Servicio para la resolución de dependencias de software basado en componentes

Autor: Lic. Martín Agüero
Directora: Dra. Luciana Ballejos
Codirectora: Dra. Claudia Pons

Tesis presentada para obtener el grado de
Magister en Ingeniería de Software

Facultad de Informática - Universidad Nacional de La Plata

Octubre de 2017

Agradecimientos

A Mis Padres, que aún en momentos de incertidumbre, siempre confiaron en mí. A María Sol, que me acompañó a lo largo de este trayecto con cariño, paciencia y fortaleza. A Uriel Cukierman, quien me guió y alentó genuinamente durante cada fase del postgrado. A Claudia Pons, por sus valiosos consejos que siguen formándome como mejor profesional. A Luciana Ballejos, cuyos desafíos y enseñanzas lograron un resultado superador. A las autoridades de la Universidad Tecnológica Nacional, Facultad Regional Buenos Aires, por darme la oportunidad de desarrollar mi carrera como docente e investigador en la Institución. A las autoridades, docentes y no docentes de la Universidad Nacional de La Plata, Facultad de Informática, por confiar en mi idoneidad para encarar este desafío y darme la oportunidad de incorporar más conocimiento sobre esta apasionante profesión que es la Ingeniería de Software.

Contenidos

Figuras

Tablas

Resumen

El desarrollo de software a escala industrial requiere de infraestructura acorde a los requerimientos de cada proyecto. La comunidad de software se retroalimenta de forma permanente, a través de la reutilización de componentes distribuidos en el formato de bibliotecas o paquetes. Actualmente, los proyectos de software tienden a ser diseñados como una composición de recursos de funcionalidad específica, promoviendo la reutilización y siendo, en muchos casos, un factor clave de éxito, ya sea por calidad probada o integración inmediata de una nueva prestación.

Desde principios de los años ’90, han surgido diferentes herramientas de soporte a la integración y actualización de sistemas operativos por medio de paquetes. Esas herramientas, denominadas Gestores de Paquetes, permiten agregar y quitar, de forma atómica, paquetes de software provenientes de repositorios externos.

La industria del software también incorporó el modelo de distribución por paquetes, definiendo a la biblioteca como un conjunto de elementos de software reutilizables, indivisibles y de alta cohesión. Tomando como base a los Gestores de Paquetes, la comunidad de software desarrolló los Gestores de Dependencias. Estas herramientas interactúan con los repositorios de bibliotecas, asistiendo a los ambientes de producción de software en los procesos de recuperación y clausura de las dependencias. Si bien la adaptación del modelo de Gestor de Paquetes al ámbito industrial ha sido exitosa, padece de una serie de inconvenientes que se abordarán a lo largo de esta Tesis.

A continuación, este trabajo se centrará en describir el contexto tecnológico actual y en aquellos aspectos con potencialidad para mejorar el proceso de gestión de dependencias del software Java. Luego, se presentarán tres prototipos de herramientas, una para medir la proporción de referencias entre bibliotecas y las otras dos, a modo de prueba y validación de conceptos. Finalmente se desarrollará una evaluación comparativa con las herramientas más utilizadas en la actualidad y se presentarán las conclusiones.

En resumen, esta Tesis presenta una alternativa al modo como se está gestionando la resolución de las dependencias Java, proponiendo un servicio especializado en resolver y ubicar las bibliotecas requeridas por el software en desarrollo, acorde a las demandas actuales y futuras de la industria.

Capítulo 1

Introducción

La organización y distribución de software en paquetes ha demostrado ser una solución efectiva para organizar la configuración y actualización del software. 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, a los que denominó Gestores de Dependencias.

Esos Gestores de Dependencias, al igual que los Gestores de Paquetes, 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 la presencia de dependencias transitivas. Estas unidades de software, las bibliotecas, 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 disponibilidad inmediata de 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 una relación estática entre el cliente y el repositorio central, donde el cliente sólo tiene acceso a las fuentes que son declaradas en los metadatos, estableciéndose así un alto nivel de acoplamiento entre el cliente y el servidor [5]. Por otro lado, también se ha demostrado que en muchos casos esto trae como consecuencia un alto grado de subutilización de dependencias, impactando significativamente en la mantenibilidad y portabilidad del sistema [6].

En el caso de la tecnología Java [13], cada paquete o biblioteca está integrado por un conjunto de clases o *binarios* altamente cohesionados, 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.

La última versión (9) de la plataforma Java (recientemente lanzada el pasado 21 de septiembre) incorpora un sistema de módulos (proyecto Jigsaw) con el propósito de fragmentar internamente la plataforma y, a la vez, ofrecer herramientas para construir software modular [8].

Los principales objetivos de Jigsaw son:

- Incrementar la seguridad y mantenibilidad de la plataforma.
- Optimizar el rendimiento.
- Facilitar la construcción y mantenimiento de bibliotecas (estándar o para empresa).

Este sistema de módulos redefine la organización de la plataforma y los programas creados con ella, pasando de una arquitectura monolítica a una integrada por módulos. A diferencia del sistema de bibliotecas Java anterior (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 específicas según el ambiente de ejecución (modular run-time images) [9].

Por otro lado, 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, 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 (PC de escritorio, dispositivos móviles, 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. En relación a esto, 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.

1.1. Objetivos

En las siguientes secciones, este trabajo se orienta en distinguir y proponer soluciones para los siguientes inconvenientes derivados de una inadecuada fragmentación y del modo en el que se gestionan las dependencias del software Java en la actualidad:

- **Subutilización de dependencias (SD):** Instalar software que está integrado por demasiados recursos inutilizados limita la portabilidad a ambientes de producción de recursos limitados.
- **Alto acoplamiento (AA):** 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].
- **Desempeño (DE):** En casos donde los recursos son limitados (por ejemplo: dispositivos IoT o servidores virtuales) es muy costoso -en términos de ciclos de procesador- buscar clases en un class-path con gran número de bibliotecas.
- **Conflictos entre bibliotecas (CB):** Es habitual que existan incompatibilidades entre bibliotecas que impiden la resolución de la clausura, 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 para corregir estos errores.

Tomando como base las características particulares de la compilación y ejecución del software Java [13], el nuevo nivel de fragmentación de componentes de software para Java 9 [25] y la posibilidad de extraer contenido parcial de las fuentes web [56], este trabajo propone una solución para la resolución de dependencias libre de metadatos y rutas a fuentes estáticas. Se desarrolla en tres fases: estudio del problema, desarrollo de una propuesta inicial y, por último, revisión y actualización de la propuesta.

La primera fase, se enfocó en medir el grado de subutilización de las dependencias en el software Java actual. Para lograrlo, se desarrolló una herramienta específica con la que se analizó una muestra de productos de software desarrollados con tecnología Java y sus dependencias, verificando esa subutilización con un resultado promedio inferior al 10% de recursos referenciados, sobre el total disponible en las bibliotecas.

La segunda fase se orientó en estudiar el código intermedio generado por el compilador Java y desarrollar un prototipo basado en tecnología OSGi, donde un intermediario entre el ambiente de desarrollo y los repositorios de bibliotecas, deberá inicialmente preparar subrogantes de bibliotecas que permitan la compilación, para luego suministrar sólo las clases que formarán parte de la clausura de sus dependencias. Con ese prototipo, y a partir de

pruebas comparativas con gestores de dependencias a nivel biblioteca, se llegó a la conclusión preliminar que el empaquetado en bibliotecas no ofrecería el nivel de granularidad adecuado para construir el software Java que la industria estaría necesitando.

Por último, y a modo de adaptación a la próxima versión de la tecnología Java (sistema de módulos), durante la tercera fase se diseñó y desarrolló otro prototipo de resolución de dependencias, pero a nivel de módulos. Previamente se estudiaron las especificaciones, la documentación preliminar y las versiones de acceso temprano del JDK 9.

1.2. Aportes

Durante el desarrollo de esta Tesis, se decribirá el proceso a través del cual se presentaron y desarrollaron diferentes propuestas "materializadas" en software, que pueden ser considerados como aportes significativos de la misma:

- **Compilación a partir de representativos (o proxies) de bibliotecas:** Archivos Jar integrados sólo por clases Java públicas, en cuyos métodos públicos sólo está la firma (sin instrucciones), de modo tal que sea posible la compilación de programas externos a partir de un sustituto. Se desarrolló un prototipo para validar este concepto.
- **Suministro de clases a demanda de programas enlazados a representativos:** Para conseguir la ejecución, sólo se suministran las clases referenciadas por el programa de usuario. A su vez, las dependencias transitivas también son suministadas a nivel de clase, logrando así un suministro más eficiente de dependencias. Se desarrolló un prototipo para validar este concepto.
- **Medición de tasa de dependencia:** Para cuantificar el nivel de utilización de bibliotecas Java, se midieron las dependencias de una serie de productos masivos creados con Java, a fin de conocer la proporción de recursos referenciados por sobre el total de acceso público. Se desarrolló y validó una herramienta de medición mediante la cual se llegó a la conclusión que en promedio esa proporción no supera el 10 %.
- **Extractor de archivos remotos:** Se desarrolló un componente de software para extraer archivos comprimidos desde repositorios remotos. A fin de optimizar el tiempo de respuesta para localizar recursos de bibliotecas, se adaptó para la tecnología Java un algoritmo que extrae porciones de bytes de servidores que implementan la RFC2616.
- **Servicio para la resolución de dependencias:** Se desarrolló un servicio que localiza dinámicamente las dependencias a partir del análisis del descriptor de módulo. A diferencia de otros Gestores de Dependencias, esta solución no requiere instalación local, ni el agregado de metadatos a bibliotecas, ni la definición de rutas estáticas a repositorios.
- **Comparativa con Gestores de Dependencias actuales:** Se diseñaron y ejecutaron una serie de pruebas para comparar esta propuesta con las herramientas más utilizadas en la industria del software Java. Se llegó a la conclusión que es factible migrar el modo

como se está gestionando la resolución de dependencias, a un servicio específico que pueda dar soporte a las demandas actuales y futuras de la industria del software.

En los Anexos I a IV están disponibles el código fuente y los diagramas de clases de los principales componentes de cada solución.

A continuación, en el siguiente capítulo se describe el contexto tecnológico donde se desarrolla esta Tesis. Luego, en el Capítulo 3 se explican los antecedentes que dan fundamento a la propuesta. El Capítulo 4 describe conceptualmente la propuesta inicial del prototipo, mientras el Capítulo 5 explica las características del diseño, implementación, pruebas y resultados alcanzados con el primer prototipo. En el Capítulo 6 se describe una nueva propuesta adaptada a la próxima versión de Java, para la cual el Capítulo 7 explica las características de diseño, implementación, pruebas y resultados. En el Capítulo 8 se desarrolla un relevamiento de otros gestores de dependencias/paquetes. Luego, en el Capítulo 9 se documentan los resultados de un estudio comparativo entre el prototipo propuesto y otros gestores de dependencias. Finalmente, el Capítulo 10 presenta las conclusiones y un plan de trabajos futuros.

Capítulo 2

Contexto tecnológico

2.1. Java

El resultado de compilar código fuente Java (archivo de texto con extensión java) genera como salida un archivo de extensión class. Un archivo class[1] o binario, principalmente contiene instrucciones (o bytecode) agrupadas como métodos, y una tabla de símbolos (constant pool), que luego son interpretadas por la Máquina Virtual de Java (JVM) en tiempo de ejecución [13]. El Constant pool contiene diferentes tipos de elementos que pueden ser desde literales numéricos hasta referencias a métodos, o campos que deberán ser resueltos en tiempo de ejecución.

La Máquina Virtual de Java (JVM) es una máquina de computación abstracta, similar a una máquina de computación real. Es la implementación del patrón de diseño Máquina Virtual [14] [15]. Posee un conjunto de instrucciones y gestiona varias áreas de memoria en tiempo de ejecución. Su función es la de ejecutar software Java compilado desde cualquier plataforma de hardware o sistema operativo.

Dentro de una biblioteca Java (archivo Jar), la mínima unidad contenedora de información son los archivos binarios o de bytecode. En su tabla de símbolos también se definen referencias simbólicas a recursos externos, que luego deberán estar disponibles en el class-path en tiempo de ejecución. Esos recursos pueden ser clases pertenecientes a otros paquetes del mismo programa de usuario, o estar contenidos en bibliotecas de terceros. La estructura interna de un archivo class se define por los campos de la Fig. 1.

[1] Según la especificación, también podría tener otra extensión.

```
ClassFile {
    u4                  magic;
    u2                  minor_version;
    u2                  major_version;
    u2                  constant_pool_count;
    cp_info             constant_pool[constant_pool_count-1];
    u2                  access_flags;
    u2                  this_class;
    u2                  super_class;
    u2                  interfaces_count;
    u2                  interfaces[interfaces_count];
    u2                  fields_count;
    field_info          fields[fields_count];
    u2                  methods_count;
    method_info         methods[methods_count];
    u2                  attributes_count;
    attribute_info      attributes[attributes_count];
}
```

Fig. 1 – Estructura interna de un archivo class.

Un binario Java está principalmente integrado por elementos del Constant pool, campos (fields) y métodos (methods). Los métodos contienen las instrucciones a ejecutar por la máquina virtual, que originalmente son las sentencias del código fuente. Las instrucciones pueden ser:

- De almacenamiento y carga,
- aritméticas,
- de conversión de tipos,
- de creación y manipulación de objetos,
- de gestión de pila,
- de control de transferencia,
- y de invocación y retorno de métodos.

Un ejemplo es el método spin que es un iterador, tal como muestra la Fig. 2.

```
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
          ; //vacío
    }
}
```

Fig. 2 – Código fuente del método spin.

Una vez compilado el método spin presentado en la Fig. 2, se mostrará como resultado el listado de instrucciones que describe la Fig. 3.

```
0       iconst_0        // empujar int
1       istore_1        // almacenar en la variable i (i=0)
2       goto 8          // la primera vez no incrementar
5       iinc 1 1        // incrementar en 1 la variable
8       iload_1         // empujar el valor de la variable
9       bipush 100      // empujar el valor de la const. 100
11      if_icmplt 5     // comparar (si i<100)
14      return          // retornar al finalizar
```

Fig. 3 – Resultado de compilar el método spin.

La Máquina Virtual de Java es orientada a pila (stack-oriented), donde la mayoría de las operaciones consisten en tomar o empujar (push) resultados de la pila [13]. Como se explica en la Fig. 4, durante la ejecución de programas, la JVM carga, enlaza e inicializa dinámicamente clases e interfaces. *Cargar* implica buscar la representación binaria de una clase o interfaz con un nombre en particular y crear esa clase. *Enlazar* es el proceso de seleccionar una clase o interfaz y combinarla con el objetivo de ser ejecutada. *Inicializar* consiste en ejecutar el método de inicialización 'clinit' [13].

Las técnicas de cargar y enlazar piezas de software son recursos disponibles y utilizados desde principios de los años '70. El trabajo de Presser y White [16] las describe conceptualmente.

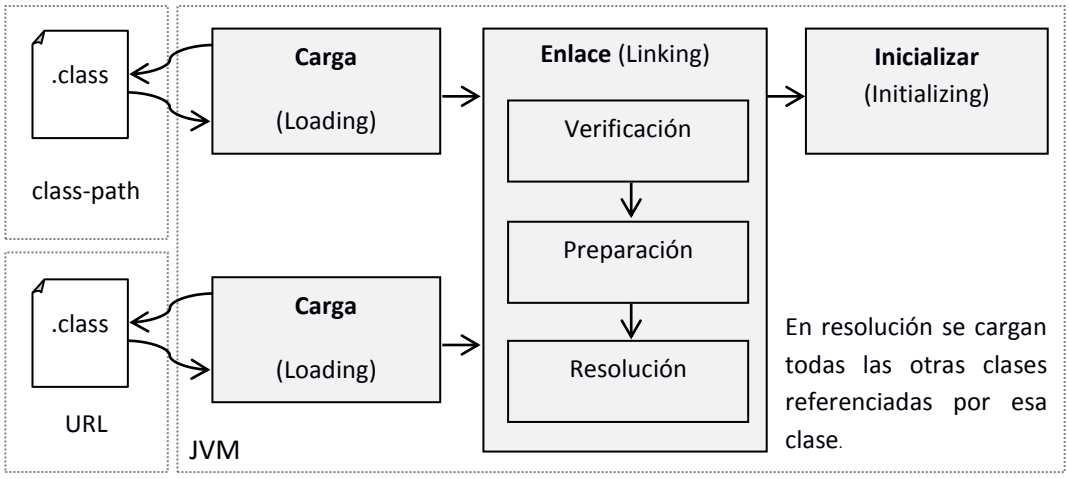


Fig. 4 - Carga, enlace e inicialización de clases en la máquina virtual de Java.

La *carga* de clases (class loading) es la capacidad que posee la JVM 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 [7].

Como se visualiza en la Fig. 4, el origen de las clases puede ser el sistema de archivos local o una fuente remota de Internet [5].

Una 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. A esto se lo llama enlace dinámico (dynamic linking). 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 [17].

2.1.1. Enlace dinámico de clases en la JVM

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. 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 entre clases ocurre en tiempo de ejecución y es transparente, tanto para programadores como para usuarios.

La *verificación* asegura que la clase es consistente y estructuralmente correcta, es decir, su constant pool es válida, también los tipos de las variables y todas las variables son inicializadas antes de ser procesadas.

La *preparación* es la fase donde se inicializan todos los campos estáticos a los valores por omisión (default) de sus respectivos tipos.

La *resolución* es el proceso donde se evalúan que las referencias simbólicas de la constant pool apunten a clases válidas de los tipos requeridos. La resolución de las referencias simbólicas implica la carga de esas clases. De acuerdo con la especificación, esta acción también puede ser ejecutada en modo perezoso (lazy), donde la resolución es diferida hasta que la clase es utilizada.

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 podría postergar hasta el

momento en que el programa es ejecutado. En la Fig. 5 un ejemplo de referencia simbólica a la clase org.geotools.data.FileDataStore desde la clase prueba.Quickstart (entrada #80).

```
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
```

Fig. 5 - Referencias simbólicas en la clase Quickstart.

2.1.2. Java HotSpot

La tecnología HotSpot o puntos calientes es parte de la JVM e incrementa el desempeño de los programas mediante la compilación en tiempo de ejecución o Just-In-Time (JIT) del bytecode en código nativo. Cuando un compilador JIT es parte de la JVM, ciertas porciones de bytecode son compiladas (en tiempo real) en código ejecutable. Cabe destacar que no todas las secuencias de bytecode son compiladas: sólo aquellos “puntos calientes” que pueden producir un significativo aumento de rendimiento. El código restante es interpretado [18].

2.2. Java Community Process

Es el proceso creado por Sun Microsystems para desarrollar estándares de Java. Actualmente es operado por Oracle. La JCP produce solicitudes de especificaciones Java (Java Specification Request o JSR) a través de un proceso formal, donde cada JSR es desarrollada por un Grupo de Expertos que son parte de la JCP. Habitualmente son designados por las empresas interesadas en expandir la plataforma Java. Cada grupo de trabajo es responsable de entregar la especificación completa y una implementación de referencia, a modo de prueba de concepto, junto con un conjunto de pruebas de conformidad, para verificar que la tecnología cumple con la especificación. La membrecía es abierta, es decir, tanto individuos como empleados de empresas pueden ser parte de la comunidad y contribuir en el desarrollo de los estándares. La JCP existe desde el año 1998 y continúa en plena actividad y expansión [19] [20].

2.2.1. JSR 376

El Sistema de Módulos para la Plataforma Java (JMPS) es una propuesta de especificación liderada por Mark Reinhold (Oracle) cuyo objetivo es definir un sistema de

módulos escalable para Java. En este sistema se destaca la simplicidad y facilidad para usar, crear y mantener bibliotecas y aplicaciones Java de gran magnitud, tanto para la edición estándar (SE) como empresarial (EE). Su alcance define qué se deberá utilizar para modularizar la plataforma Java y sus implementaciones [21]. Esta especificación es parte de la reciente versión 9[2]. El conjunto de propuestas y estándares que conforman el sistema de módulos para Java se denomina proyecto Jigsaw [8].

2.3. Java versión 9

El grupo de trabajo a cargo de la JSR 376 definió una serie de Propuestas de Mejoras para el JDK (JDK Enhancement Proposals o JEP) a modo de guía para alcanzar los objetivos definidos en la propuesta de especificación marco. En esas propuestas (200, 201, 220, 260, 261 y 282) se formalizan definiciones de diseño, implementación, riesgos y pruebas que en conjunto se denominan Jigsaw. Este proyecto será la contribución más importante para la plataforma Java 9 y, si bien los autores no esperan una inmediata adopción por parte de la industria del software, son optimistas respecto del impacto a mediano y largo plazo. Originalmente Jigsaw fue propuesto para la versión 7 de Java, luego postergado para la versión 8 y finalmente se incorporó formalmente para la versión 9 [22].

Al momento de desarrollar este trabajo, se había liberado una versión de acceso temprano (JDK 9 Early Access) disponible para probar las nuevas funcionalidades y evaluar retrocompatibilidad con sistemas desarrollados para versiones anteriores [23].

2.3.1. Sistema de módulos

JDK 9 determina cambios profundos a nivel estructural, compilación y en la sintaxis del lenguaje Java. Cada Jar modular (biblioteca del módulo) deberá incorporar un descriptor con:

- Nombre del módulo,
- módulos que requiere,
- paquetes que exporta,
- servicios que provee,
- servicios que consume y otros.

Una sentencia de declaración de módulo estará definida por la estructura que muestra la Fig. 6.

