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].

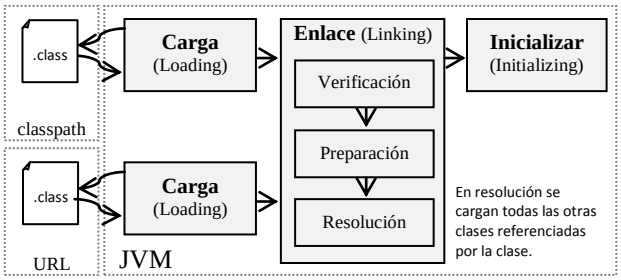

Figura 1. Carga, enlace e inicialización de clases.

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).

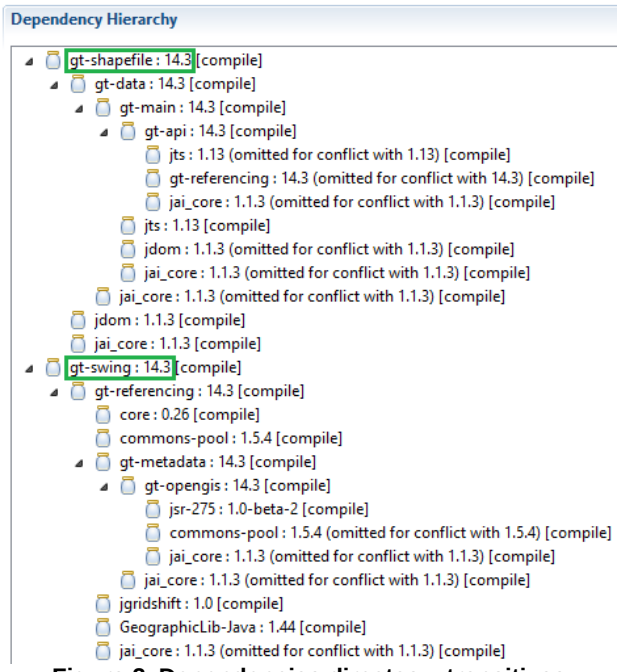

Figura 2. Dependencias directas y transitivas requeridas por el programa Quickstart de GeoTools.

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.

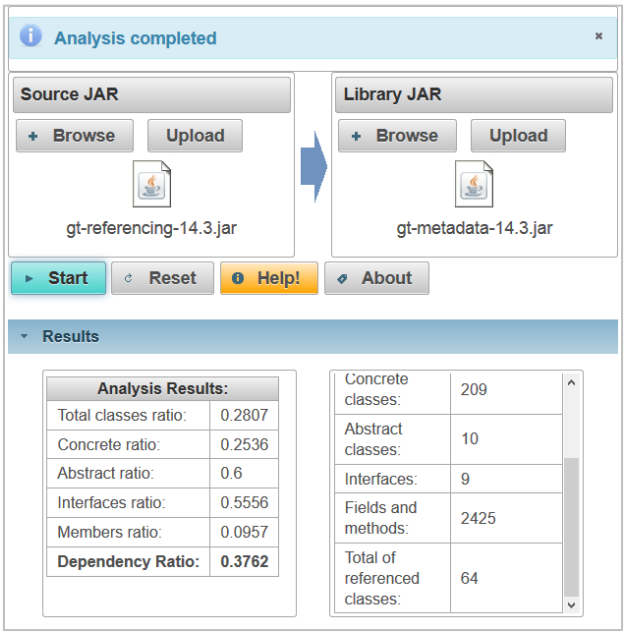

Figura 4. Interfaz gráfica web de la herramienta Deep.

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.

Nivel 0	Nivel 1	Nivel 2	
quickstart	gt-swing (→ 0.0125)	gt-referencing (→ 0.0166)	core (→ 0.1325)
			commons-pool (→ 0.2602)
			gt-metadata (→ 0.3762)
			jgridshift (→ 0.3125)
			geographicLib (→ 0.2288)
		gt-render (→0.1817)	-sin medir -
		miglayout (→0.0714)	
	gt-shapefile (→ 0.0000)	-sin medir-	

[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.

