solución que considera paquete a un guión (o script) de construcción. Dada la gran variedad de arquitecturas, trabajar con bibliotecas compiladas no es lo suficientemente flexible, por lo tanto, cada vez que un componente de software es referenciado, se ejecuta una construcción bajo ciertas condiciones de versionado, arquitectura y ambiente de ejecución. Al igual que sbt, Gralde y GUIX, Spack emplea un DSL para describir los proyectos (o paquetes). Durante el proceso de clausura de dependencias, esta solución genera y valida un grafo acíclico dirigido (DAG) para garantizar una construcción consistente. Otra particularidad de este sistema son las interfaces virtuales entre dependencias: una Dependencia Virtual es un nombre abstracto que representa una interfaz (o capacidad) de biblioteca en lugar de un nombre de biblioteca. De este modo, los paquetes que necesitan esa interfaz no dependen de una implementación específica. Como Spack no trabaja con bibliotecas compiladas, sino con scripts que construyen esas bibliotecas, el proceso de validación del modelo representado por el DAG se produce durante la fase de concretización del modelo. En esa instancia se evalúa la consistencia del modelo a partir de las características de cada nodo (biblioteca) del grafo. Una vez finalizada la validación del plan, el algoritmo construye e instala cada biblioteca del DAG en orden bottom-up.

6.1 Comparación

Básicamente, todos estos gestores de paquetes descriptos previamente son aplicaciones de ejecución local que analizan las dependencias asociadas a cada paquete y, recursivamente, descargan los requeridos para alcanzar la clausura. La portabilidad del modelo "Gestor de Paquetes" al ambiente de desarrollo puede estar combinada con herramientas de construcción de software (build tools), donde se suma configuración específica de compilación al descriptor de proyecto. Además se puede mencionar que la tendencia apunta a emplear lenguajes específicos del dominio (DSL) para describir paquetes/bibliotecas o proyectos. Otra característica común es que todas estas herramientas ofrecen la posibilidad de agregar más de un repositorio como fuente de bibliotecas. En todos los casos siempre existe uno por omisión (default) y se permite configurar alternativas. A medida que la complejidad y tamaño del software se incrementa, el proceso de construcción es cada vez más abarcativo, por lo tanto, resulta imprescindible poder fragmentar un proyecto en subproyectos con el fin de evitar construcciones masivas en cascada. Gradle ofrece soporte para esto, siendo una cualidad muy valorada en la comunidad del software. Por último, es fundamental destacar la criticidad de la construcción del grafo de dependencias. Tanto Bower como Spack, se diferencian de otras herramientas por dar especial atención a la precisión de los algoritmos que crean y validan estos grafos, evitando repeticiones y ciclos entre bibliotecas.

7 Conclusiones y Trabajos Futuros

La construcción de software a gran escala es un desafío que no dejará de crecer, ni en tamaño, ni complejidad. La resolución de dependencias es una actividad que, al igual que el proceso de construcción, no deberían requerir demasiados recursos, ya que el objetivo es el sistema en desarrollo y no los mecanismos de soporte alrededor de él.

Con esta propuesta se busca dar solución a los problemas de alto acoplamiento entre el cliente-repositorios y minimizar los conflictos entre bibliotecas, para ello y a modo de prueba de concepto, se diseñó e implementó el prototipo de una solución con las siguientes caracterticas particulares: 1) No es una solución invasiva, es decir, no requiere sumar metadatos a los módulos. 2) La resolución de la clausura no depende del listado de fuentes (repositorios) definidos en el archivo descriptor del proyecto. Es el intermediario el que conoce esas fuentes y luego de verificar disponibilidad, establece esa relación en el resultado. 3) El intermediario es un especialista en resolver dependencias, conoce las fuentes y puede, a lo largo de sucesivas versiones, incorporar algoritmos más sofisticados, como por ejemplo herramientas de inteligencia artificial. Desde el punto de vista arquitectónico: 4) El cliente siempre ejecuta la última versión. 5) Las direcciones (paths) a los módulos siempre son verificadas. 6) Los tiempos de respuesta son aceptables, dado que sólo se transfiere al intermediario el archivo descriptor del módulo. 7) La concentración de la solución en un servicio en la nube permite realizar permanentes ampliaciones y correcciones, de forma transparente para el usuario.

En resumen, este trabajo tiene por objetivo actualizar el modo en que se resuelven las dependencias de software Java, cambiando de un modelo centrado en: meta-datos, estación de trabajo y repositorios, a un servicio especializado en resolver dependencias. Asimismo, también apunta a simplificar las acciones operativas vinculadas a un proyecto de software, de modo tal que el foco del equipo esté en el sistema en desarrollo, y no en las herramientas de soporte.