2 En la votación del 29/11/2016 esta JSR obtuvo 21 votos a favor sobre un total de 23 votos. En línea: https://jcp.org/en/jsr/results?id=5893

```
ModuleStatement:
    requires {RequiresModifier} ModuleName ;
    exports PackageName [to ModuleName {, ModuleName}] ;
    opens PackageName [to ModuleName {, ModuleName}] ;
    uses TypeName ;
    provides TypeName with TypeName {, TypeName} ;
```

Fig. 6 – Definición formal de la sentencia que define a un módulo.

Formalmente, se incorporará a la próxima versión de la Especificación del Lenguaje Java (Java Language Specification o JLS) [24].

La Fig. 7 muestra un ejemplo donde de define el descriptor para el módulo com.socket y org.fastsocket. El primero usa el servicio provisto por NetworkSocketProvider y org.fastsocket provee el servicio a través de la implementación FastNetworkSocketProvider.

```
src/com.socket/module-info.java
      module com.socket {
              exports com.socket;
              exports com.socket.spi;
              uses com.socket.spi.NetworkSocketProvider;
          }

src/org.fastsocket/module-info.java
      module org.fastsocket {
              requires com.socket;
              provides com.socket.spi.NetworkSocketProvider
                  with org.fastsocket.FastNetworkSocketProvider;
          }
```

Fig. 7 – Ejemplo de descriptores de módulos.

El sistema de módulos también requiere modificaciones al documento sobre la Especificación de Máquina Virtual de Java (Java Virtual Machine Specification o JVMS) [13]. Allí se explican los nuevos atributos necesarios para describir el bytecode de los módulos y su información [25].

2.3.2. Imágenes modulares para tiempo de ejecución

Como parte de los hitos requeridos para cumplir con la JSR 376, el proyecto Jigsaw también incluye a la JEP 220: Modular Run-Time Images, con el objetivo de reestructurar el JDK y la JRE (Edición tiempo de ejecución) de modo tal que sea posible generar imágenes a medida a fin de incrementar desempeño, seguridad y mantenibilidad de la plataforma Java. Esta propuesta de mejora suma la posibilidad de generar perfiles compactos y personalizados, donde sólo sean parte de la imagen a desplegar los módulos requeridos para dar soporte a la ejecución al conjunto de módulos del programa de usuario y sus módulos dependientes [26].

2.4. OSGi

La tecnología OSGi es un conjunto de especificaciones que definen un sistema de módulos para Java. Estas especificaciones permiten desarrollar software bajo un modelo donde las aplicaciones están integradas por módulos reutilizables. Los módulos OSGi se comunican entre sí a través de servicios. En esta tecnología, los módulos son archivos Jar que explícitamente definen qué servicios requieren, qué servicios exportan, versiones, entre otros. La particularidad de OSGi es que define un ciclo de vida para los módulos que pueden estar en alguno de los siguientes estados:

- Instalado.
- Resuelto.
- Detenido.
- Iniciado.
- Activo.
- Desinstalado.

Los servicios se registran en un *registro de servicios* y pueden ser agregados y quitados dinámicamente, posibilitando actualizaciones en tiempo de ejecución de forma transparente. Otra particularidad es que los módulos o *bundles* deben ser desplegados en un marco de trabajo o framework. Algunas implementaciones de frameworks OSGi más conocidas son Equinox, Apache Felix, Eclipse Gemini. La última versión (6) fue editada en 2014 [27]. Otra importante contribución de OSGi es R-OSGI (Remote OSGi) que permite crear ambientes distribuidos donde los bundles acceden a servicios localizados en frameworks remotos, como si fueran parte del ambiente local [28].

A diferencia de la tecnología Java (anterior a la versión 9), OSGi define dependencias en sus metadatos. Cada bundle debe incluir el nombre y versión de los paquetes que importa, al igual que los servicios que consume [29]. Esto permite conocer la clausura del entorno de ejecución o desarrollo (el conjunto de bundles), sin necesidad de contar datos externos. El ambiente de desarrollo Eclipse (Platform Plug-in Development Environment o PDE), que está basado en OSGi, establece la necesidad de contar con una plataforma de destino o *target platform* que son el conjunto de *plug-ins* (bundles OSGi) requeridos para poder compilar y ejecutar una aplicación de usuario. Las diferentes plataformas de destino se declaran como definiciones de destino o *target definitions* [30].

2.5. Gestores de Paquetes

Es habitual contar con un gestor de paquetes en un entorno Linux. Han demostrado ser una solución exitosa para administrar distribuciones de software disponibles en repositorios remotos. Es una solución donde un cliente recupera paquetes y mediante el análisis de sus metadatos, también obtiene todos los demás paquetes necesarios para conseguir la clausura [31] [32]. Es un modelo que ha transcendido el ámbito de los sistemas

operativos y fue incorporado a la comunidad de desarrollo de software personal o industrial, bajo el nombre de Gestores de Dependencias. Ambos recuperan el nuevo software desde repositorios remotos y donde cada unidad de software (paquete/biblioteca) posee asociado un archivo con metadatos que describen sus dependencias y dónde localizarlas.

2.6. Gestores de Dependencias

En el ámbito del desarrollo de software Java, desde hace varios años se están empleando Gestores de Dependencias para administrar las bibliotecas del software en desarrollo. Actualmente la herramienta *de facto* para la industria es Apache Maven, que también es un gestor de proyectos y una herramienta de construcción de software (build tool) [33]. Al igual que Maven, existen otros gestores de dependencias que también están basados en el modelo de gestor de paquetes, donde es el cliente el que debe contar con los suficientes metadatos para conseguir la correcta clausura de las bibliotecas disponibles en repositorios remotos. Si bien estos gestores de dependencias han hecho un significativo aporte al soporte de proyectos de software de gran escala, su efectividad todavía es limitada, requiriendo intervención humana en casi un 40% de los casos [12].

2.7. Conclusiones del Capítulo

En este Capítulo se describió cómo un archivo de código fuente Java es traducido durante el proceso de compilación a instrucciones en bytecode, para luego ser interpretadas por la máquina virtual de Java (JVM) durante su ejecución.

Por otro lado, también se explica el modo en que la especificación de la JVM establece que deben cargarse, enlazarse e inicializarse las clases Java en tiempo de ejecución.

A su vez, también se hace referencia a los procesos formales de desarrollo y validación de estándares vinculados a la tecnología Java, donde surgen, entre otros, las especificaciones que se materializan en el reciente lanzamiento de la versión 9 de la plataforma y que presenta como principal novedad el sistema de módulos (JPMS). La Java Specification Requirements 376 (JSR-376) define las características del JPMS que establece la presencia obligatoria de un descriptor para cada módulo, donde se indica el nombre del módulo, los módulos requeridos, recursos que provee, los que consume y otros. Como parte de las ampliaciones, se suma a la plataforma la posibilidad de crear imágenes a medida según el ambiente de ejecución, logrando así un más eficiente aprovechamiento de recursos.

También en este Capítulo se presenta una breve descripción técnica de la tecnología OSGi, mencionando sus principales características e implementaciones más difundidas.

Por último, se hace una breve referencia a los Gestores de Paquetes y su adaptación al ámbito del desarrollo de software como Gestores de Dependencias, que al igual que los Gestores de Paquetes, dependen de la presencia de metadatos, rutas a repositorios y una correcta configuración del ambiente, para conseguir la clausura de recursos externos de un proyecto de desarrollo de software.

En este contexto tecnológico es donde surge la propuesta de avanzar en el modo que se resuelven y obtienen las dependencias de software Java. Pasando de un modelo centrado en la estación de trabajo, metadatos y repositorios, a un servicio en la nube y especializado en resolver dependencias.

Capítulo 3

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 [33]. 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 [34]. 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. 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 las dependencias transitivas (las dependencias de las dependencias) [35]. Si bien Maven y otras herramientas similares proponen perfiles de compilación o 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 [33]. Otro inconveniente que presenta este modelo es la

subutilización de recursos de bibliotecas [5] [6]. A continuación, se presenta un caso de ejemplo.

3.1. Gestión de Bibliotecas con Maven

3.1.1. Quickstart de GeoTools, Parte 1

Maven provee y gestiona dependencias mediante la copia local de las bibliotecas directas e indirectas. En relación a esto, la Fig. 8 muestra una vista jerárquica de las bibliotecas (o artefactos) requeridos para compilar y ejecutar un ejemplo introductorio del software GeoTools[3]. Esta vista parcial muestra el listado de bibliotecas directas (gt-shapefile y gt-swing) y las indirectas (las transitivas).

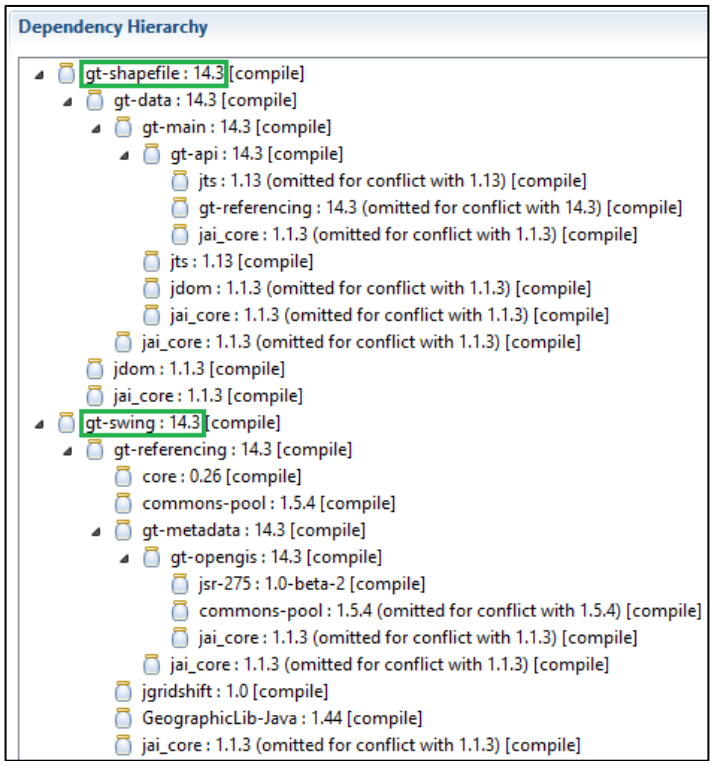


Fig. 8 - Dependencias directas y transitivas requeridas por la clase Quickstart de GeoTools.

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 58 bibliotecas adicionales para poder compilar y ejecutar el programa de 16 líneas de código que se muestran en la Fig. 9.

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

```
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. 9 - Código fuente de la clase Quickstart.

Para conseguir la jerarquía presentada en la Fig. 8, primero se debe configurar el archivo descriptor (pom.xml) especificando los repositorios y las dependencias directas, como se muestra en la Fig. 10.

```
<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>org.geotools</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. 10 - Archivo descriptor de proyecto Quickstart con Maven (pom.xml).

Esta forma de establecer la fuente de dependencias trae como consecuencia:

- **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 en el 40% de los casos donde falla la

construcción de un proyecto Maven, el problema está relacionado con la resolución de dependencias.

- **Conflictos entre bibliotecas:** Es habitual que existan problemas durante la resolución de dependencias [35]. El sistema de gestión de dependencias debe contar con suficiente capacidad analítica para evitar estos conflictos. Además, es imprescindible que los metadatos estén completos, incluyendo casos especiales de incompatibilidad [38].

A continuación, se presenta un análisis que profundiza acerca del uso de bibliotecas Java.

3.2. Medición de uso de dependencias Java

Con el propósito de obtener un dato cuantitativo acerca de la cantidad de referencias entre el programa Quickstart y sus dependencias, se emplea una herramienta desarrollada específicamente para esta tesis. Se llama Deep[4] y mide sobre el total de recursos públicos disponibles en biblioteca (destino), qué proporción es referenciada por el Jar de origen (el programa de usuario) [5]. Previamente a desarrollar esta herramienta, se analizaron y descartaron las siguientes por diversas causas que se detallan, que también miden acoplamiento con dependencias. A continuación, un resumen de los resultados obtenidos con cada una.

3.2.1. Herramientas de Medición

3.2.1.1. CodePro Analytix

Es un conjunto de herramientas de soporte al aseguramiento de la calidad del software integradas al IDE Eclipse. Cuenta con una importante variedad de funcionalidades, entre las que se destacan: métricas de software, análisis de código muerto, generación de casos de prueba unitarios, y otros.

Para este estudio, se evaluó la funcionalidad “Análisis de Dependencias”, donde cada referencia es contada individualmente, es decir, las repeticiones se cuentan individualmente. Por ejemplo, si una clase tiene 2 referencias a una misma clase externa (de biblioteca), son contadas como 2 referencias. Además, en el informe detallado, los resultados son globales, en relación a todas las dependencias, y no a una en particular. En la Fig. 11 se visualizan las referencias de Deep a sus dependencias.

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

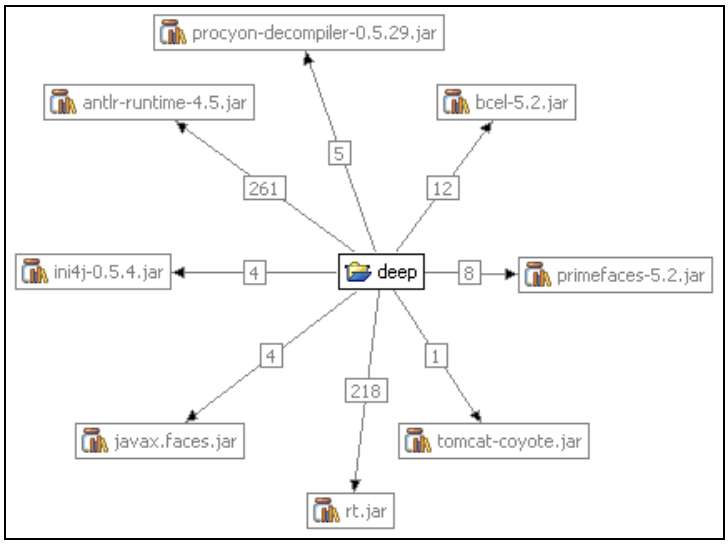


Fig. 11 - Vista Dependencies de CodePro Analytix.

3.2.1.2. STAN (Structure Analysis for Java)

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, métricas, composición, acoplamiento, entre otros [36]. Desde la vista Dependencias, STAN muestra un gráfico donde un valor numérico mide el peso de la fortaleza de la dependencia (ver Fig. 12).

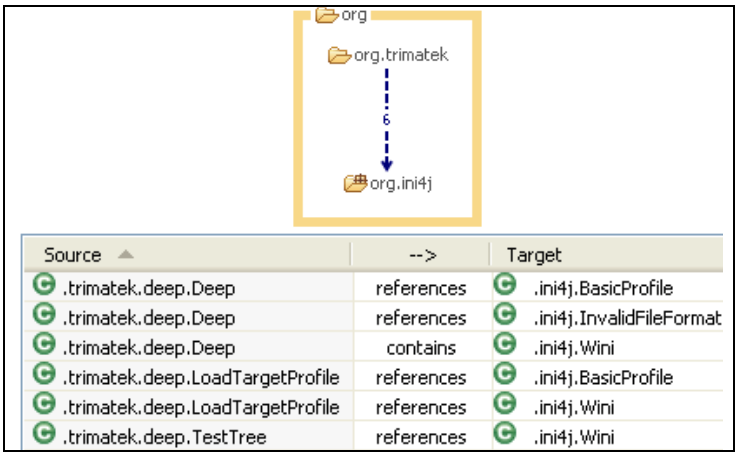


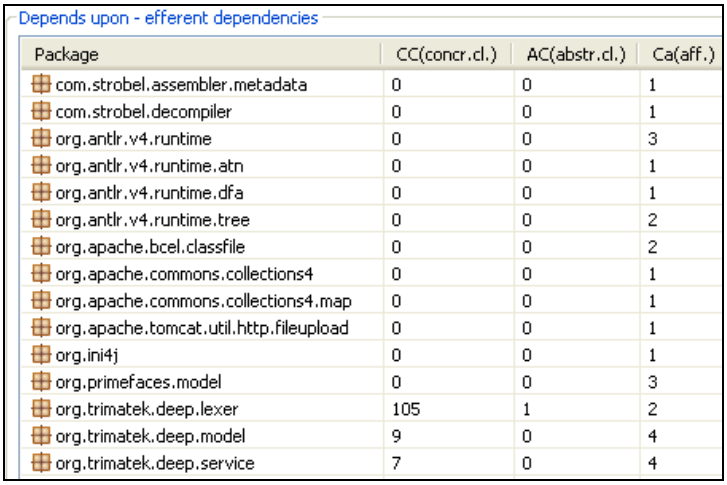
Fig. 12 - Medición de atributo "peso" con STAN.

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) porque repite las referencias. Por ejemplo, como se ve en la Fig. 12, hay 2 referencias de Deep a la clase Wini, cuando el objetivo es que sólo se registre que existe una referencia a ese recurso. Otra desventaja es que no es software de acceso libre.

3.2.1.3. JDepend

Es otra herramienta para analizar las dependencias de software Java. Se puede ejecutar de forma independiente o desde Eclipse.

A partir de una serie de pruebas (Fig. 13), también fue descartada por los siguientes motivos: 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 entre el software en análisis y sus bibliotecas, es decir, muestra a sus dependencias como un todo sin especificar en detalle a cuál corresponde el cálculo.



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

Fig. 13 - Análisis de dependencias con JDepend.

3.2.1.4. Jdeps

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

Como puede verse en la Fig. 14, esta herramienta no muestra resultados cuantitativos, sino sólo una relación entre clases propias y externas unidas con el símbolo '->', por lo que también fue descartada.

```
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";
```

Fig. 14 - Salida de análisis de dependencias con Jdeps.

3.2.1.5. CDA (Class Dependency Analysis)

De manera similar a Jdeps, Class Dependecy Analysis (CDA) grafica la relación entre bibliotecas (Fig. 15).

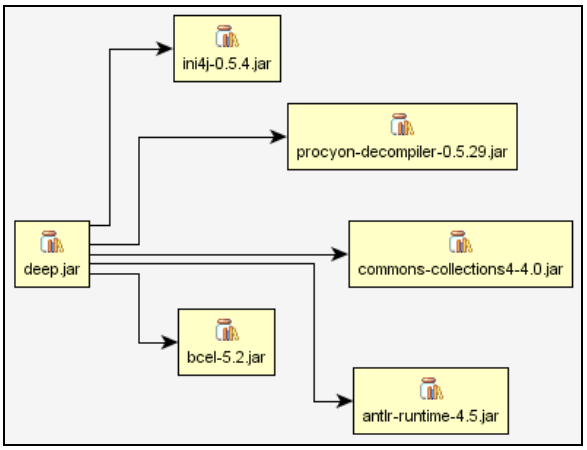


Fig. 15 - Análisis de dependencias con CDA.

CDA también genera un listado por clases o paquetes pero, como Jdeps, tampoco muestra resultados numéricos.

3.2.1.6. Deep

Dado que ninguna de estas herramientas cumplía con el requisito de medir referencias únicas, se decidió desarrollar una a medida. Deep está disponible como software de código abierto[5] y realiza un análisis similar al propuesto por Wang [6] (que releva dependencias a nivel módulo, pero sólo analiza código fuente en C++) para calcular una métrica basada en el trabajo de Martin [36]. En una misma sesión, Deep analiza dos archivos Jar y establece una tasa de dependencia entre sí. Básicamente, el algoritmo principal realiza lo siguiente:

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 dependencias y muestra los resultados (Analysis Results).

Parte del código fuente está disponible en el Anexo IV de esta Tesis.

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

3.2.1.6.1. 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 [36] (Fig. 16), 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.

Fig. 16 - Métrica Inestabilidad (R. Martin).

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:

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

Siendo:

S: El archivo Jar origen.
T: El archivo 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, significa que la proporción entre recursos disponibles y utilizados es alta. Por otro lado, un resultado cercano a 0 significa mínima presencia de referencias al Jar objetivo. Es un promedio de las proporciones entre los recursos referenciados y el total de disponibles.

3.2.1.6.2. Validación de Deep

Para validar la precisión de Deep, se analizaron sus dependencias (Tabla 1), comparando los resultados con CodePro Analytix, STAN y un relevamiento manual.

Tabla 1 – Dependencias de Deep.

Dependencia	Uso
Ini4J	Se emplea para leer la configuración desde un archivo de texto (consola).
Apache Bcel	Relevar los recursos públicos disponibles en el Jar objetivo (T).
Procyon Decompiler	Descompilar las clases del archivo Jar origen (S).
Common Collections	Organizar resultados parciales en un mapa de clave múltple.
ANTLR	Separar en tokens las clases descompiladas para identificar los recursos del objetivo (T) en el archivo fuente del origen (S).
PrimeFaces	Conjunto de componentes visuales para implementar la Interfaz gráfica web.

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 – Resultados de mediciones de dependencias.

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

En el gráfico de la Fig. 17 se destacan en color azul los resultados de Deep, y en verde la revisión manual del código fuente.