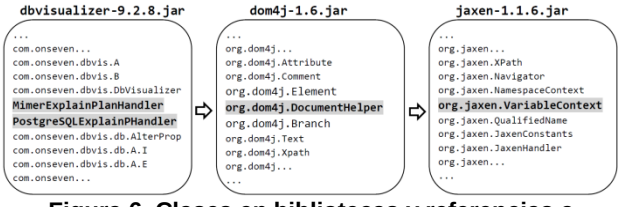

Figura 6. Clases en bibliotecas y referencias a dependencias.

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.

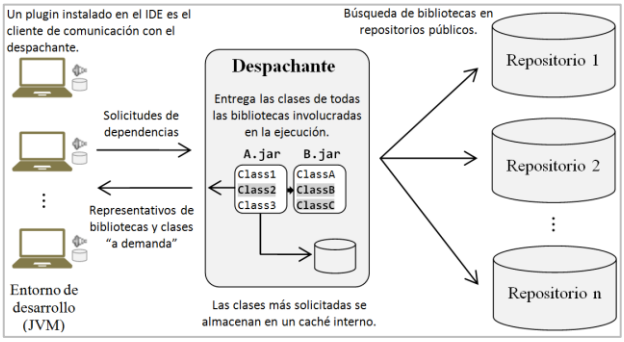

Figura 7. Servicio despachante de clases a demanda.

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.

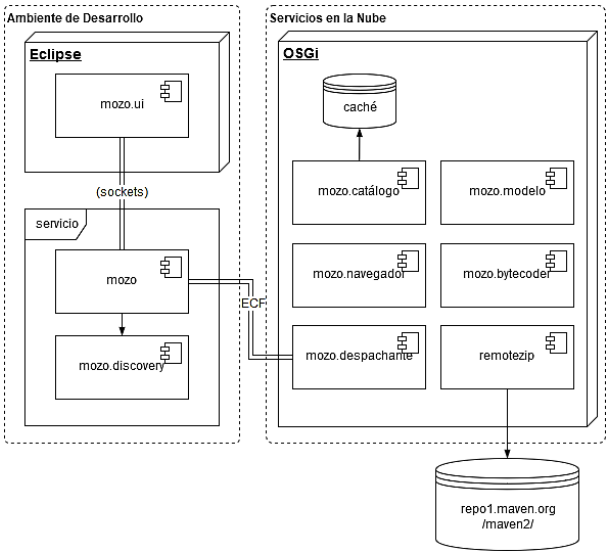

Figura 8. Arquitectura general del prototipo.