La siguiente fase del proyecto, y a modo de trabajo futuro, el objetivo es poder transformar el prototipo en un servicio productivo de acceso libre. Para fortalecer su efectividad, la primera ampliación consistiría en sumar más repositorios de módulos. Incluso, no se descarta la posibilidad de publicar uno propio. También está previsto sumar una interfaz web, de modo que no sólo se deba interactuar desde la consola de comandos. El desarrollador ingresaría el nombre del módulo que necesita y el sitio web respondería con el conjunto de direcciones a los archivos Jar que conforman la clausura. De esta forma, se

ofrecería un servicio de resolución de dependencias dirigido a desarrolladores de software principiantes y, así, se evitaría la frustración que muchas veces significa tener que aprender a usar y configurar herramientas complejas de gestión de proyectos.

Una vez disponible la versión estable, el siguiente paso estaría enfocado en medir precisión y tiempos de respuesta.

REFERENCIAS

[1] Spinellis, D., "Package Management Systems", *IEEE Software*, 2012.

[2] Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S., "A modular package manager architecture", *Information and Software Technology*, Special Section: Component-Based Software Engineering (CBSE), 2011.

[3] 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.

[4] Regalado, A., "Who coined 'Cloud Computing'?", *MIT Technology Review*, 2012. Disponible en: https://www.technologyreview.com/s/425970/who-coined-cloud-computing/.

[5] Agüero, M., Ballejos, L., Pons, C., "Deep: Una Herramienta para Medir Dependencias Java", *3er Congreso Nacional de Ingeniería Informática / Sistemas de Información (CoNaIISI)*, 2015.

[6] Wang, P., Yang, J., Tan, L., Kroeger, R., Morgenthaler, D., "Generating precise dependencies for large software", *Proceedings of the 4th International Workshop on Managing Technical Debt*, IEEE Press. 2013.

[7] Liang, S., Bracha, G., "Dynamic Class Loading in the Java Virtual Machine", *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1998.

[8] Compiler Group, "Project Jigsaw", OpenJDK, Oracle Corporation, 2017. Disponible en: http://openjdk.java.net/projects/jigsaw/.

[9] Reinhold, M., "JEP 220: Modular Run-Time Images", OpenJDK, Oracle Corporation, 2016. Disponible en: http://openjdk.java.net/jeps/220/.

[10] Flotyński J., Krysztofiak K., Wilusz D., "Building Modular Middlewares for the Internet of Things with OSGi", *The Future Internet.* FIA 2013. Lecture Notes in Computer Science, vol 7858. Springer, Berlin, Heidelberg, 2013.

[11] McKenzie, C., "Rod Johnson: OSGi is not easy to use. Not as productive as it should be", *The Server Side*, 2011. Disponible en: http://www.theserverside.com/news/thread.tss?thread_id=62523.

[12] Sulír, M., Porubän, J., "A quantitative study of Java software buildability", *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools.* (PLATEAU 2016). ACM, 2016.

[13] Jezek, K., Dietrich, J., "On the use of static analysis to safeward recursive dependency resolution", *40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2014.

[14] Kuniyasu, C., Di Cosmo, R., Treinen, R, Zacchiroli, S., "Why do software packages conflict?", *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12).* 2012.

[15] Varanasi, B., Belida, S., "Introducing Maven", Apress, 1era Edición. 2014.

[16] McIntosh, S., Adams, B., Hassan, A., "The evolution of Java build systems", *Empirical Software Engineering*, Springer, 2012.

[17] Buckley, A., "JPMS: Modules in the Java Language and JVM", Oracle Corporation. Disponible en http://cr.openjdk.java.net/~mr/jigsaw/spec/lang-vm.html.

[18] Bartlett, N., "OSGi and Java 9 Modules Working Together", Disponible en: http://njbartlett.name/2015/11/13/osgi-jigsaw.html

[19] Reinhold, M., "The State of the Module System", OpenJDK. En línea: http://openjdk.java.net/projects/jigsaw/spec/sotms/

[20] "Apollo Spotify", Spotify, 2017. Disponible en http://spotify.github.io/apollo/

[21] "Java 9 Early Access with Project Jigsaw", Oracle Corporation, 2016. Disponible en: https://jdk9.java.net/jigsaw/

[22] "NetBeans JDK 9 Support", NetBeans, 2017. Disponible en: http://wiki.netbeans.org/JDK9Support/

[23] "PackageManagement", Microsoft TechNet, 2015. Disponible en: https://blogs.technet.microsoft.com/packagemanagement/2015/04/28/introducing-packagemanagement-in-windows-10/

[24] "GNU GUIX", Free Software Foundation. 2017. Disponible en: https://www.gnu.org/software/guix/

[25] Gamblin, T., LeGendre, M., Collette, M., Lee, G., Moody, A., de Supinski, B., Futral, S., "The Spack package manager: bringing order to HPC software chaos," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, 2015.

[26] Måhlén, P., "Dealing with Java Linking Problems", Spotify Labs, 2015. Disponible en: https://labs.spotify.com/2015/09/01/java-linking/