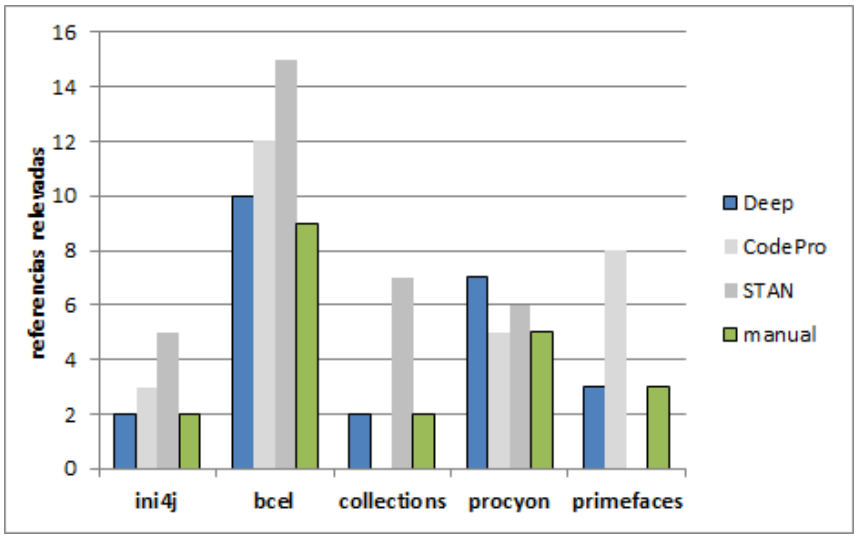


Fig. 17 - Resultados de la Tabla 2.

A partir del resultado del análisis entre Deep y sus dependencias, a continuación, se explican los casos donde se presentó alguna diferencia entre las mediciones llevadas adelante con herramientas de software (Deep, CodePro, STAN) y la inspección manual realizada (Tabla 2):

#1: 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

#2: 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.

#3: 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.

#4: 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.

#5: La diferencia de 11 respecto del conteo manual se da porque Deep detecta clases anidadas que no son parte del código fuente, pero sí del bytecode.

#6: 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.

A modo de conclusión, se puede decir que 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 la revisión manual vs. CodePro y STAN es mayor, porque 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.

3.2.1.6.3. Mediciones con Deep

Para cuantificar esta supuesta subutilización de bibliotecas, 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[6]: Es un framework orientado a simplificar el desarrollo de aplicaciones de uso empresarial. Abarca desde interfaces gráficas, seguridad, comunicaciones, servicios web hasta mensajería.

Drools[7]: Es un sistema de gestión de reglas de negocio (BRMS) que provee un núcleo denominado motor de reglas de negocio (BRE) y un módulo compilador de reglas.

Hibernate[8]: Es un conjunto de herramientas que facilita la persistencia de objetos en bases de datos relacionales. A su vez, también implementa la especificación Java Persistence API (API).

SymmetricDS[9]: Es una solución de sincronización automática de bases de datos que también realiza transformaciones en ambientes heterogéneos.

DbVisualizer[10]: 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.

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

[6] https://spring.io/

[7] https://www.drools.org/

[8] http://hibernate.org/orm/

[9] https://www.symmetricds.org/

[10] https://www.dbvis.com/

Tabla 3 - Mediciones de tasa de dependencia con Deep.

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 particular es la relación entre la biblioteca ANTLR Runtime 3.5 para Drools Compiler 6.2, donde el análisis entrega un resultado de 0.3160. Esto es esperable, dado que el análisis sintáctico (parsing) es una función central del módulo compilador de Drools.

En base al resultado obtenido, se puede afirmar que la tasa de utilización de dependencias para el software creado con Java, es bastante baja. Esto supone que el modelo de distribución por bibliotecas no sería el apropiado, teniendo en cuenta que sumar al class-path bibliotecas, significa demorar el proceso de búsqueda de clases durante la carga (class loading).

3.2.2. Quickstart de GeoTools, Parte 2

Retomando al caso inicial y con el propósito de cuantificar la tasa de uso (referencias) entre el programa Quickstart y sus dependencias, se emplea la herramienta Deep (Fig. 18).

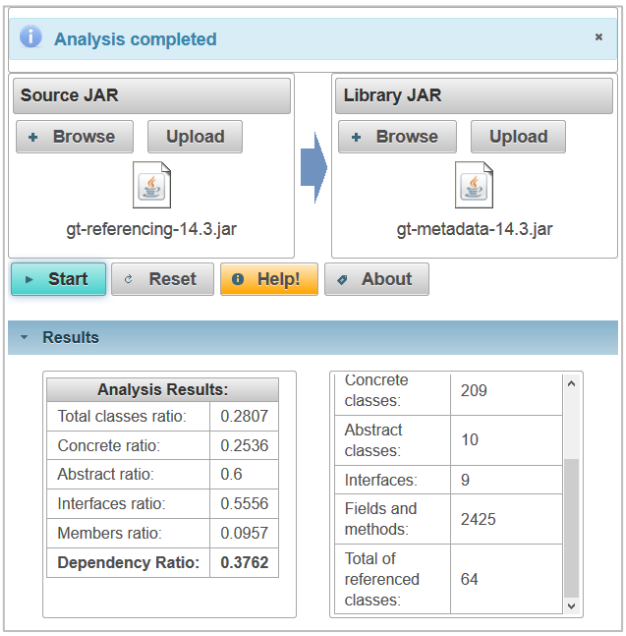


Fig. 18 – Interfaz gráfica de Deep.

La Tabla 4 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 4 - Tasa de referencias entre Quickstart y dependencias.

<table>
<tr><td>Nivel 0</td><td colspan="2"></td></tr>
<tr><td rowspan="8">quickstart</td><td colspan="2">Nivel 1</td></tr>
<tr><td rowspan="6">gt-swing
(→ 0.0125)</td><td>Nivel 2</td></tr>
<tr><td rowspan="4">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>

En la muestra de la Tabla 4 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 alcanzan 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 estos resultados, se puede afirmar a modo de conclusión parcial que, 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. Sólo fragmentar un programa Java a nivel de bibliotecas y gestionarlas a través de un Gestor de Dependencias, además de alto acoplamiento (AA) y conflictos entre bibliotecas (CB), trae como consecuencia:

Subutilización de dependencias (SD): Observada en el trabajo de Wang [6] y validada por este estudio. Estaría indicando que el modelo de vuelco total de bibliotecas a un entorno local posiblemente haya sido apropiado para los años ‘90 y principios de los 2000, cuando el acceso a Internet era un recurso limitado.

Desempeño (DE): En casos donde los recursos son limitados, por ejemplo: dispositivos IoT o servidores virtuales, el costo de enlazar clases se incrementa si las bibliotecas contienen un gran número de clases no referenciadas. Buscar clases en el class-path tiene un importante impacto en el tiempo de respuesta del sistema, que se incrementa a medida que se suman bibliotecas [37].

3.3. Conclusiones del Capítulo

El modo como se distribuye el software Java, es decir en bibliotecas, trae como consecuencia un alto nivel de subutilización de recursos. Durante el tiempo de compilación y ejecución, todas las clases que conforman esas bibliotecas comparten un espacio denominado class-path, esto provoca que en muchos casos ocurran conflictos entre diferentes versiones de una misma clase o también que la búsqueda de clases sea significativamente más costosa en términos de desempeño de un programa. Para ejemplificar y cuantificar esta situación, este Capítulo presenta un caso típico de resolución de dependencias (el programa Quickstart de Geotools) donde, trabajando con el Gestor de Dependencias Maven, sus 2 dependencias directas requieren la presencia de 58 dependencias transitivas. Además, como se muestra en la Fig. 10, con esta tecnología es necesario indicar la ubicación de estas bibliotecas.

Inicialmente se desarrolla un relevamiento de herramientas para medir la proporción de recursos referenciados de una biblioteca Java en relación a todos sus recursos de alcance público. Se probaron y evaluaron las herramientas CodePro Analytix, STAN, JDepend, Jdeps y CDA llegando a la conclusión que no eran apropiadas para este estudio. Al no contar con una herramienta que se ajustara a los requerimientos de este estudio, se desarrolló una específica para analizar el bytecode y obtener la Tasa de Dependencia entre software Java. Se validó con

las mismas dependencias de la herramienta, comparando los resultados con CodePro Analytix, CDA y una inspección manual del código. Por último, se midió una muestra de 5 soluciones de software Java de uso industrial (Spring framework, Hibernate, Drools core y compiler, Symmetric y DBVisualizer), obteniendo un promedio general inferior al 10%, es decir, del total de recursos públicos disponibles en las dependencias directas del software medido, sólo se referencia el 8 % en promedio. Finalmente se midieron las dependencias directas del programa Quickstart, llegando a un resultado promedio inferior al 2%.

De este modo se comprueba cuantitativamente que la tasa de utilización de recursos de bibliotecas Java es habitualmente bajo. Esto estaría indicando que el nivel de granularidad ofrecido por la “componentización” del software Java en bibliotecas no sería apropiado, dado que esto también repercute en el desempeño del software en tiempo de ejecución.

Capítulo 4

Propuesta inicial

Tomando como base el estudio de las secciones anteriores, se puede afirmar que:

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 y enlazará las dependencias simbólicas.
3. Analizando la información disponible en el constant pool del binario, es posible conocer qué clases son referenciadas por el programa de usuario y las bibliotecas.

A partir del trabajo de Petrea y Grigoraș para plataformas móviles de recursos limitados [39], inicialmente, esta tesis propone migrar del vuelco total de bibliotecas para la resolución de dependencias, a un suministro a demanda de clases para los ambientes de desarrollo.

4.1. Proxy de bibliotecas y clases a demanda

Para centralizar y optimizar la gestión de bibliotecas, se propone reemplazar el sistema actual por uno que resuelva y entregue dependencias a nivel de clase Java. Como recién en tiempo de ejecución es necesario contar con el recurso real, en tiempo de compilación sería suficiente con conocer la firma de los recursos públicos. 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. A modo de ejemplo, en la Fig. 19 se muestra una extracción del constant pool de la clase Quickstart, obtenida con el programa javap[11], 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.

11 http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html

```
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
```

Fig. 19 - Extracto del constant pool de la clase Quickstart.

Se propone entonces, compilar a partir de un proxy de la biblioteca, un representativo que contenga únicamente descriptores y firmas de las clases [40]. Luego, analizar y entregar sólo las clases que son referenciadas por el programa del usuario en las dependencias directas y transitivas, como se ilustra en el ejemplo de la Fig. 20, entre las bibliotecas dbvisualizer-9.2.8.jar, dom4j-1.6.jar y jaxen-1.1.6.jar.

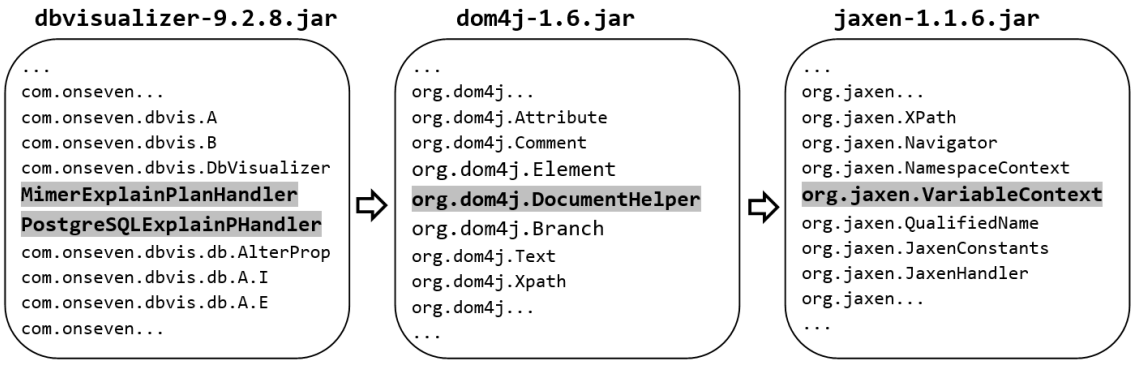


Fig. 20 – Referencias entre clases de bibliotecas.

De este modo, en tiempo de compilación, es posible establecer las referencias simbólicas en el programa de usuario y con la dependencia directa. Recién en tiempo de ejecución será necesario contar con las clases referenciadas.

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 serán parte las firmas de los recursos públicos. 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 de la Fig. 21.

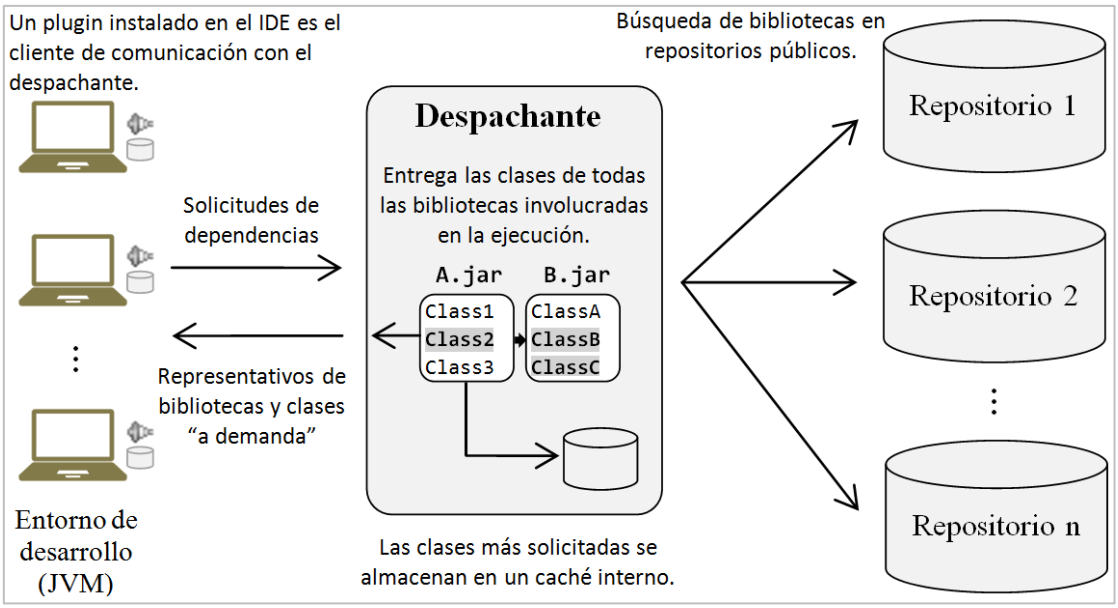


Fig. 21 - Servicio intermediario entre repositorios y los ambientes de desarrollo.

En esta propuesta, el servicio despachante conoce a qué biblioteca pertenecen las clases referenciadas a partir del archivo de sus meta-datos (POM). 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 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. 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 contar con un caché de bibliotecas, a fin de agilizar el tiempo de respuesta.

4.2. Conclusiones del Capítulo

El fundamento principal de la propuesta presentada en este Capítulo radica en que la mayoría de los proyectos de software, sólo se emplea una mínima porción de los recursos públicos de sus dependencias. Como se explica en el Capítulo 3, habitualmente la tasa de utilización de bibliotecas Java no supera el 10% del total de recursos públicos (clases, interfaces, métodos y variables de clase) [5]. Esta subutilización de recursos justificaría que se trabaje con granularidad de dependencias a nivel de clase Java en lugar de hacerlo con la biblioteca completa. Además, se propone que un intermediario entre el ambiente de desarrollo y los repositorios, 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 del cómo trabajan los Gestores de Dependencias, 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 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 (el intermediario) 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.

De este modo se estaría dando solución al problema de subutilización de dependencias (SD) y al mismo tiempo, contar con una plataforma de ejecución que sólo esté integrada por los recursos referenciados, también solucionaría los problemas de desempeño (DE) en tiempo de ejecución.

Capítulo 5

Primer Prototipo

5.1. Diseño e implementación

A modo de prueba de conceptos, se diseñó e implementó un prototipo. El software se divide en tres componentes principales:

1. Interfaz con el usuario (UI).
2. Despachante (intermediario o middleware).
3. Repositorio de bibliotecas.

Los componentes 1 y 2 se desarrollaron para este proyecto, mientras que el componente 3 es un servicio de terceros[12] [41]. Tanto para la UI, como para el servicio despachante, la tecnología marco es OSGi [42]. El diseño general del prototipo se puede representar con el siguiente diagrama (Fig. 22):

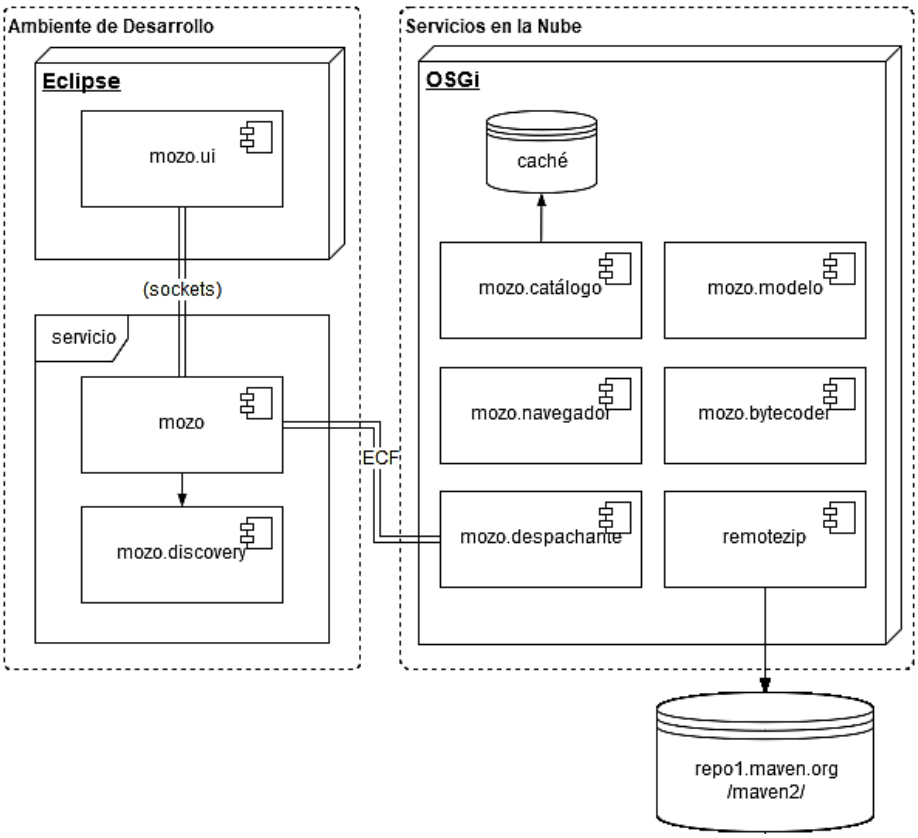


Fig. 22 - Arquitectura general del primer prototipo.

12 https://repo1.maven.org/maven2/

Se denomina Mozo y está integrado por 9 módulos que, a través de servicios OSGi, asisten a la compilación y ejecución de las solicitudes de un programa en desarrollo. Todo el software del prototipo está disponible como código abierto: https://github.com/martinaguero/mozo. A continuación, se presenta una breve descripción de cada componente.

5.1.1. Interfaz con el usuario

Del lado del usuario, el sistema es un plug-in para Eclipse (mozo.ui) y un proceso en segundo plano (mozo). El plug-in es la interfaz con el usuario y el proceso es el componente que interactúa con los servicios del módulo Despachante (mozo.despachante), a través de Eclipse Communication Framework (ECF)[13]. El módulo Discovery (mozo.discovery) localiza los servicios en la nube y guía 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 plug-in 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. La segunda, releva todas las clases referenciadas en el proyecto y solicita al Despachante obtener las clases necesarias para ejecutar el programa.

5.1.2. Intermediario o Despachante

El núcleo del sistema es un conjunto de microservicios, donde la interfaz con el ambiente de desarrollo son los dos servicios que expone el módulo Despachante (Fig. 23). El servicio Mozo (ambiente de desarrollo) invoca los métodos remotos loadJarProxy y fetchDependencies. 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.

13 http://www.eclipse.org/ecf/

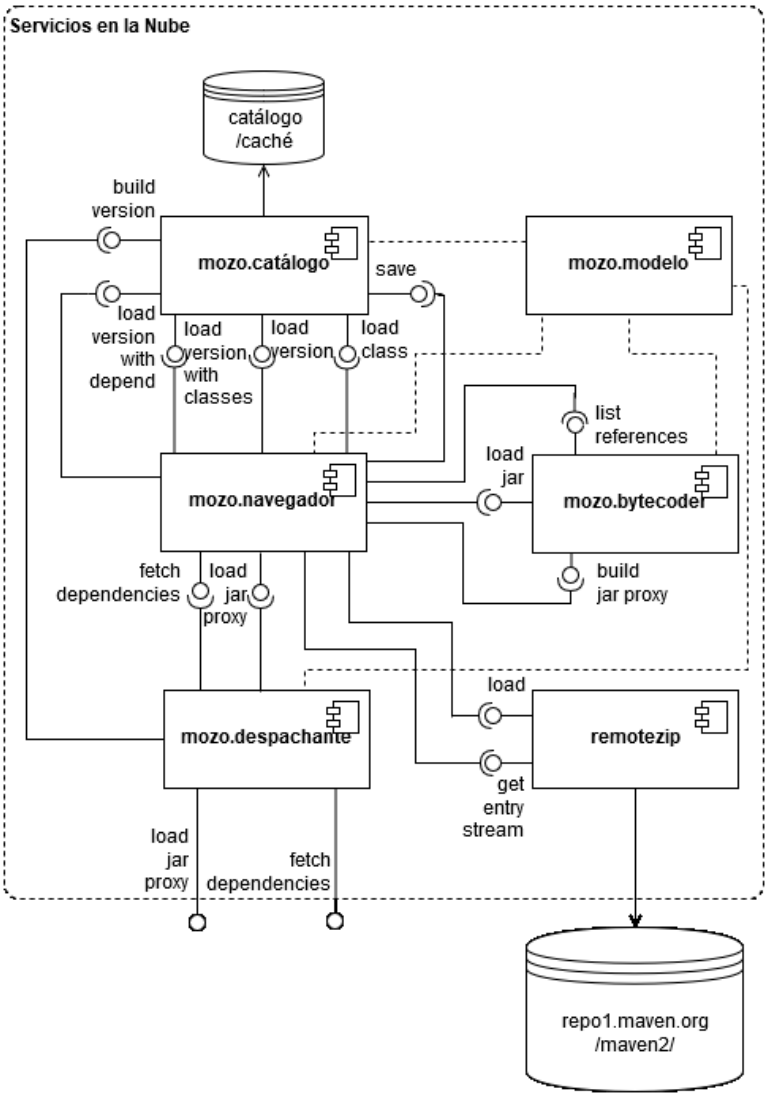


Fig. 23 – Componentes del Despachante.