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/

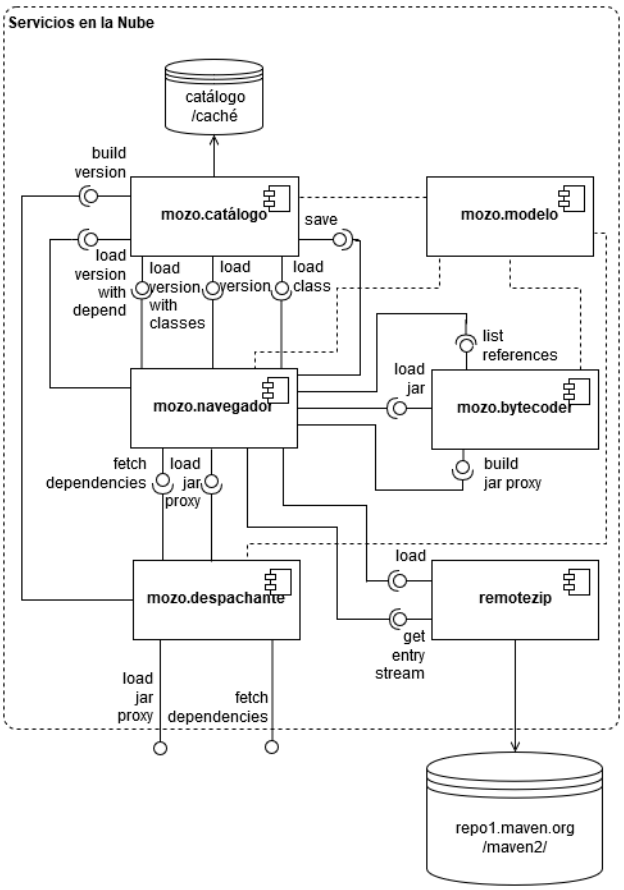

Figura 9. Conjunto de microservicios que conforman el intermediario entre el IDE y los repositorios.

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).

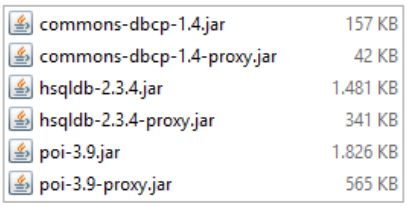

Figura 10. Diferencia de tamaño entre bibliotecas originales y representativos (proxy).

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.

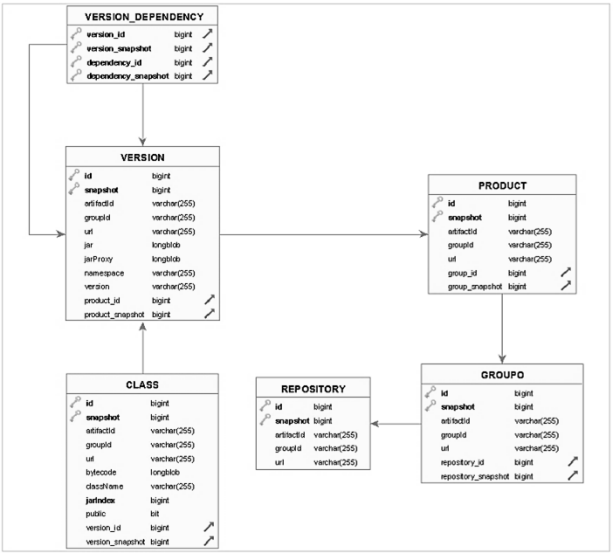

Figura 11. Esquema de persistencia relacional simple de catálogo / caché.

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).

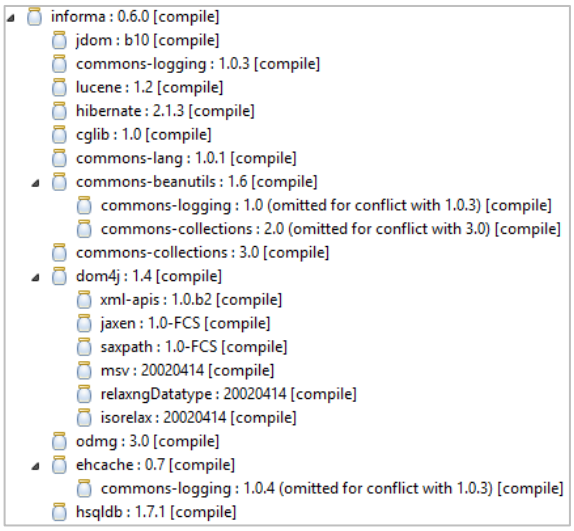

Figura 12. Dependencias de *Informa* 0.6.0 y sus transitivas.

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].

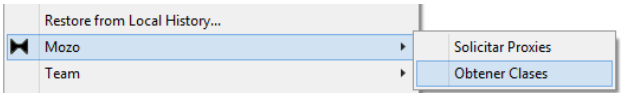

Figura 14. Menú contextual de Eclipse con plugin Mozo.

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.