View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Servicio de Difusión de la Creación Intelectual

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].

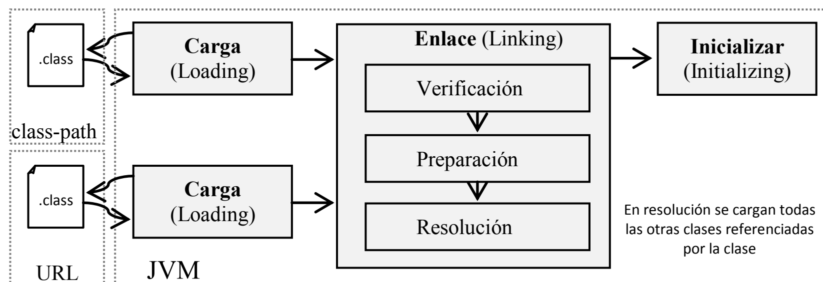

Fig. 1. Proceso de carga de clases en la JVM [12].

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.

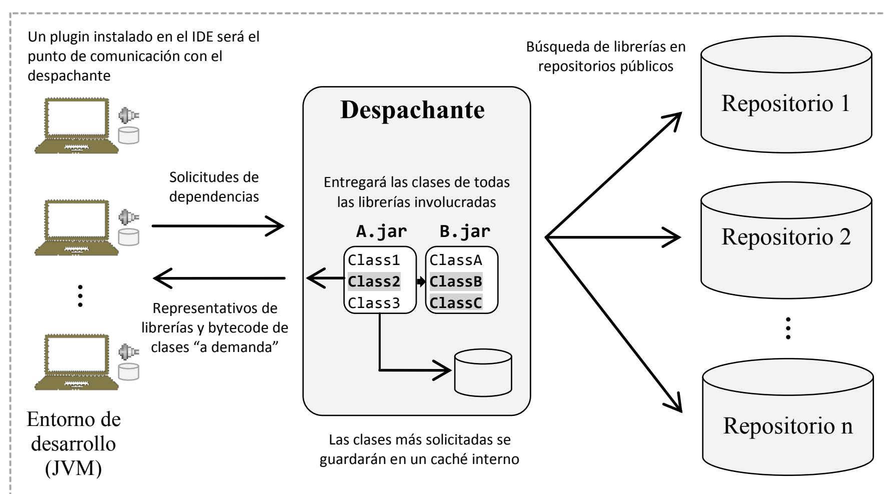

Fig. 2. Intermediario entre entornos de desarrollo (IDE) y repositorios de librerías.

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.

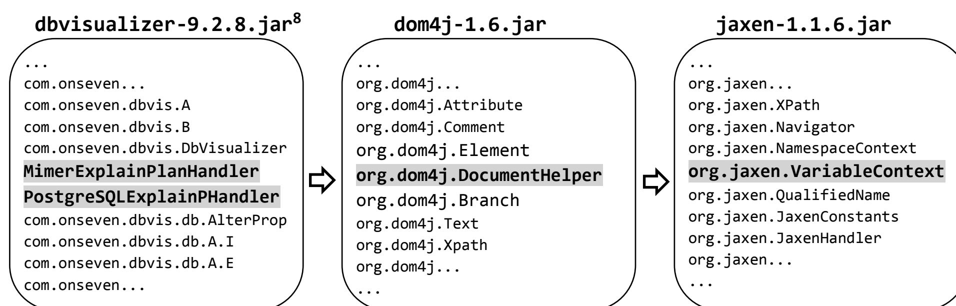

Fig. 3. Resolución de dependencia transitiva.

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)