El sistema almacena en un repositorio local copias de las bibliotecas entregadas al IDE, para optimizar el tiempo de respuesta en futuras solicitudes. El módulo Despachante provee dos funcionalidades:

1. Generar y entregar proxies de las bibliotecas.
2. Seleccionar y suministrar 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 servicios. Interactúa con el plug-in 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:

1. Primero genera una solicitud al catálogo, en caso de no contar con esa biblioteca, solicita al módulo Remotezip el Jar desde el repositorio.
2. Activa el servicio buildJarProxy del módulo Bytecoder y el resultado es persistido como una cadena de bytes en el catálogo.
3. Retorna el proxy al Despachante y es éste el que serializa la colección del resultado.
4. Cuando recibe una solicitud de clases, activa el servicio listReferences del módulo Bytecoder.
5. En caso de no encontrar clases referenciadas en el catálogo, genera una solicitud a Remotezip, que 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 [43]. Para este trabajo de tesis se hizo una migración a tecnología Java y permite obtener fracciones de archivos ZIP, sin necesidad de copiar el archivo localmente:

1. En primer lugar, el algoritmo busca el encabezado del directorio central del Zip, para poder conocer el índice y ubicación de cada archivo contenido.
2. Luego, con ese dato y si el servidor soporta solicitudes de rango de bytes (RFC2616: 206 - Partial Content) [56], el módulo copiará únicamente la fracción que corresponde al archivo objetivo (Fig. 24). El código fuente completo está disponible en el Anexo III de esta Tesis.

```
HttpURLConnection req = (HttpURLConnection) url.openConnection();
req.setRequestProperty("Range", "bytes=" + "-"
        + (currentLength + 22));
req.connect();
logger.log(Level.INFO, "Response Code: " + req.getResponseCode());
logger.log(Level.INFO, "Content-Length: " + req.getContentLength());
InputStream is = req.getInputStream();
byte[] bb = new byte[req.getContentLength()];
int endSize = readAll(bb, 0, req.getContentLength(), is);
req.disconnect();
int pos = endSize - 22;

while (pos >= 0) {
    if (bb[pos] == 0x50) {
        if (bb[pos + 1] == 0x4b && bb[pos + 2] == 0x05
                && bb[pos + 3] == 0x06) {
            logger.log(Level.INFO, "Central directory found!");
            break;
        }
        pos -= 4;
    } else
        pos--;
}
```

Fig. 24 - Parte del algoritmo para buscar remotamente el directorio central del archivo Zip.

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: 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 5).
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 5 - Instrucciones del método original y luego de ser 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 […] 35: putfield 38: return LineNumberTable: line 59: 0 […] 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 (Fig. 25).

Fig. 25 - Bibliotecas originales y convertidas a representativos.

Esta técnica posee ciertas limitaciones derivadas de la herramienta empleada para manipular el bytecode: Apache BCEL 5.4[14], que 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.

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 → Versión (equivalente a Artefacto de Maven) → Class. En la base están las clases (Class) que son los archivos class de Java. La Fig. 26 muestra la representación de las tablas del esquema.

14 https://commons.apache.org/proper/commons-bcel

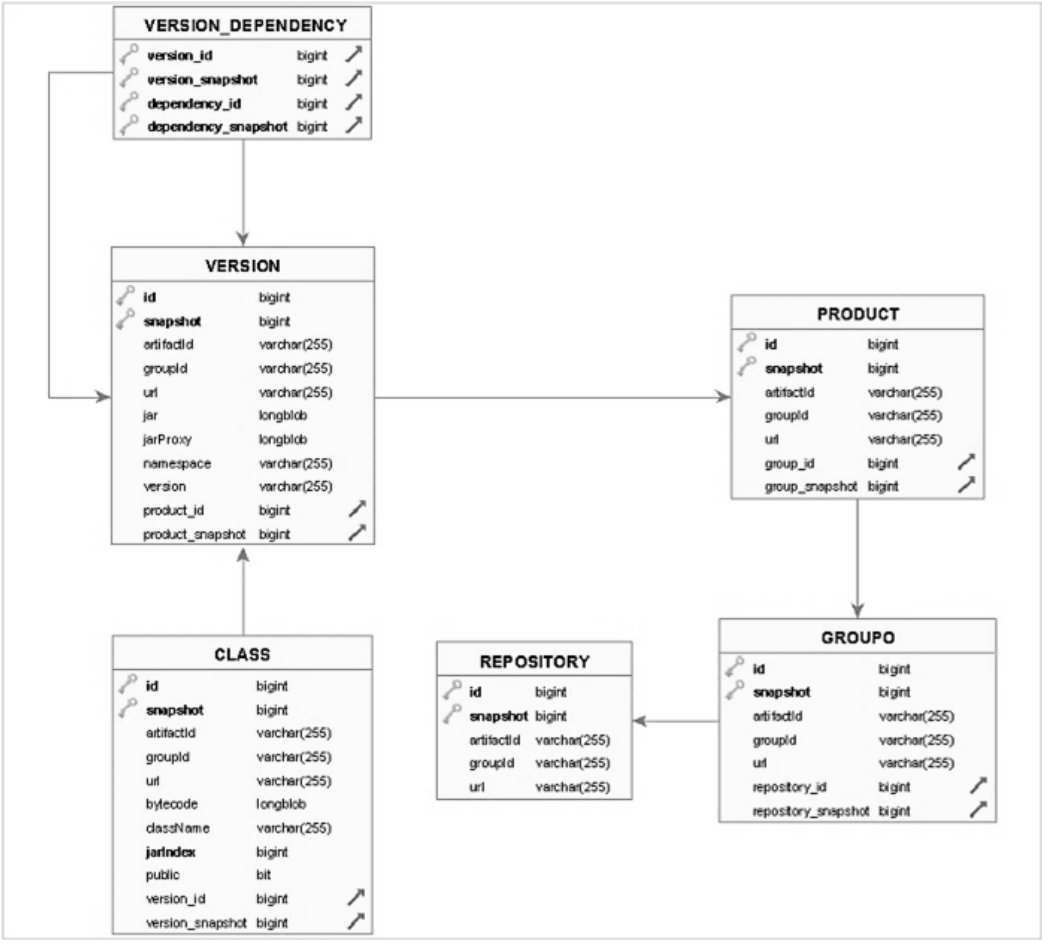


Fig. 26 - Esquema relacional simple para persistir bibliotecas y proxies de bibliotecas.

El soporte Objeto-Relacional está diseñado según la especificación JPA 2.1 (JSR 238)[15]. Este módulo abstrae al módulo 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 concreto.

5.2. Pruebas y Resultados

A modo de ejemplo, se presenta un caso para probar la propuesta con el intérprete de fuentes RSS Informa.

5.2.1. Gestión de dependencias con Maven

RSS Informa es una biblioteca que está publicada en el repositorio Maven y, según el archivo POM, posee 18 dependencias de primero y segundo nivel. Gestionando las

15 https://jcp.org/en/jsr/detail?id=338

dependencias del proyecto con Maven, se descargarán todas esas bibliotecas a una ubicación local y las referenciará como parte del class-path del proyecto (Fig. 27).

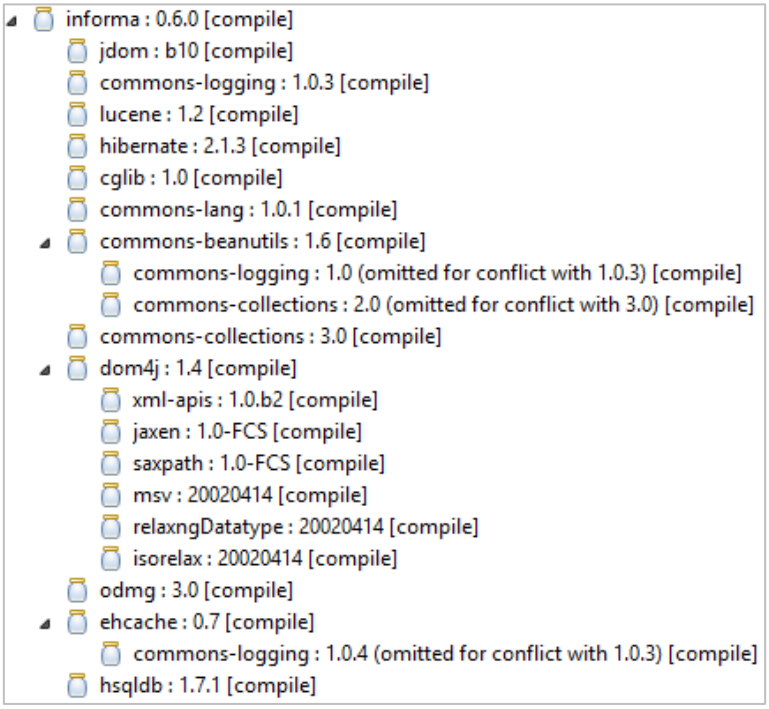


Fig. 27 - Biblioteca informa y sus dependencias.

Como puede observarse en la Fig. 28, el programa del ejemplo son 13 líneas de código, donde se obtiene una fuente en formato XML y se muestran algunos atributos por pantalla.

```
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);
        }
    }
}
```
Fig. 28 - Código fuente de la clase PruebaInforma.

Esta clase PruebaInforma 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 transitivas de Nivel 1, y así sucesivamente.

En la

Tabla 6 se listan las dependencias de la biblioteca Informa y las dependencias transitivas medidas con Deep. En los casos donde se obtiene una medición 0, se estima que están disponibles para ser referenciadas por el programa de usuario.

Tabla 6 - Dependencias de Informa medidas con Deep.

<table>
<tr><th colspan="2">Nivel 0</th><th colspan="2"></th></tr>
<tr><td rowspan="20">prueba-informa</td><th colspan="2">Nivel 1</th><td></td></tr>
<tr><td rowspan="19">informa
(→ 0.0451)</td><th colspan="2">Nivel 2</th></tr>
<tr><td>jdom-b10
(→ 0.1881)</td><td rowspan="6"></td></tr>
<tr><td>commons-logging (→0.6888)</td></tr>
<tr><td>lucene (→0.1259)</td></tr>
<tr><td>hibernate (→0.0260)</td></tr>
<tr><td>cglib (→0.0000)</td></tr>
<tr><td>commons-lang (→0.0000)</td></tr>
<tr><td rowspan="2">commons-
beanutils (→0.0000)</td><td>commons-logging
(→0.6814)</td></tr>
<tr><td>commons-
collections
(→0.0063)</td></tr>
<tr><td>commons-collections
(→0.0191)</td><td></td></tr>
<tr><td rowspan="6">dom4j (→0.0000)</td><td>xml-apis (→0.5694)</td></tr>
<tr><td>jaxen (→0.7939)</td></tr>
<tr><td>saxpath (→0.9233)</td></tr>
<tr><td>msv (→0.0000)</td></tr>
<tr><td>relaxngDataty
(→0.2183)</td></tr>
<tr><td>isorelax (→0.0000)</td></tr>
<tr><td>odmg (→0.0000)</td><td></td></tr>
<tr><td>ehcache (→0.0000)</td><td>commons-logging
(→0.6869)</td></tr>
<tr><td>hsqldb (→0.0000)</td><td></td></tr>
</table>

En el Nivel 0, la tasa de uso de 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.

5.2.2. Gestión de dependencias con Mozo (primer prototipo)

Usando a Mozo para resolver dependencias y desde un proyecto Java de Eclipse, el usuario define las dependencias de Nivel 0, a través de un descriptor POM. Desde el menú contextual, selecciona la opción Solicitar Proxies, que es parte de la sección Mozo (Fig. 29).

Internamente el plug-in solicita al servicio Mozo y éste reenvía la solicitud al middleware, que entregará un proxy de la biblioteca informa.jar. El programa de usuario se compilará, quedando enlazado a los nombres de las clases. Una vez que el programa fue compilado, antes de ejecutarlo, el usuario deberá solicitar recibir las clases referenciadas, también a través de la opción Obtener Clases del menú contextual.

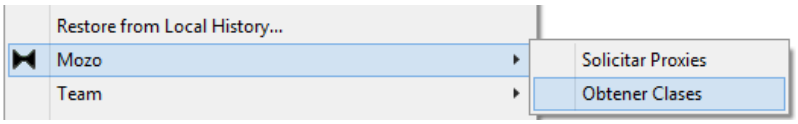


Fig. 29 - Menú contextual de Eclipse con plug-in de Mozo.

El plug-in relevará los imports del código fuente y envía el listado el servicio Mozo que la reenvía al servicio fetchDependencies() de Despachante. Internamente, el middleware obtiene las clases referenciadas en todos los niveles y retorna al cliente (IDE) 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 class-path.

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 class-path jdom-b10, commons-logging y commons-collections, sólo integrados por las clases referenciadas directa o indirectamente por la clase PruebaInforma. En la Tabla 7 se muestran las bibliotecas seleccionadas por Mozo para que el programa PruebaInforma pueda ser ejecutado.

Tabla 7 - Dependencias obtenidas con Mozo y medidas con Deep.

<table>
<tr><th colspan="2">Nivel 0</th><th colspan="2"></th></tr>
<tr><td rowspan="4">prueba-informa</td><th colspan="2">Nivel 1</th><td></td></tr>
<tr><td rowspan="3">informa (→ 0.1423)</td><th colspan="2">Nivel 2</th></tr>
<tr><td>jdom-b10 (→ 0.2618)</td><td rowspan="3"></td></tr>
<tr><td>commons-logging (→0.7089)</td></tr>
<tr><td>common-collections (→0.0170)</td></tr>
</table>

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 6. 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. La demostración de esta prueba está disponible desde un video de acceso público[16].

5.3. Conclusiones del Capítulo

La tecnología Java establece que durante la compilación sólo deben definirse las referencias simbólicas entre clases, postergando al momento de ejecución la necesidad real de contar con esos recursos referenciados.

En este Capítulo se presenta el diseño de un prototipo que analiza el binario Java y, en función de las referencias simbólicas, localiza y suministra dinámicamente las clases de sus dependencias. Asimismo, también se explica de qué modo es posible extraer únicamente archivos puntuales de un Jar/Zip remoto. Con este prototipo funcional se validaron estos conceptos, proponiendo la presencia de un intermediario entre el ambiente de desarrollo y los repositorios de bibliotecas. De este modo se desacopla al cliente de las fuentes (repositorios) mediante un servicio web que suministra representativos o proxies para permitir la compilación (la definición de esas referencias simbólicas en el nuevo binario) para que luego, previo a la ejecución, ese mismo intermediario suministre únicamente las clases referenciadas, tanto directas como indirectas (transitivas).

El prototipo está conformado por un plug-in para el IDE Eclipse (cliente) desde donde el usuario primero solicita representativos de bibliotecas para conseguir la compilación del software en desarrollo, y luego, también desde el plug-in, solicitar el suministro de las clases requeridas. Tanto el servicio web como el plug-in Eclipse, están desarrollados con tecnología OSGi. El intermediario cuenta con un repositorio local de clases y bibliotecas que se completa a medida que recibe solicitudes. De este modo se optimiza el tiempo de respuestas para futuras solicitudes del mismo recurso.

A partir de los resultados alcanzados en este Capítulo, 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 de suministrar dependencias a nivel de biblioteca, a analizar los archivos de clases y transferir sólo el binario requerido por el ambiente de desarrollo.

[16] http://bit.ly/demo-mozo-v1

Capítulo 6

Propuesta actualizada

En base a la experiencia adquirida durante el diseño, construcción y pruebas del primer prototipo, y a modo de actualización para la reciente nueva versión 9 de la plataforma Java, la propuesta de suministrar clases a demanda se reformula adaptando el concepto de suministro de clases a demanda a un nivel de granularidad intermedio entre clases y bibliotecas que es el sistema de módulos.

Según la última versión preliminar de la JLS (Especificación del Lenguaje Java) para la versión 9, cada módulo debe incluir un descriptor: un archivo de nombre fijo 'module-info.java' que será compilado junto con todo el código fuente de módulo [25]. 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 o Jigsaw) establece que por cada módulo se deben definir los módulos que éste 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, sí durante la ejecución.

Jigsaw también modulariza la JDK, pasando de una plataforma monolítica a una modular, que reorganiza los paquetes de la biblioteca única anterior rt.jar[17] en una estructura jerárquica sin ciclos (Fig. 30 y Fig. 31). Es importante observar, que en la plataforma modular todos los módulos requieren directa o indirectamente del módulo java.base (Fig. 31).

17 Tamaño en Java 8: 66MB.

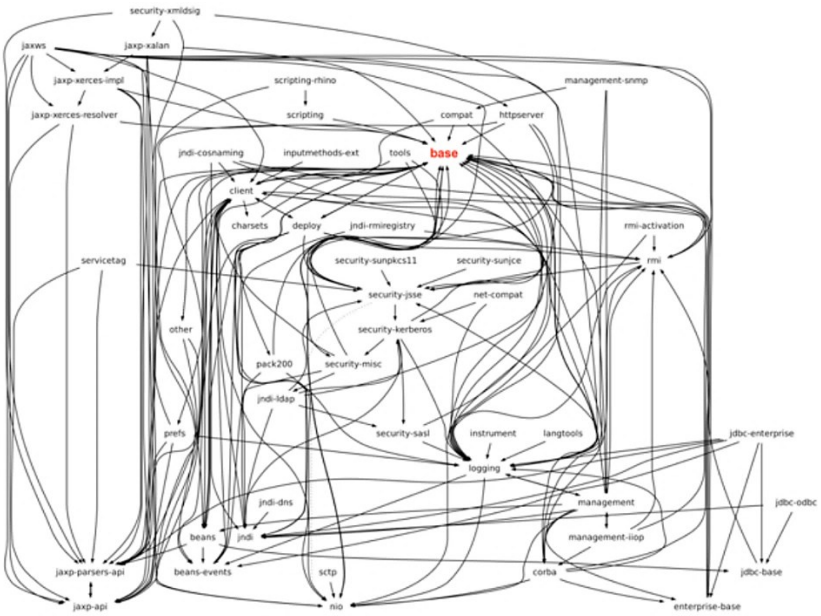


Fig. 30 – Dependencias de la plataforma Java 8.

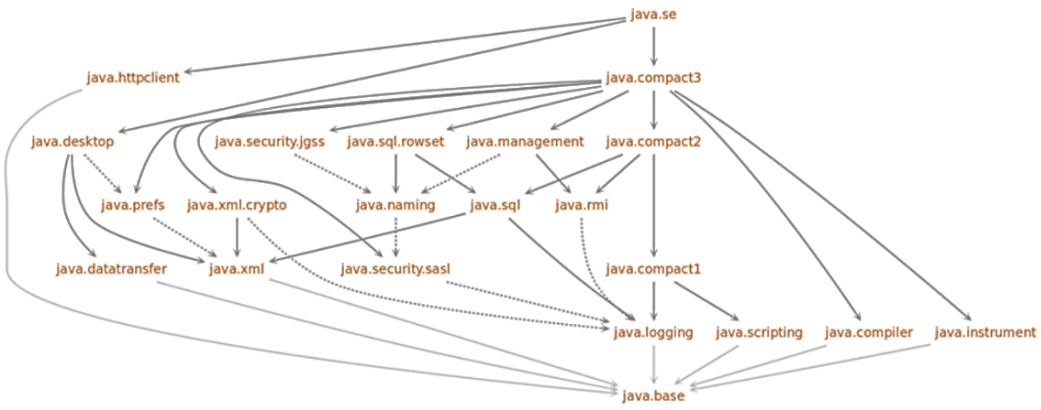


Fig. 31 – Dependencias de la plataforma Java 9.

Al fragmentar la plataforma, se reducen los tiempos de búsqueda de clases, ya que el class-path también queda dividido en módulos (module paths) [44]. De este modo, fragmentando las bibliotecas en módulos más pequeños, se estaría dando solución al problema de subutilización de bibliotecas, documentado en el Capítulo 3 y en trabajos anteriores del autor [45].

Otra innovación del sistema de módulos es la posibilidad de crear imágenes “a medida” para ejecución [9]. Con el programa jlink, se pueden crear plataformas de las que sólo formen parte los componentes y configuraciones que sean requeridos en el ambiente de ejecución [46]. Jlink también permite personalizar con más detalle la imagen, a través de una serie de plug-ins predefinidos (Ver Tabla 8).

Tabla 8 - Plugins disponibles con Java 9 para crear imágenes de plataformas de ejecución.

Nombre del plug-in	Descripción
class-for-name	Optimación de clase. Convierte las llamadas Class.forName a constantes.
compress	Comprime todos los recursos de la imagen. Nivel 0: comprime las constantes de cadenas de caracteres Nivel 1: ZIP Nivel 3: ambos
copy-files	Si los archivos a copiar no están en un path absoluto, se emplea el home del JDK.
exclude-files	Permite especificar qué archivos excluir. Por ejemplo: **.java,glob:/java.base/native/client/**
exclude-jmod-section	Especifica qué módulo excluir.
exclude-resources	Especifica qué recursos se excluirán. Por ejemplo: **.jcov,glob:**:/META-INF/**
include-locales	Permite definir qué idiomas incluir según la RFC 4647.
order-resources	Ordena los recursos según el criterio indicado.
release-info	Permite agregar información acerca de la versión.
strip-debug	Excluye la información para debug de la imagen.
strip-native-commands	Excluye los comandos nativos (tales como java/java.exe) de la imagen.
system-modules	Carga rápida de descriptores de módulos (siempre activo).
Vm	Selecciona la máquina virtual con HotSpot de la imagen.

Si bien está disponible una API para crear más plug-ins, esta funcionalidad apunta a optimizar el despliegue de la aplicación, no sería de utilidad para gestionar el ambiente de desarrollo. Poder crear versiones adaptadas a la plataforma ejecución mediante un enlazador o linker, es sin dudas una solución que cubriría al problema de desempeño en tiempo de ejecución destacado en el Capítulo 3 de esta Tesis.

6.1. Gestores de dependencias

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 preliminar difundida a través de conferencias[18], para Java 9 está previsto que

18 Maven: http://www.slideshare.net/RobertScholte/java-9-and-the-impact-on-maven-projects
Gradle: https://gradle.org/uncategorized/java-components-solving-the-puzzle-with-jigsaw-and-gradle/

las dos herramientas mantengan esta arquitectura e integración directa entre el ambiente de desarrollo y los repositorios de bibliotecas/módulos representada en la Fig. 32.

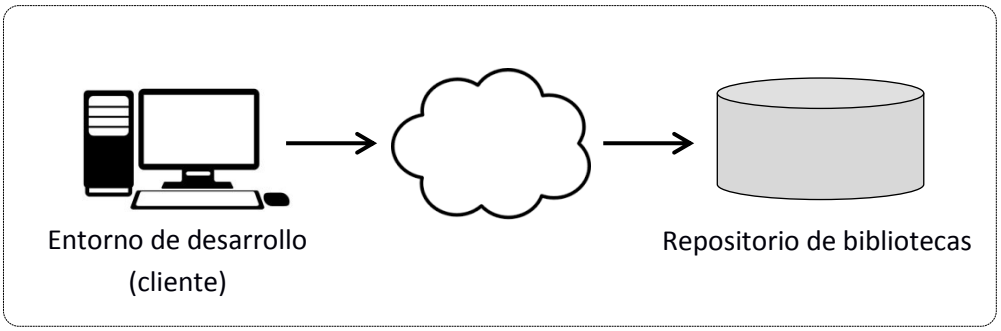


Fig. 32 - Arquitectura general de los Gestores de Dependencias.

Como se mencionó en capítulos anteriores, este modelo trae como consecuencia un alto acoplamiento entre el cliente y los repositorios, que a su vez puede derivar en generar conflictos entre bibliotecas. Por otro lado, la duplicación de información que significa tener, por un lado, un descriptor de módulo Java, y por otro lado un descriptor externo, tampoco es buena práctica para la ingeniería de software.

6.2. Actualización de la propuesta

Al igual que en la primera versión del prototipo, 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 (dependencias nivel 0) del proyecto en desarrollo, y un servicio en la nube que concentre el conocimiento para resolver la clausura y ubicación esas dependencias (Fig. 33).

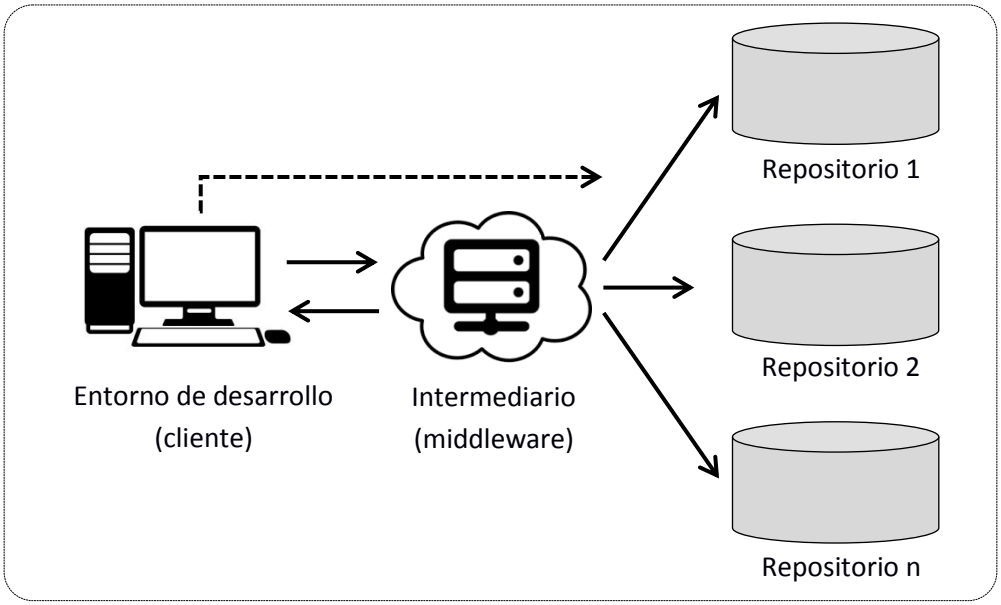


Fig. 33 - Arquitectura general propuesta para la nueva versión del prototipo.

Con esta nueva propuesta, 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 que el cliente sólo envíe las solicitudes de dependencias directas al servicio en la nube para que resuelva clausura y ubicación.

6.2.1. Clausura

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'.

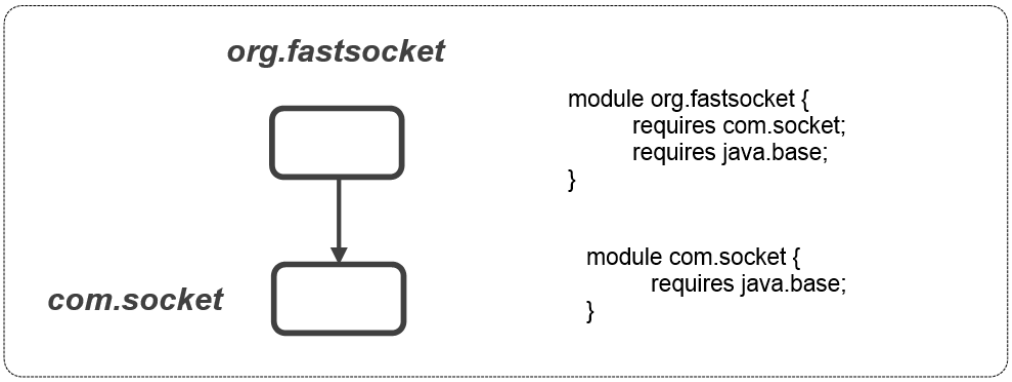


Fig. 34 - Relación entre módulos definida en el descriptor module-info.

La Fig. 34 representa la relación entre el módulo org.fastsocket y com.socket, definida a través del descriptor de org.fastsocket. A su vez, ambos requieren del módulo base. Entonces, con la información disponible en el 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. 35, 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).

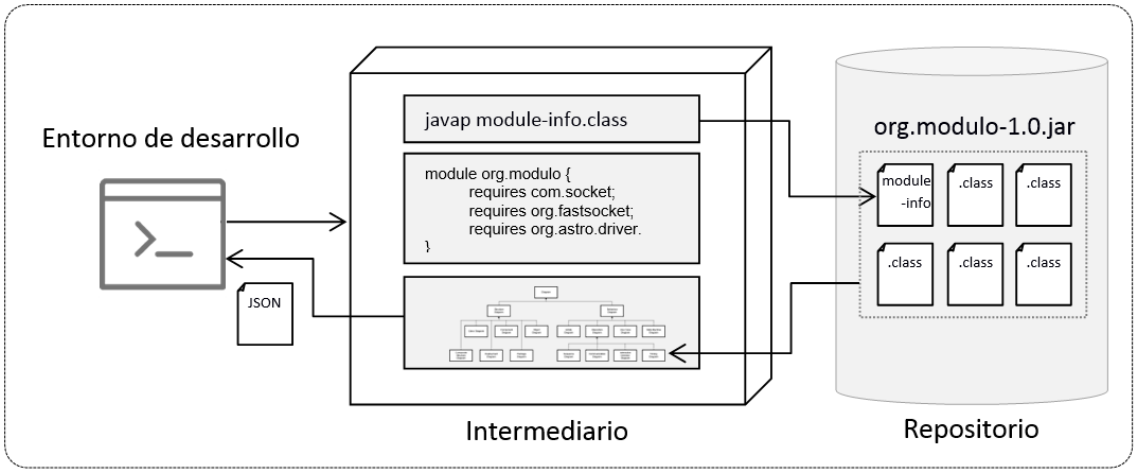


Fig. 35 – Representación dinámica de la nueva propuesta.

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, 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 sería usar el argumento print-module-descriptor para el programa jar. No obstante, este programa también requiere de la presencia local del archivo Jar.

6.2.2. Versiones

Si bien OSGi maneja versiones de módulos, el grupo de expertos responsable de Java 9 prefirió omitir dar soporte a esta característica [47][19]. 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 [48]. Además, se estandariza que los Jars modulares agregan al final del nombre de archivo una cadena con el número de versión. Por ejemplo: com.socket-1.1.jar, y al crear el Jar se agrega el parámetro --module-version=1.0

6.2.3. 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[20], 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 meta-datos.

[19] Neil Bartlett es parte del grupo de expertos responsable de la JSR 376 y miembro activo de la OSGi Alliance.
[20] https://github.com/martinaguero/remotezip

6.2.4. Interfaz con el usuario

Una solución basada en un cliente pesado 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. Para 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. 36).

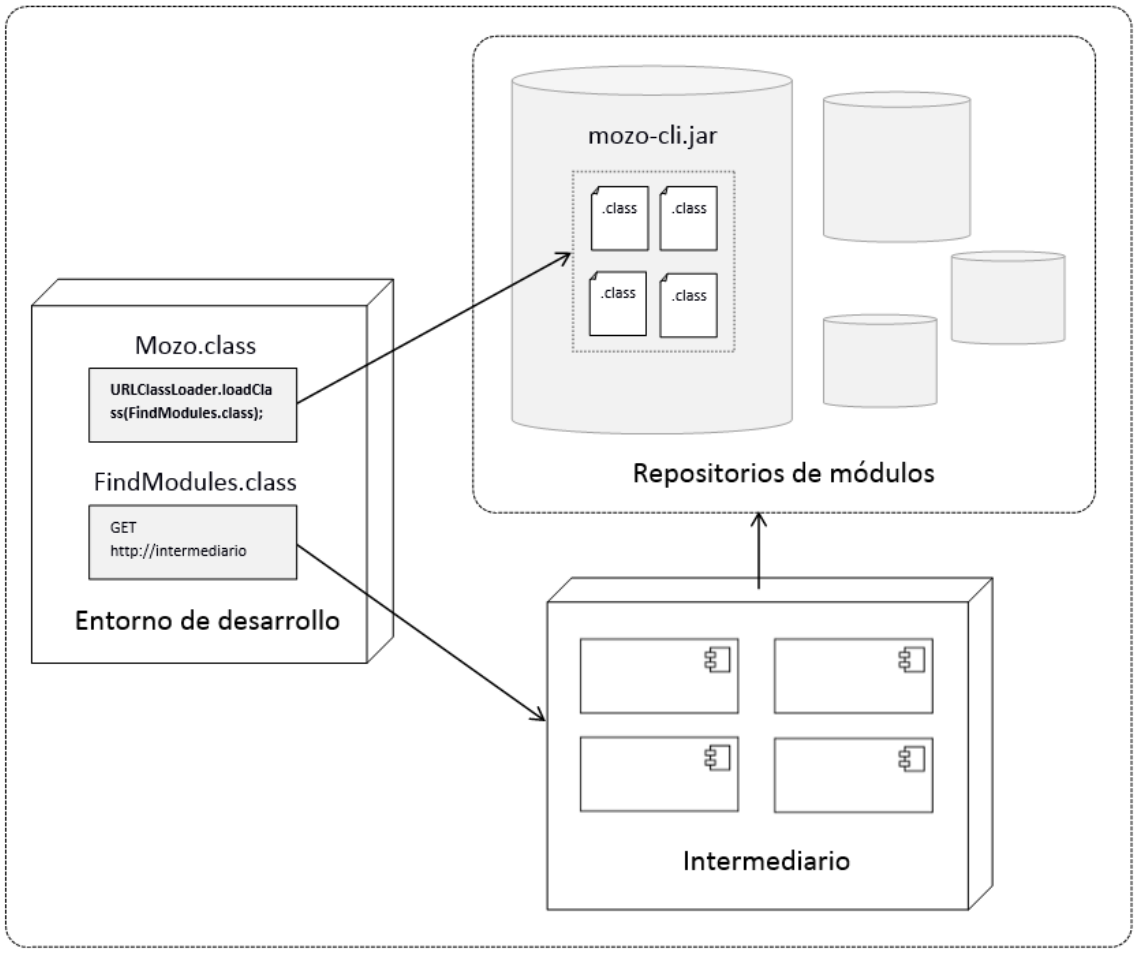


Fig. 36 - Carga remota de clases (entorno de desarrollo).

Entonces, 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 complementaria.

6.3. Conclusiones del Capítulo

El sistema de módulos para la plataforma Java (JPMS) presenta la oportunidad de actualizar el modo como se están gestionando las dependencias de esta tecnología. Según la especificación del lenguaje Java (JLS), en la reciente versión 9, la plataforma pasa de una

arquitectura monolítica, a una integrada por módulos. Lo mismo establece para todo el software creado a partir de ella. Cada módulo requiere la presencia de un archivo descriptor que debe indicar: nombre del módulo, módulos requeridos (sus dependencias), paquetes que exporta, servicios que provee y requiere, entre otros. Al fragmentar las dependencias en módulos y definir espacios de clases (class-path) exclusivos para cada módulo, la misma plataforma estaría dando solución al problema detectado y medido en el Capítulo 3: Subutilización de Dependencias (SD).

Otra innovación de esta nueva versión será la posibilidad de crear imágenes "a medida" de la plataforma de ejecución, donde sólo formarán parte de la instalación aquellos módulos que son referenciados y evitando así que en el class-path haya más clases de las necesarias. Esta solución cubriría el problema de desempeño en tiempo de ejecución (DE) también detectado en el Capítulo 3.

Para las restantes dificultades detectadas en esta Tesis: alto acoplamiento entre el ambiente de desarrollo y los repositorios (AA) y la habitual presencia de conflictos entre bibliotecas (CB) durante la resolución de las dependencias, se orientó la siguiente fase del proyecto.

Los Gestores de Dependencias más utilizados actualmente en la industria del software Java están basados en una arquitectura cliente-servidor y descriptores propietarios agregados a las bibliotecas. También definen la relación entre el cliente (el ambiente de desarrollo) y el servidor (los repositorios de bibliotecas) a través de direcciones estáticas. Esta configuración heredada de los Gestores de Paquetes deriva en que, en muchos casos, la resolución de dependencias requiera una fuerte intervención del programador para lograr la clausura de bibliotecas en su ambiente de desarrollo. A su vez, este modelo cliente-servidor establece la necesidad de instalar un programa en el ambiente de local y será allí donde se creará el plan de resolución para recuperar las bibliotecas necesarias para compilar y ejecutar el programa en desarrollo. Según los adelantos referidos a la compatibilidad con Java 9, tanto Maven como Gradle no prevén cambios significativos de adaptación al JPMS, mantendrán la arquitectura cliente-servidor y también seguirán identificando a los módulos a través de descriptores propietarios.

En este Capítulo se propone una solución basada en un cliente liviano y un servicio en la nube especialista en resolver y ubicar todas las dependencias requeridas por el software en desarrollo. Con esta propuesta se desacopla el ambiente de desarrollo de los repositorios, siendo el intermediario la entidad que determina la ubicación de esas dependencias. Esta solución se basa en analizar los descriptores de módulos y a partir de la información concentrada allí, resolver y localizar todas las dependencias requeridas para compilar y ejecutar. Al igual que la primera versión presentada en los Capítulos 4 y 5 de esta Tesis, esta nueva propuesta contempla extracción remota de archivos Jar/Zip (cada módulo también es un Jar), en este caso del descriptor de módulo de donde el intermediario confirmará la localización del módulo objetivo y también para conocer las dependencias del módulo en análisis.

De este modo la resolución de dependencias se traslada de un ambiente de desarrollo fuertemente ligado a la configuración manual del descriptor de proyecto, a un servicio especialista en la nube que analiza y resuelve, a demanda, las dependencias de software Java.

Capítulo 7

Segundo Prototipo

En base a la experiencia de construir el primer prototipo y con el objetivo de adaptarlo al sistema de módulos que propone JDK 9, se rediseñó la solución reutilizando algunos componentes de la versión anterior.

7.1. Diseño e Implementación

Siguiendo las definiciones conceptuales de: Crear un servicio que analice solicitudes de módulos y de respuesta contemplando las dependencias transitivas y casos particulares, se diseñó un prototipo a partir de un estilo arquitectónico cliente – servidor y un modelo de integración de arquitectura orientada a servicios (SOA) [49] [50] [51]. El sistema se divide en 3 componentes principales:

1. Entorno de desarrollo (cliente).
2. Intermediario (servicio en la nube).
3. Repositorio de módulos.

El diseño general de esta nueva propuesta se puede representar con el siguiente diagrama:

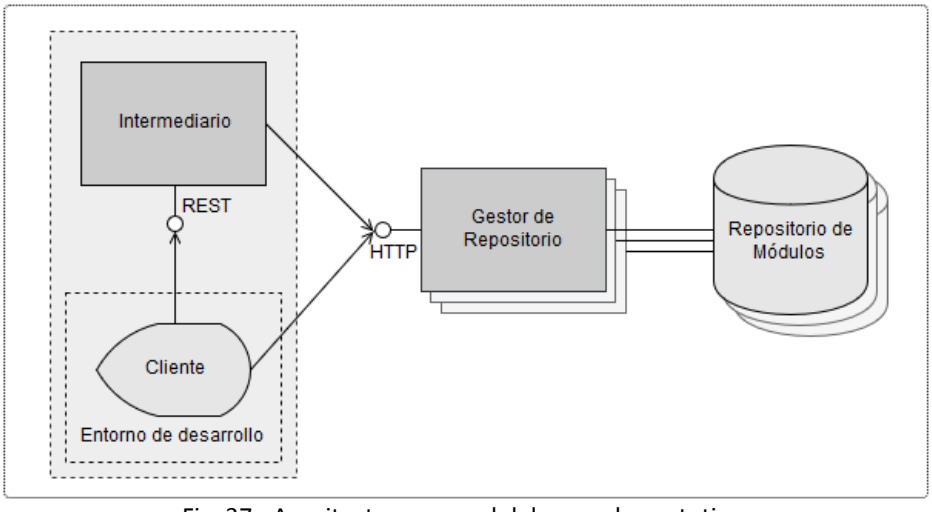


Fig. 37 - Arquitectura general del segundo prototipo.

Como muestra la Fig. 37, un servidor RESTful recibe solicitudes del cliente y responde con un listado de módulos organizados jerárquicamente y representados en formato JSON. Al igual que en la propuesta original, el repositorio es una entidad independiente del

intermediario, permitiendo sumar otros repositorios como fuente de módulos. Cabe destacar que en esta versión no existe un repositorio local o caché. La tarea del middleware (intermediario) se centraliza en analizar los módulos necesarios para conseguir la clausura de dependencias y sólo proporciona la dirección a esos módulos.

7.1.1. Interfaz con el usuario (cliente)

El modo de interactuar del usuario es a través de la interfaz de línea de comandos (CLI o command line interface). El cliente es un programa de consola desde donde se envían solicitudes al middleware. Como se explica en el capítulo anterior, el programa cliente se inicia desde una clase principal que, en función de las acciones del usuario, carga a demanda y desde una ubicación remota, las clases que ofrecen esa funcionalidad (con el cargador de clases remotas URLClassLoader). Este modelo permite realizar ampliaciones y correcciones tanto en la aplicación cliente como el servidor, de manera transparente para el usuario. De este modo, el usuario siempre ejecuta la última versión del sistema, sin necesidad de solicitar reinstalar software en su entorno local.

Como se ve en el código fuente de la Fig. 38, el método loadClass() carga la clase indicada desde el método findModules().

```
private static Class loadClass(String className) {
           URL[] classLoaderUrls = new URL[] { new URL(jarpath) };
           URLClassLoader loader = new URLClassLoader(classLoaderUrls);
           return Class.forName(className, true, loader);
    }

private static void findModules(String target) {
           Class c = loadClass("org.trimatek.mozo.cli.FindModules");
           Method m = c.getMethod("exec", String.class);
           String key = "res" + cont++;
           results.put(key, (String) m.invoke(c.newInstance(), target));
           printResult(key);
           System.out.println("resultado guardado en: " + key);

    }
```

Fig. 38 - Carga remota de clases en el cliente.

La clase FindModules, que es cargada remotamente a través de la clase URLClassLoader, de esta forma se desacopla la funcionalidad de buscar módulos del entorno local de ejecución. El diagrama de clases de la Fig. 39, representa la estructura interna del cliente.

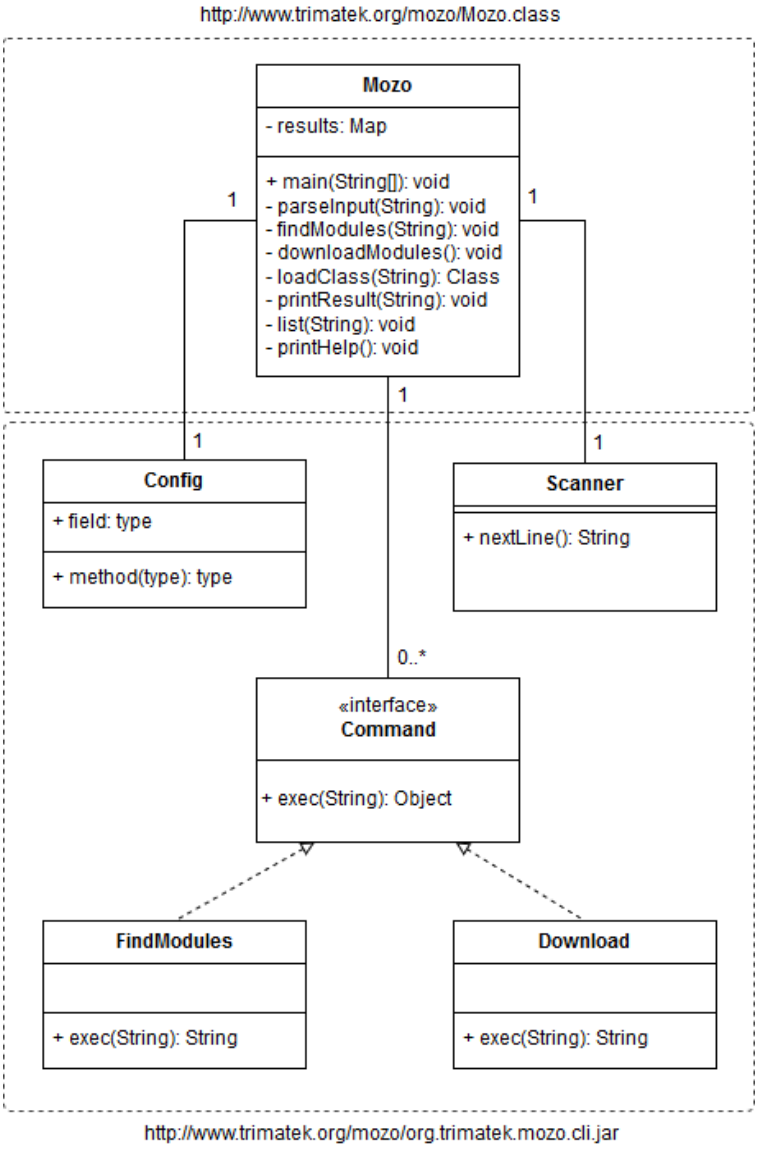


Fig. 39 - Diagrama de clases del cliente.

Todo el código fuente del cliente está disponible en el Anexo II de esta Tesis.

7.1.2. Intermediario

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 [52]. Internamente, su diseño se puede representar con el diagrama que muestra la Fig. 40.

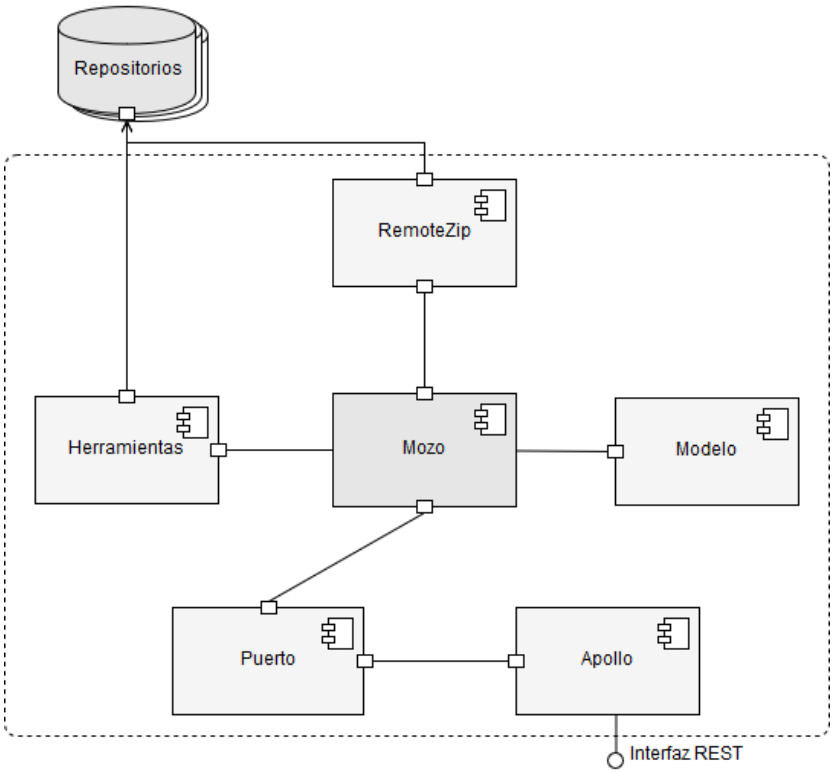


Fig. 40 - Diagrama de componentes del intermediario.

La funcionalidad principal de este servicio 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. 41.

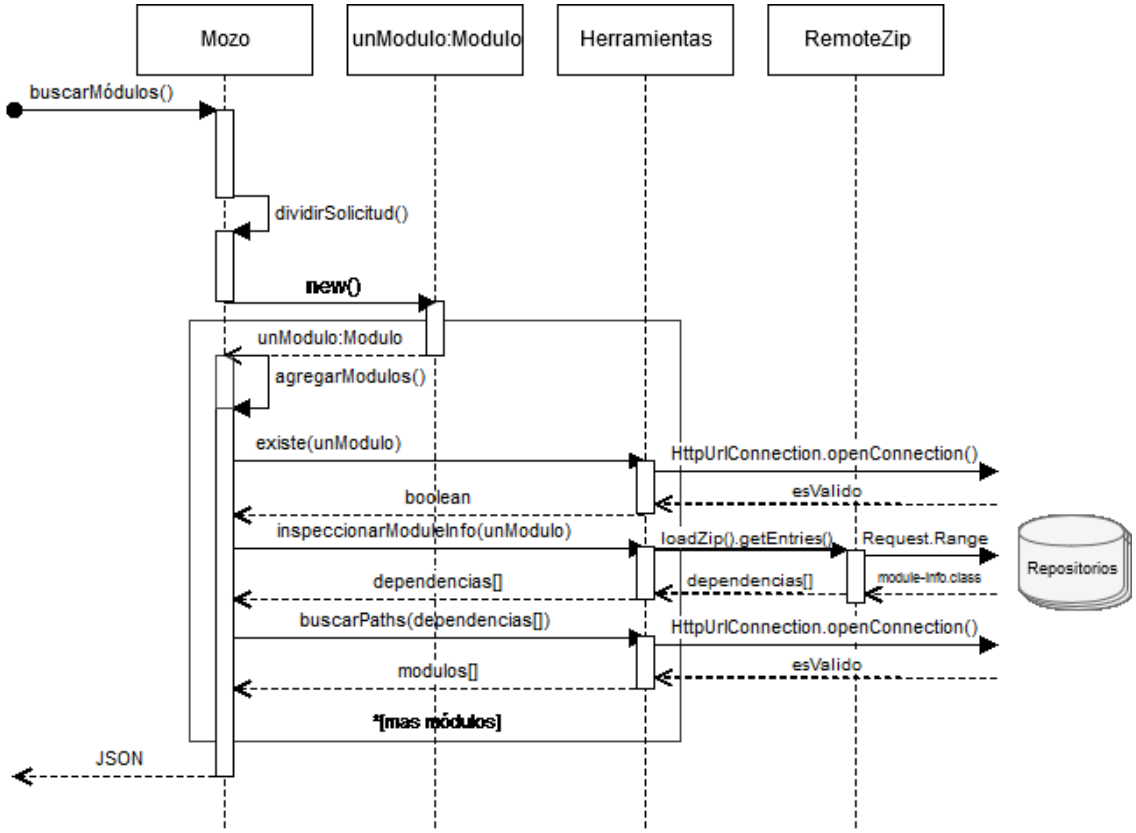


Fig. 41 - Diagrama que representa la secuencia de acciones al buscar módulos.

El resultado para una solicitud de resolución de dependencias es una estructura en formato JSON como se ve en la Fig. 42:


```
{
 "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"
      }
     ]
    }
   ]
```

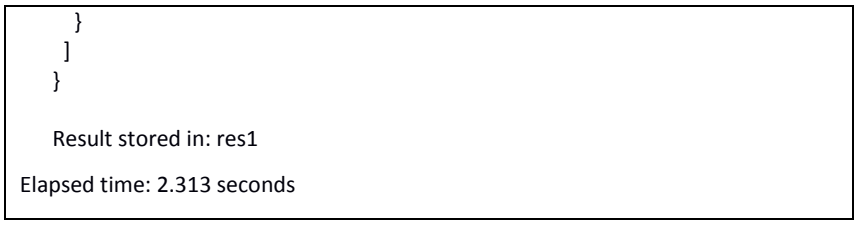


Fig. 42 - Respuesta a una solicitud de módulo.

El cliente recibe un árbol jerárquico donde los módulos indicados en la solicitud del usuario integran la raíz, con sus respectivas ubicaciones y los demás módulos requeridos en sus descriptores. Todo el código fuente del intermediario está disponible en el Anexo I de esta Tesis.

7.1.3. Repositorio

Esta evolución del prototipo no contempla un caché de bibliotecas, dado que en ningún caso entrega ni bibliotecas ni representativos, sólo direcciones validadas a ellos. Como se muestra en la Fig. 37, el entorno de desarrollo no precisa conocer los repositorios que poseen cierto módulo. Es el intermediario el que posee las rutas a los repositorios y, luego de validar la disponibilidad de los módulos requeridos, pasa ese dato al cliente.

7.1.4. Implementación

Para crear este prototipo, en una primera instancia se evaluó utilizar la versión de acceso temprano (EA o early access) de Java 9 [53]. 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 disponible al momento de desarrollar este prototipo, la versión disponible (build 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 [54].
Esta nueva versión del prototipo también se llama Mozo y está disponible en el repositorio del proyecto: https://github.com/martinaguero/mozo. Son los proyectos org.trimatek.mozo.dock (intermediario) y org.trimatek.mozo.cli (cliente).

7.1.5. Uso

Para usar el prototipo, se debe acceder al sitio: http://trimatek.org/mozo/ y descargar la clase Mozo.class disponible en http://trimatek.org/mozo/Mozo.class
La consola 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 res1, res2, 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 (res1, res2, resN). Al invocar este comando, la clase Mozo carga remotamente la clase Download e instancia y ejecuta hasta 5 threads de descarga en simultáneo.

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.

7.2. Pruebas y resultados

En esta segunda fase, se desarrolló un prototipo para evaluar la factibilidad de establecer un servicio intermediario, especialista en resolver dependencias de módulos. El objetivo es presentar un modelo a escala, que valide esta alternativa para la gestión de dependencias. En este caso, la resolución es a nivel módulo, quedando en un punto intermedio entre la resolución a nivel biblioteca (habitual) y a nivel clase Java (prototipo versión 1).

El nivel de granularidad esperado para un módulo, puede dimensionarse de manera preliminar a partir de la configuración de la versión temprana (EA) de JDK 9. En esa distribución, la biblioteca principal (rt.jar) fue fragmentada en 22 módulos independientes. Se espera que el software creado a partir de esta plataforma, siga una proporción equivalente. Es decir, de 1:20. Teniendo en cuenta las mediciones de la Tabla 3 del Capítulo 3, donde el promedio general de utilización no supera al 10%, se espera que con el sistema de módulos esta subutilización de bibliotecas disminuya significativamente.

7.2.1. Caso de prueba

Se diseñó un caso de un sistema de estadísticas, donde los módulos que lo conforman son com.stats.cli, com.stats.core, com.google.guava, org.apache.math, org.apache.rng (Fig. 43).

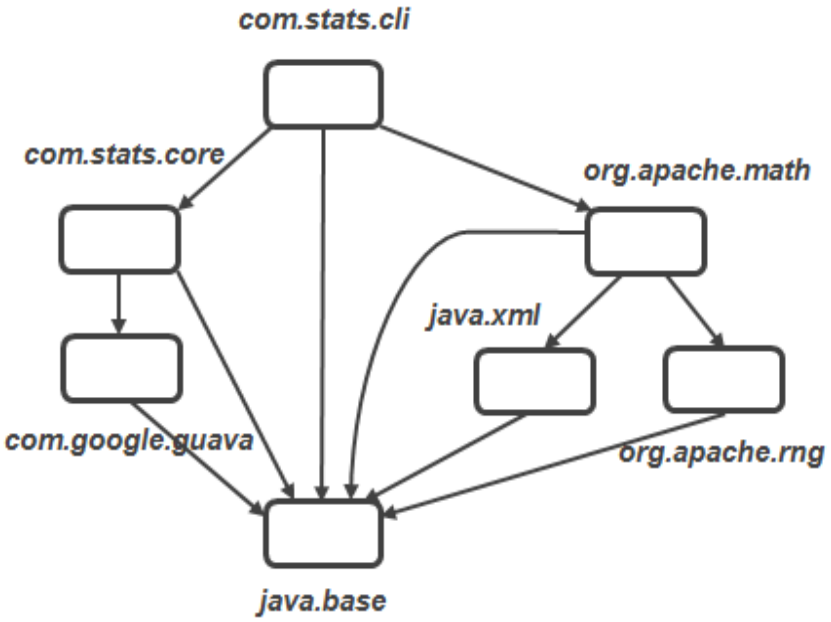


Fig. 43 - Módulo com.stats.cli y dependencias.

En este caso[21], el módulo com.google.guava[22] 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. 44.

21 Basado en http://openjdk.java.net/projects/jigsaw/talks/intro-modular-dev-j1-2016.pdf

22 La biblioteca Guava real tiene 10 dependencias.

```
mozo> fm com.stats.cli
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"
      }
     ]
    }
   ]
  }
 ]
}
```

Fig. 44 - Respuesta del intermediario.

5. Mostrará por consola el árbol jerárquico de la Fig. 44 en formato JSON, donde en la raíz está el nodo con la solicitud del usuario. En el siguiente nivel aparece el módulo com.stats.cli, y luego, los módulos requeridos por los anteriores. En cada rama con un módulo, también aparece el campo path con la dirección verificada al módulo. No necesariamente todos deben estar en el mismo repositorio.
6. Por último, el usuario ingresa el comando:

```
download-modules res0
```

7. Entonces, la clase del cliente en ejecución (Mozo.class) cargará la clase Download disponible en http://trimatek.org/mozo/org.trimatek.mozo.cli.jar. Luego, se invocará al método exec() 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, tal como muestra la Fig. 45:

```
mozo> dm res1
Downloading: com.stats.cli.jar
Downloading: org.apache.math.jar
Downloading: org.apache.rng.jar
Downloading: com.stats.core.jar
Downloading: com.google.guava.jar
Total downloaded: 5
Elapsed time: 0.429 seconds
mozo>
```

Fig. 45 - Salida por consola durante la descarga local de módulos.

8. Ahora, el usuario dispone en su entorno local el módulo objetivo (com.stats.cli) y todas sus dependencias.

7.2.2. Supuestos

El módulo com.stats.cli y sus dependencias están en, al menos, uno de los repositorios que son parte del listado de fuentes del intermediario. No aparecen en el resultado los módulos java.base y java.xml, porque forman parte del JDK.

7.2.3. Escalabilidad

En esta versión del prototipo, la Interfaz Hombre-Máquina (IHM) es una consola de comandos. Como el intermediario recibe solicitudes a través de un conector REST, fácilmente se puede agregar una interfaz Web o una aplicación de escritorio activable desde JNLP [55].

7.2.4. Ventajas conceptuales

- No es una solución invasiva, es decir, no requiere sumar metadatos a los módulos.
- La resolución de la clausura no depende del listado de fuentes (repositorios) definidos en el archivo descriptor del proyecto/módulo. Es el intermediario el que conoce esas fuentes y luego de verificar disponibilidad, establece esa relación en el JSON del resultado. De este modo se desacopla el proyecto de software de los repositorios.
- 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 computación cognitiva.

7.2.5. Ventajas de la arquitectura

- El cliente siempre ejecuta la última versión.
- Las rutas (paths) a los módulos siempre son verificadas.
- Los tiempos de respuesta son aceptables, dado que sólo se transfiere al intermediario el archivo descriptor del módulo [56].
- Concentrar la solución en un servicio en la nube permite realizar permanentes ampliaciones y correcciones de forma transparente para el usuario.

7.3. Conclusiones del Capítulo

En este Capítulo se describe una implementación a modo de prueba de conceptos presentados en el Capítulo 6. El sistema está integrado por 3 partes principales: un cliente (ambiente de desarrollo), el intermediario (servicios REST) y los repositorios de módulos (hosting web con transferencia http/https de terceras partes).

En esta versión, el intermediario recibe solicitudes de resolución de dependencias (módulos) desde un cliente de línea de comandos (CLI) y luego de "visitar" los repositorios en busca de los módulos requeridos, retorna un listado de rutas (paths) a esas dependenciasen formato JSON. En esta versión no hay repositorio local de módulos o clases, el intermediario únicamente extrae los descriptores, analiza sus dependencias y así recursivamente hasta completar la construcción del árbol de módulos. Cabe destacar que, para agilizar el tiempo de respuesta, el intermediario sí guarda en memoria una copia de cada descriptor recuperado, junto la ruta al origen. De este modo, en las sucesivas resoluciones en las que esté involucrado un módulo ya localizado anteriormente, éste no deberá volver a ser ubicado entre los repositorios conocidos por el intermediario.

Este prototipo también utiliza el subproyecto Remote Zip para extraer de archivos Jar remotos los descriptores (module-info.class) que luego se descompila para analizar sus dependencias.

El cliente se lo considera liviano dado que sólo es una clase Java que al ejecutarla, activa una interfaz de comandos desde donde se ingresa el nombre de las dependencias directas y, en función al comando ejecutado, la clase base carga remotamente (mediante el class loader URLClassLoader) las específicas según la operación seleccionada. De este modo el cliente siempre ejecuta la última versión de forma transparente para el usuario. Una vez que el cliente recibe ese árbol de dependencias, el usuario ingresa el comando para descargar que, luego de cargar remotamente la clase específica, se inicia la descarga desde los repositorios donde el intermediario localizó los módulos hacia el cliente (el ambiente de desarrollo). Para futuras versiones está previsto sumar una interfaz web para como cliente.

Se puede afirmar que esta propuesta no es una solución invasiva, es decir, no requiere sumar metadatos a los módulos. Otra característica es que la resolución de la clausura no depende del listado de fuentes (repositorios) definidos en el archivo descriptor del

proyecto/módulo, como ocurre actualmente con los gestores de dependencias, sino que es el intermediario quien conoce esas fuentes y luego de verificar su disponibilidad, establece esa relación en el JSON del resultado. De este modo se desacopla el ambiente de desarrollo de los repositorios. El intermediario es un "especialista" en resolver dependencias, conoce las fuentes y puede, a lo largo de sucesivas versiones, incorporar algoritmos más sofisticados para tratar casos de conflictos entre módulos, como por ejemplo herramientas de computación cognitiva.

Además, con esta arquitectura se garantiza que el cliente siempre ejecute la última versión, que las rutas (paths) a los módulos siempre son verificadas. También, al concentrar la solución en un servicio en la nube (tanto del cliente como el servidor), esto permite realizar ampliaciones y correcciones de forma transparente para el usuario.

Capítulo 8

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 [57] [58]. La última implementación de Gestor de Paquetes para Windows se denomina PackageManagement (anteriormente OneGet) e incorpora como novedad ser una agregación de Gestores de Paquetes [59].

Básicamente, estos gestores de paquetes son aplicaciones de ejecución local que analizan las dependencias asociadas a cada paquete y, recursivamente, descargan los requeridos para alcanzar la clausura.

La portabilidad este modelo 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.

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

8.1. 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 [60].

Comentario: Esta particularidad asegura que la versión utilizada es exactamente la creada por el autor, no obstante, tiene como desventaja el acoplamiento que esto significa.

8.2. 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 [61].

Comentario: Esta fragmentación que significa tener asociado a un proyecto varios archivos de requerimientos, más otros con restricciones y las diferentes posibilidades de combinaciones, puede impactar en la gestión de la configuración de un proyecto de software. Además, el requerimiento de configuración de tantos aspectos particulares de un proyecto, no favorece la reutilización y refinamiento del conocimiento.

8.3. 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 versión [62].

Comentario: Si bien se denomina gestor de paquetes, es un gestor de dependencias, donde un paquete puede estar integrado por una o más bibliotecas. Permite personalizar la configuración del cliente a través del archivo Nuget.Config y definir reglas particulares para las dependencias del proyecto en el archivo project.json. No presenta innovaciones respecto de otros gestores de paquetes/dependencias.

8.4. 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 [63]. Cuando se activa esta funcionalidad, Composer descargará a demanda las clases referenciadas en el código fuente [64].

Comentario: Es un proyecto joven, que se enfoca en la gestión de dependencias. Incorpora una implementación del estándar Autoloader para PHP, documentado y aprobado en la PSR-04 [65].

8.5. 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 [73].

Comentario: Está orientado a gestionar recursos visuales web y se diferencia de otros gestores de paquetes, porque optimiza la carga de recursos a través de una implementación de Flat Dependency Graph [74].

8.6. 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) [66]. 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 [67].

Comentario: Como en sbt la gestión de dependencias es responsabilidad de Apache Ivy, no presenta innovaciones respecto de ésta. A continuación se presentará una breve introducción a Ivy donde se destacarán sus aspectos más importantes y originales.

8.7. 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 [66].

Comentario: Apache Ivy es un subproyecto de la herramienta de construcción de proyectos Java Apache Ant [68]. Si bien ofrece mayor flexibilidad que otras herramientas, también se distribuye como un cliente que concentra toda su capacidad analítica en un ambiente local y que depende principalmente de la parametrización del archivo de configuración del proyecto y los metadatos de las bibliotecas. La última versión es del año 2014.

8.8. Apache Maven (Java):

Es una solución integrada para la gestión de proyectos Java [69]. 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 [70].

Comentario: Al igual que otras soluciones de gestión de dependencias, Maven tiene sus raíces arquitectónicas y conceptuales en los gestores de paquetes y el programa Make [71]. No hay dudas que desde su aparición en 2004 y con sus sucesivas versiones 2 y 3 (2005 y 2010) ha contribuido positivamente a las demandas del momento de su creación [72]. No obstante, su capacidad analítica está principalmente limitada a la parametrización manual del proyecto a través de un archivo XML y la funcionalidad adicional de una serie de plug-ins específicos para la resolución de conflictos en las dependencias. Recientemente liberó un plug-in de compatibilidad con JDK 9 (maven-compiler-plugin 3.6.1).

8.9. 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 una herramienta de soporte a la construcción de software con 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 [73].

Comentario: Sin dudas que Gradle ha sumado innovaciones respecto de sus antecesores. Sin embargo, deriva de un diseño centrado en la estación de trabajo. Posiblemente esta sea la más apropiada configuración para una herramienta de construcción, no así, para un sistema de gestión de dependencias donde la capacidad resolutiva está limitada a los algoritmos del cliente y las configuraciones del proyecto.

8.10. 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 [76].

Comentario: Su implementación aún está en una fase experimental, por lo que será necesario esperar unos años para conocer si el paradigma funcional es adecuado para interactuar y organizar paquetes.

8.11. 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 [77]. 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.

Comentario: Si bien la computación de altas prestaciones presenta desafíos particulares, las innovaciones propuestas por Spack podrían ser adaptadas a los requerimientos de la industria.

8.12. Conclusiones del Capítulo

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 y no los mecanismos de soporte alrededor de él.

Este breve relevamiento de tecnologías tiene como objetivo destacar las cualidades particulares de cada solución. Asimismo, descubrir cuáles son las características que influyeron positivamente en la actividad y fueron adoptadas por la mayoría. También se pone especial atención en las soluciones emergentes que pueden ser el factor clave hacia el futuro.

A modo de resumen, se puede mencionar que la tendencia apunta a emplear lenguajes específicos del dominio (DSL) para describir paquetes/bibliotecas o proyectos. También se observa que JSON estaría reemplazando a XML como recurso para representar los metadatos asociados al proyecto.

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.

Si bien la autocarga (autoload) de clases es una técnica que existe desde hace más 30 años, aún no ha sido masivamente adoptada como sí ocurre, por ejemplo, con el enlace dinámico entre clases. Actualmente, con la ubicuidad y velocidad de Internet, autoload es un recurso que está pendiente evaluar su empleo para los ambientes de desarrollo contemporáneos.

La repetitividad de las construcciones es un tema pendiente de muchas soluciones de gestión de dependencias y construcción de software. Algunas herramientas como Bundler, Pip y Gradle proponen recursos para asegurar copias de la construcción en diferentes ambientes. Otras herramientas no cubren este aspecto.

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.

Tanto sbt como Ant delegan en Ivy la gestión de dependencias. Si bien Ivy es un proyecto que desde 2014 no tiene actualizaciones, los principios de reutilización y especialización también deberían ser aprovechados en estas herramientas.

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.

Capítulo 9

Evaluación de la propuesta

A continuación, se desarrolla un estudio donde inicialmente se caracteriza la propuesta a partir de una serie de perspectivas basadas en atributos cualitativos para luego dimensionar el desempeño a través de un estudio comparativo cuantitativo.

9.1. Acoplamiento

La indirección que se consigue al no acoplar un descriptor de módulo o biblioteca a un repositorio específico, elimina la rigidez de las soluciones donde los metadatos deben especificar la dirección de las fuentes (repositorio). Con esta propuesta es el intermediario quien conoce todos los repositorios y a donde se dirige en busca de los módulos involucrados en la resolución de las dependencias. Comentario: Se puede afirmar que con esta propuesta el acoplamiento entre módulos es nulo dado que el intermediario es quien debe localizar dinámicamente las dependencias.

9.2. Eficiencia

Durante la resolución de la clausura de dependencias, el intermediario sólo obtiene la porción de bytes que representa al descriptor de módulo. En ningún momento el módulo es transferido desde el repositorio al intermediario, sólo el archivo module-info.class que contiene el listado de dependencias (requires) y recursos que expone (exports). Asimismo, el intermediario, guarda una copia de los módulos resueltos en un caché local, por lo tanto, en las próximas solicitudes de resolución no necesitará repetir el acceso remoto al repositorio. Comentario: De este modo además de minimizar la transferencia de bytes desde los repositorios al intermediario, se optimiza la eficiencia al contar con un pool de descriptores previamente transferidos.

9.3. Rendimiento

Si bien el prototipo se probó en una instancia de servidor privado virtual (VPS) de recursos algo limitados: procesador de 1 núcleo de 2.4 GHz y 1 GB de memoria RAM, como se muestra en la Fig. 46, durante las pruebas de validación éste en ningún momento alcanzó ningún límite, ni de CPU, RAM o ancho de banda.

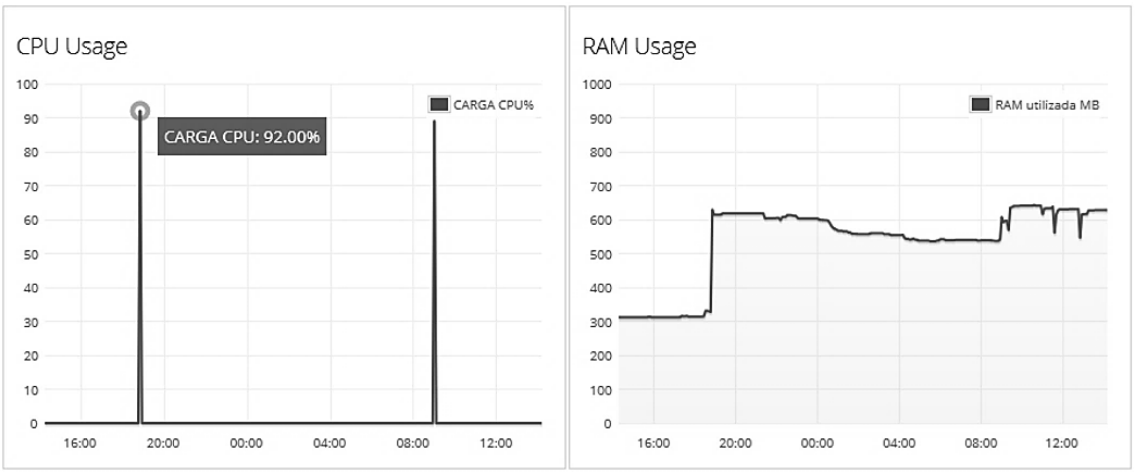


Fig. 46 - Historial de uso de recursos en VPS con el prototipo.

Comentario: Por más que el prototipo se ejecute en una máquina virtual de Java, como se trata de un diseño simple basado en web services (REST) y procesamiento de archivos descriptores de módulos que habitualmente no superan los 5 KB de tamaño, esta solución no requiere de gran capacidad de procesamiento ni ancho de banda. Como se verá a continuación, se consiguen buenos resultados de desempeño aún en ambientes con recursos limitados.

9.4. Validación

Para evaluar aspectos funcionales de la propuesta, se retomó el caso presentado en la Sección 3.1 de esta tesis.

9.4.1. Caso programa Quickstart de GeoTools, Parte 3:

Con el objetivo de obtener la clausura de dependencias necesaria para ejecutar el ejemplo Quickstart de la herramienta GeoTools, se diseñó un caso de prueba donde a cada dependencia (directa y transitiva) se le agregó un descriptor de módulo requerido para Java 9. De este modo se consigue igualar la situación, estableciendo una relación de "1 biblioteca = 1 módulo" a fin de crear un caso comparable con gestores de dependencias más utilizados en la industria como son Apache Maven, Apache Ivy y Gradle. Vale aclarar que aún no existen versiones compatibles con Java 9 de GeoTools[23], por lo tanto, la decisión fue la de "modularizar" las dependencias de GeoTools 14.3, creando descriptores de módulos a partir de los archivos POM que definen las dependencias de las bibliotecas.

[23] http://docs.geotools.org/latest/userguide/build/install/jdk.html

9.4.1.1. Lote de pruebas

El ejemplo Quickstart requiere un total de 60 bibliotecas (2 directas y 58 transitivas). En la Fig. 47 se muestran resaltadas las 2 dependencias directas (gt-shapefile y gt-swing) y las transitivas (sin repeticiones).

```
org.geotools:Quickstart:jar:0.0.1-SNAPSHOT
+- org.geotools:gt-shapefile:jar:14.3:compile
|  +- org.geotools:gt-data:jar:14.3:compile
|  |  \- org.geotools:gt-main:jar:14.3:compile
|  |     +- org.geotools:gt-api:jar:14.3:compile
|  |     \- com.vividsolutions:jts:jar:1.13:compile
|  +- org.jdom:jdom:jar:1.1.3:compile
|  \- javax.media:jai_core:jar:1.1.3:compile
\- org.geotools:gt-swing:jar:14.3:compile
   +- org.geotools:gt-referencing:jar:14.3:compile
   |  +- com.googlecode.efficient-java-matrix-library:core:jar:0.26:compile
   |  +- commons-pool:commons-pool:jar:1.5.4:compile
   |  +- org.geotools:gt-metadata:jar:14.3:compile
   |  |  \- org.geotools:gt-opengis:jar:14.3:compile
   |  |     \- net.java.dev.jsr-275:jsr-275:jar:1.0-beta-2:compile
   |  +- jgridshift:jgridshift:jar:1.0:compile
   |  \- net.sf.geographiclib:GeographicLib-Java:jar:1.44:compile
   +- org.geotools:gt-render:jar:14.3:compile
   |  +- org.geotools:gt-coverage:jar:14.3:compile
   |  |  +- javax.media:jai_imageio:jar:1.1:compile
   |  |  +- it.geosolutions.imageio-ext:imageio-ext-tiff:jar:1.1.13:compile
   |  |  |  +- it.geosolutions.imageio-ext:imageio-ext-utilities:jar:1.1.13:compile
   |  |  |  +- it.geosolutions.imageio-ext:imageio-ext-geocore:jar:1.1.13:compile
   |  |  |  |  \- it.geosolutions.imageio-ext:imageio-ext-streams:jar:1.1.13:compile
   |  |  |  \- javax.media:jai_codec:jar:1.1.3:compile
   |  |  +- org.jaitools:jt-zonalstats:jar:1.4.0:compile
   |  |  +- org.jaitools:jt-utils:jar:1.4.0:compile
   |  |  +- it.geosolutions.jaiext.affine:jt-affine:jar:1.0.8:compile
   |  |  |  +- it.geosolutions.jaiext.iterators:jt-iterators:jar:1.0.8:compile
   |  |  |  +- it.geosolutions.jaiext.utilities:jt-utilities:jar:1.0.8:compile
   |  |  |  \- it.geosolutions.jaiext.scale:jt-scale:jar:1.0.8:compile
   |  |  |     \- it.geosolutions.jaiext.translate:jt-translate:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.algebra:jt-algebra:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.bandmerge:jt-bandmerge:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.bandselect:jt-bandselect:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.bandcombine:jt-bandcombine:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.border:jt-border:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.buffer:jt-buffer:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.crop:jt-crop:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.lookup:jt-lookup:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.mosaic:jt-mosaic:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.nullop:jt-nullop:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.rescale:jt-rescale:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.stats:jt-stats:jar:1.0.8:compile
   |  |  |  \- com.google.guava:guava:jar:17.0:compile
   |  |  +- it.geosolutions.jaiext.warp:jt-warp:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.zonal:jt-zonal:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.binarize:jt-binarize:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.format:jt-format:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.colorconvert:jt-colorconvert:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.errordiffusion:jt-errordiffusion:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.orderdither:jt-orderdither:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.colorindexer:jt-colorindexer:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.imagefunction:jt-imagefunction:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.piecewise:jt-piecewise:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.classifier:jt-classifier:jar:1.0.8:compile
   |  |  +- it.geosolutions.jaiext.rlookup:jt-rlookup:jar:1.0.8:compile
   |  |  \- it.geosolutions.jaiext.vectorbin:jt-vectorbin:jar:1.0.8:compile
   |  \- org.geotools:gt-cql:jar:14.3:compile
   \- com.miglayout:miglayout:jar:swing:3.7:compile
```

Fig. 47 - Árbol de dependencias[24] del programa Quickstart.

[24] Creado con el comando mvn dependency:tree de Maven.

Entonces, para probar a Mozo y compararlo con Maven, Ivy y Gradle, a cada biblioteca se le agregó el descriptor de módulo (module-info.java) utilizando un programa creado específicamente para crear módulos a partir de bibliotecas, el cual:

1. Lee el archivo POM.
2. Por cada dependencia, crea una línea 'requires' para el archivo module-info.java
3. Agrega la línea 'exports' para exponer un paquete y así permitir la compilación de módulos dependientes.
4. Agrega una línea con el nombre del módulo (el mismo del nombre de la biblioteca) y reemplazando el símbolo '–' por '_'.
5. Compila el descriptor y una clase maqueta (Prueba.java) dentro del paquete que exporta.
6. Empaqueta las clases en un archivo Jar con el nombre del módulo.

Vale aclarar que, durante la compilación del descriptor, el programa javac valida la presencia de los módulos dependientes, por lo tanto la secuencia se fue ejecutando desde las bibliotecas periféricas del árbol (hojas), como por ejemplo: it.geosolutions.imageio-ext, hasta llegar a los nodos raíz (org.geotools:gt-shapefile y org.geotools:gt-swing) (Fig. 47).
Una vez finalizada la creación de los 60 módulos, se incorporaron a cada uno las clases de la biblioteca original, de modo tal que permitan compilar y ejecutar el ejemplo. En la Fig. 48 se muestra el descriptor de un módulo creado con el programa ModuleBuilder:

```
module it.geosolutions.imageio_ext.imageio_ext_geocore {
        exports it.geosolutions.imageio_ext.imageio_ext_geocore;
        requires it.geosolutions.imageio_ext.imageio_ext_utilities;
        requires it.geosolutions.imageio_ext.imageio_ext_streams;
}
```

Fig. 48 - Descriptor del módulo imageio_ext_geocore.

Durante las pruebas, los 60 módulos se distribuyeron en partes iguales a los repositorios: trimatek.org/repository[25], thenewrepository.000webhostapp.com[26] y anotherrepository.000webhostapp.com[27]. Según el caso, se transfirieron la totalidad, la mitad en cada repositorio o 20 en cada uno al probar los 3. Cabe destacar que los 3 repositorios fueron creados específicamente para estas pruebas y tienen nombres "de fantasía".

9.4.1.2. Ejecución

Para ejecutar el programa Quickstart resolviendo dependencias con Mozo, una vez obtenida la clase cliente desde: http://trimatek.org/mozo/Mozo.class se inició la consola de comandos con: **java -cp . Mozo** y se solicitó buscar las dependencias de los módulos

[25] http://www.trimatek.org/repository
[26] http://thenewrepository.000webhostapp.com/
[27] http://anotherrepository.000webhostapp.com/

gt_shapefile y gt_swing con el comando: **fm org.geotools.gt_shapefile,org.geotools.gt_swing** (fm es la versión abreviada de find-modules) como se ve en la Fig. 49:

Fig. 49 - Comando para solicitar las dependencias del programa Quickstart con Mozo.

Luego de unos segundos, el cliente mostrará por consola el árbol de dependencias con las direcciones a cada módulo (Fig. 50). Vale aclarar que, en el caso de prueba que se muestra a continuación, los módulos estaban repartidos entre los 3 repositorios, por lo tanto, todas las ubicaciones provienen de esas fuentes. En un caso real, esas rutas (paths) pueden variar de acuerdo a dónde el intermediario localice los módulos entre su listado de repositorios activos.

```
mozo> fm org.geotools.gt_shapefile,org.geotools.gt_swing
GET request: http://trimatek.org:8080/mozo/find?modules=org.geotools.gt_shapefile,org.geotools.gt_swing
Response code: 200
{
 "from": "user-request: /mozo/find?modules=org.geotools.gt_shapefile,org.geotools.gt_swing",
 "requires": [
  {
   "module": "org.geotools.gt_shapefile",
   "path": "https://thenewrepository.000webhostapp.com/org.geotools.gt_shapefile.jar",
   "requires": [
    {
     "module": "org.geotools.gt_data",
     "path": "https://thenewrepository.000webhostapp.com/org.geotools.gt_data.jar",
     "requires": [
      {
       "module": "org.geotools.gt_main",
       "path": "https://thenewrepository.000webhostapp.com/org.geotools.gt_main.jar",
       "requires": [
        {
         "module": "org.geotools.gt_api",
         "path": "https://thenewrepository.000webhostapp.com/org.geotools.gt_api.jar",
         "requires": [
          {
           "module": "com.vividsolutions.jts",
           "path": "http://www.trimatek.org/repository/com.vividsolutions.jts.jar"
          },
          {
           "module": "org.geotools.gt_referencing",
           "path": "https://thenewrepository.000webhostapp.com/org.geotools.gt_referencing.jar",
           "requires": [
            {
             "module": "com.googlecode.efficient_java_matrix_library.core",
             "path":
"http://www.trimatek.org/repository/com.googlecode.efficient_java_matrix_library.core.jar"
            },
            {
             "module": "commons_pool.commons_pool",
             "path": "http://www.trimatek.org/repository/commons_pool.commons_pool.jar"
            },
                                          .... [recorte ] ....
```

```
      {
       "module": "org.jdom.jdom",
       "path": "https://thenewrepository.000webhostapp.com/org.jdom.jdom.jar",
       "requires": [
        {
         "module": "jaxen.jaxen",
         "path": "https://thenewrepository.000webhostapp.com/jaxen.jaxen.jar"
        },
        {
         "module": "xerces.xercesImpl",
         "path": "https://thenewrepository.000webhostapp.com/xerces.xercesImpl.jar"
        }
       ]
      }
     ]
    },
    {
     "module": "org.geotools.gt_swing",
     "path": "https://thenewrepository.000webhostapp.com/org.geotools.gt_swing.jar",
     "requires": [
      {
       "module": "org.geotools.gt_referencing",
       "path": "https://thenewrepository.000webhostapp.com/org.geotools.gt_referencing.jar",
       "requires": [
        {
         "module": "com.googlecode.efficient_java_matrix_library.core",
         "path": "http://www.trimatek.org/repository/com.googlecode.efficient_java_matrix_library.core.jar"
        },
                                                    .... [recorte ] ....
            {
             "module": "it.geosolutions.jaiext.piecewise.jt_piecewise",
             "path":
"https://anotherrepository.000webhostapp.com/it.geosolutions.jaiext.piecewise.jt_piecewise.jar",
              "requires": [
               {
                "module": "it.geosolutions.jaiext.utilities.jt_utilities",
                "path":
"https://anotherrepository.000webhostapp.com/it.geosolutions.jaiext.utilities.jt_utilities.jar",
                 "requires": [
                  {
                   "module": "it.geosolutions.jaiext.iterators.jt_iterators",
                   "path":
"https://anotherrepository.000webhostapp.com/it.geosolutions.jaiext.iterators.jt_iterators.jar"
                  }
                 ]
               }
              ]
             }
            ]
           },
           {
            "module": "it.geosolutions.jaiext.rlookup.jt_rlookup",
            "path":
"https://anotherrepository.000webhostapp.com/it.geosolutions.jaiext.rlookup.jt_rlookup.jar",
             "requires": [
              {
               "module": "it.geosolutions.jaiext.utilities.jt_utilities",
               "path":
"https://anotherrepository.000webhostapp.com/it.geosolutions.jaiext.utilities.jt_utilities.jar",
                "requires": [
                 {
                  "module": "it.geosolutions.jaiext.iterators.jt_iterators",
```

```
                "path":
"https://anotherrepository.000webhostapp.com/it.geosolutions.jaiext.iterators.jt_iterators.jar"
              }
            ]
          }
        ]
      },
                                          .... [recorte ] ....
    {
      "module": "com.miglayout.miglayout",
      "path": "http://www.trimatek.org/repository/com.miglayout.miglayout.jar"
    }
  ]
 }
]
}
Result stored in: res0
Elapsed time: 3.217 seconds
```


Fig. 50 – Resumen del árbol de dependencias para los módulos gt-shapefile y gt-swing.

Finalmente, para descargar los módulos a una ubicación local, se ejecuta el comando **dm res0** (versión corta de download-modules res0) y se iniciará la copia desde las ubicaciones definidas en el árbol guardado en la variable res0 (Fig. 51).

Fig. 51 - Descarga de módulos con cliente Mozo.

Una vez disponibles todos los módulos que son parte de la clausura para ejecutar Quickstart, se configura el Build Path hacia la ubicación donde fueron descargadas las dependencias (en este caso, el directorio D:\Temp) y luego de compilar, se ejecuta el programa obteniendo el resultado de la Fig. 52:

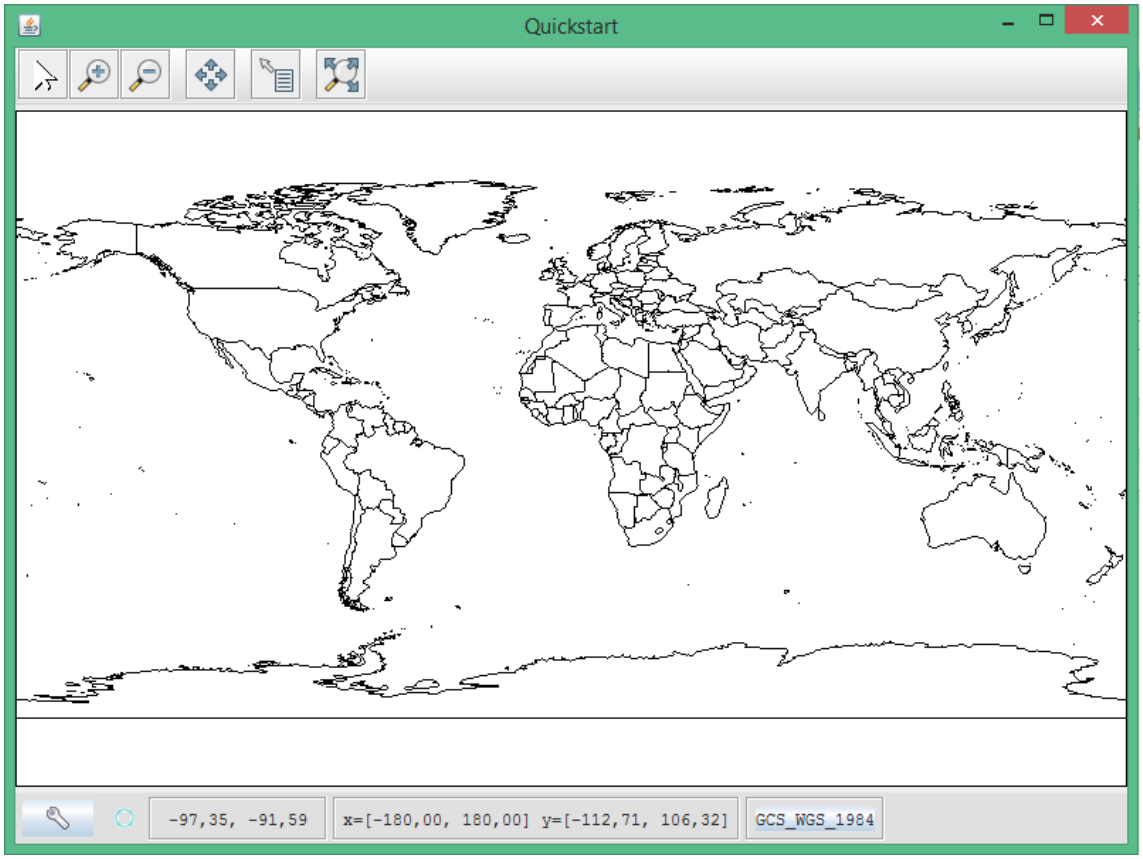


Fig. 52 - Ejecución de Quickstart compilado a partir de dependencias obtenidas con Mozo.

Con el programa Quickstart funcionando, se valida que la propuesta puede resolver la clausura de dependencias, evitando la necesidad de agregar metadatos ni asociaciones estáticas entre el descriptor del programa de usuario, los repositorios o entre los descriptores de las bibliotecas y los repositorios. Además, como se verá a continuación, esta propuesta consigue buenos resultados de desempeño en comparación con otros gestores de dependencias.

9.5. Prueba comparativa

En una prueba comparativa con Maven, Gradle e Ivy, los resultados también son alentadores. Mientras que para resolver y obtener las dependencias del programa Quickstart, Maven requiere de casi 8 minutos, con Mozo la misma operación se ejecuta en menos de 2 minutos y puede reducirse hasta casi 30 segundos cuando los descriptores involucrados están disponibles en memoria. Como se mostrará a continuación, Mozo obtiene todas las dependencias de Quickstart en menos del 30 % del tiempo que necesita Maven la primera vez y hasta menos del 10 % la segunda vez. Para el caso comparativo con Gradle, hay paridad cuando Mozo debe localizar y transferir los descriptores (sin caché), mientras que las siguientes la reducción es de casi el 70 % del tiempo frente a Gradle. Ivy obtiene marcas de desempeño inferiores a Maven.

9.5.1.1. Mediciones

Se configuraron 3 gestores de dependencia Java para obtener las dependencias de Quickstart. Se repitió 3 veces la solicitud sin caché local. Si bien las 3 herramientas comparten los mismos repositorios, con Maven no fue necesario configurar a la fuente Geo Solutions. En la Tabla 9 se muestran los resultados.

Tabla 9 - Tiempo de resolución y descarga de dependencias con Maven, Ivy y Gradle.

Caso	Gestor de dependencias	Comando de ejecución	Observaciones	Repositorios				Tiempo total (segundos)
				TC	JN	OS	GS	
M1	Maven	mvn clean install	Sin caché local.	1	1	1		453
M2	Maven	mvn clean install	Sin caché local.	1	1	1		451
M3	Maven	mvn clean install	Sin caché local.	1	1	1		407
M4*	Maven	Proyecto > Maven > Update Project	Desde plug-in M2Eclipse y sin caché local.	1	1	1		901
G1	Gradle	gradlew build -x test --refresh-dependencies	Sin ejecución de pruebas ni caché local.	1	1	1	1	85
G2	Gradle	gradlew build -x test --refresh-dependencies	Sin ejecución de pruebas ni caché local.	1	1	1	1	87
G3	Gradle	gradlew build -x test --refresh-dependencies	Sin ejecución de pruebas.	1	1	1	1	94
I1	Ivy	ant	Sin caché local.	1	1	1	1	716
I2	Ivy	ant	Sin caché local.	1	1	1	1	694
I3	Ivy	ant	Sin caché local.	1	1	1	1	707

(*) M4: Medición descartada para este estudio.

Referencias:

TC: The Central Repository

JN: Java.net Repository

OS: OSGeo

GS: Geo Solutions

En los casos G1, G2 y G3 llamativamente, el orden en que son declararos los repositorios en el archivo build.gradle afecta el proceso de búsqueda de dependencias. Por ejemplo, si primero se declara a mavenCentral() y luego los específicos para GeoTools en el bloque repositories{}, la ejecución de Gradle falla. Ubicando a mavenCentral() al final del bloque repositories {} la ejecución es normal (Fig. 57).

En la

Tabla 10 se muestran los resultados de resolver y obtener las dependencias de Quickstart con Mozo. La columna Repositorios indica las fuentes utilizadas en cada caso.

Tabla 10 - Medición de tiempo de resolución y descarga de dependencias con Mozo.

Caso	Gestor de dependencias	Comando de ejecución	Observaciones	Repositorios			Tiempo total (segundos)
				T	N	A	
Z1	Mozo	fm (*) + dm res0		1			91
Z1.1	Mozo	fm (*) + dm res1	Descriptores en caché.	1			37
Z2	Mozo	fm (*) + dm res0		1			96
Z2.1	Mozo	fm (*) + dm res1	Descriptores en caché.	1			32
Z3	Mozo	fm (*) + dm res0		1			93
Z3.1	Mozo	fm (*) + dm res1	Descriptores en caché.	1			38
Z4	Mozo	fm (*) + dm res0			1		99
Z4.1	Mozo	fm (*) + dm res1	Descriptores en caché.		1		34
Z5	Mozo	fm (*) + dm res0		1	1		96
Z5.1	Mozo	fm (*) + dm res1	Descriptores en caché.	1	1		38
Z6	Mozo	fm (*) + dm res1		1	1	1	96
Z6.1	Mozo	fm (*) + dm res1	Descriptores en caché.	1	1	1	35
Z7	Mozo	fm (*) + dm res1		1	1	1	97
Z7.1	Mozo	fm (*) + dm res1	Descriptores en caché.	1	1	1	36
Z8	Mozo	fm (*) + dm res1		1	1	1	99
Z8.1	Mozo	fm (*) + dm res1	Descriptores en caché.	1	1	1	33

(*) org.geotools.gt_shapefile,org.geotools.gt_swing

Referencias

T: Repositorio Trimatek

N: The New Repository

O: Another Repository

Observaciones:

Caché: Para Maven, Gradle e Ivy caché significa una copia local de la biblioteca, en cambio para mozo significa una copia local (en el servidor) del descriptor de módulo (module-info) y su ruta a un repositorio.

Comentario: La marcada diferencia de tiempo con Maven e Ivy es que éstos primero descargan todos los archivos POM a una ubicación local y recién después descargan las bibliotecas secuencialmente. En cambio, Gradle paraleliza estas tareas haciendo varias

descargas de archivos POM y Jar en simultáneo. Mozo también optimiza las descargas de archivos Jar mediante la ejecución de hasta 5 hilos de ejecución (threads) en simultáneo. Para todos los casos y una vez obtenidas las dependencias, se comprobó que Quickstart se compilara y ejecutara con normalidad.

9.5.1.2. Visualización de los resultados

En el gráfico de la Fig. 53 se compara visualmente el desempeño de los 4 gestores de dependencias relevados en Tabla 9 y
Tabla 10. Se agrupan por tiempo mínimo (mejor marca), media (promedio de las mediciones) y máximo (peor marca):

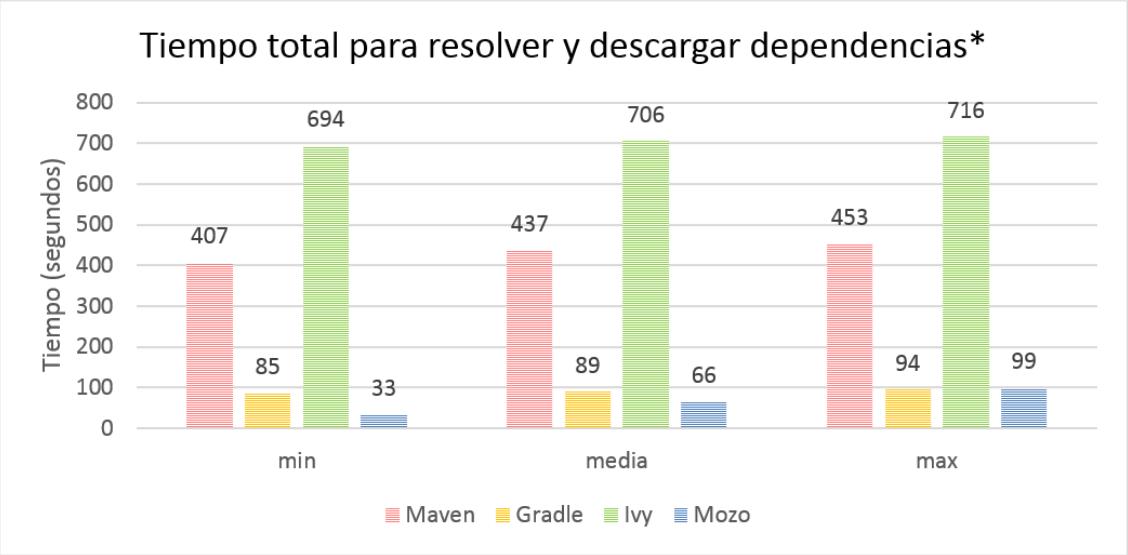


Fig. 53 – Tiempo de respuesta para resolver y descargar las dependencias de Quickstart.
(*) Mozo configurado con 3 repositorios y tomando como fuente mediciones con y sin caché (Casos: Z6, Z6.1, Z7, Z7.1, Z8, Z8.1).

Entre las mejores mediciones (min) se destaca Mozo, que demandó sólo 33 segundos para retornar el árbol de dependencias y los 60 archivos Jar involucrados. En promedio (media), también Mozo supera al tiempo de respuesta de Gradle. En el grupo de las peores marcas (max), Gradle consigue el mejor resultado. También es notable que tanto Maven como Ivy obtienen resultados de muy bajo desempeño general.

Comentario: Con el propósito de establecer condiciones equivalentes, para crear el gráfico de la Fig. 53 sólo se utilizaron las mediciones de Mozo configurado con 3 repositorios.

9.5.1.3. Escalabilidad

Como se ve en la columna “repositorios” de la
Tabla 10, los casos Z1 a Z5 fueron ejecutados con Mozo configurado con 1 ó 2 repositorios. Para las mediciones de los casos Z6 a Z8, los módulos estuvieron repartidos en

proporciones equivalentes en 3 repositorios diferentes. Sin embargo, como se muestra en la Fig. 54, el impacto en el desempeño no es significativo. De la mejor marca: 91 segundos con 1 repositorio, a la peor: 99 segundos a 3 repositorios, hay una diferencia de 8 segundos en el tiempo total de resolución y entrega de módulos.

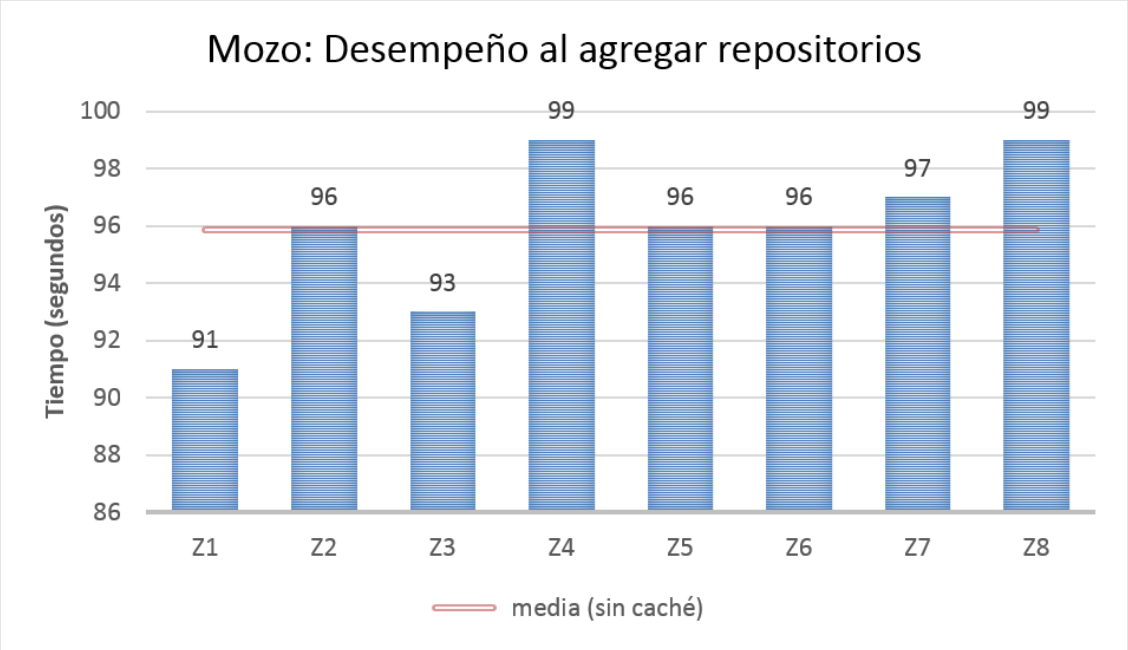


Fig. 54 - Comparación de desempeño de Mozo con 1, 2 y 3 repositorios.

Operativamente, la solución primero busca en el primer repositorio el módulo requerido y de no encontrarlo, busca en el segundo y luego en el tercero. Esta localización dinámica no impacta significativamente en el desempeño, por lo tanto, es posible escalar la cantidad de repositorios de módulos sin que esto signifique resignar desempeño.

9.5.1.4. Descriptores en caché

Para optimizar el tiempo de respuesta, el intermediario guarda en una memoria temporal los descriptores localizados junto con su dirección de origen, por lo tanto, sucesivas solicitudes del mismo módulo no implican localizarlo entre repositorios. De este modo y como se muestra en la Fig. 55, el tiempo total de respuesta para obtener las dependencias de Quickstart es de entre 35 y 36 segundos en promedio con Mozo.

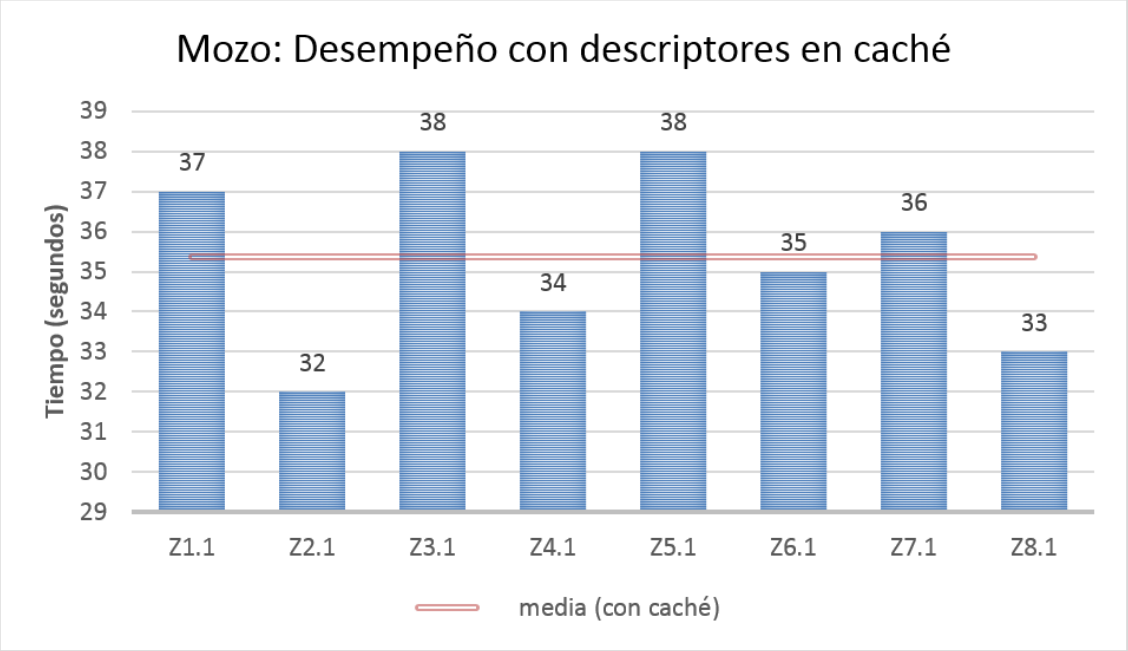


Fig. 55 - Tiempo de respuesta de Mozo a solicitudes en caché.

En condiciones de trabajo productivas, lo habitual es que se den frecuentes solicitudes de un mismo conjunto de bibliotecas, es decir, las más populares. Por lo tanto, contar con un caché de los descriptores junto con su ubicación, reduce significativamente el tiempo de respuesta, ya que el intermediario sólo debería "salir a buscar" entre los repositorios, los módulos la primera vez que son requeridos (o al refrescar esta información) y luego utilizaría el dato disponible en memoria local.

9.5.1.5. Ambiente de mediciones

Todos los casos se ejecutaron desde el mismo equipo (Intel i7 con 8GB de RAM, disco magnético, Windows 8.1 y JDK 1.8). Para Maven, el comando utilizado fue **mvn clean install** desde la interfaz de comandos y con Maven desde Eclipse, desde el menú contextual sobre el nodo raíz del **proyecto > Maven > Update Project** y en ambos casos con el repositorio local vacío. Para resolver las dependencias con Gradle, el comando utilizado fue: **gradlew build -x test --refresh-dependencies**. Ivy es parte de Ant y se ejecuta con el comando **build**. Mozo busca y crea el árbol de dependencias con **fm org.geotools.gt_shapefile,org.geotools.gt_swing** y luego **dm [nombre de la variable con resultado de fm]** para descargar los módulos.

Velocidad de acceso a Internet: 8.6 Mbps de promedio (medido con fast.com).

9.5.1.6. Versiones y configuración

Maven versión 3.5.0

Descriptor del proyecto (Mediciones M1, M2, M3):

pom.xml

```
<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>org.geotools</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. 56 - Configuración de proyecto para Maven.

Gradle versión 4.0.1

Descriptor del proyecto (Mediciones G1, G2, G3):

build.gradle

```
apply plugin: 'java'
apply plugin: 'application'
repositories {
        maven {
                url "http://download.java.net/maven/2"
  }
        maven {
                url "http://download.osgeo.org/webdav/geotools/"
  }
        maven {
                url "http://maven.geo-solutions.it"
  }
        mavenCentral()
}
dependencies {
  compile group: 'org.geotools', name: 'gt-shapefile', version: '14.3'
        compile group: 'org.geotools', name: 'gt-swing', version: '14.3'
}
mainClassName = 'Quickstart'
```

Fig. 57 - Configuración de proyecto para Gradle.

Ivy versión 2.4.0

Descriptores del proyecto (Mediciones I1, I2, I3):

build.xml

```
<project name="quickstart" default="resolve" xmlns:ivy="antlib:org.apache.ivy.ant">
  <target name="resolve" description="resolve dependencies for quickstart">
    <ivy:resolve />
  </target>
</project>
```

Fig. 58 – Configuración de proyecto Ant con Ivy.

ivy.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ivy-module version="2.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/schemas/ivy.xsd">
   <info organisation="org.geotools" module="quickstart"/>
   <dependencies>
     <dependency org="org.geotools" name="gt-shapefile" rev="14.3"/>
     <dependency org="org.geotools" name="gt-swing" rev="14.3"/>
   </dependencies>
</ivy-module>
```

Fig. 59 - Configuración de proyecto para Ivy.

ivysettings.xml

```
<ivysettings>
   <settings defaultResolver="chain"/>
   <resolvers>
      <chain name="chain">
         <ibiblio name="maven2-repository.dev.java.net" m2compatible="true"
root="http://download.java.net/maven/2"/>
         <ibiblio name="osgeo" m2compatible="true"
root="http://download.osgeo.org/webdav/geotools/"/>
         <ibiblio name="geo-solutions" m2compatible="true" root="http://maven.geo-
solutions.it"/>
         <ibiblio name="maven2" m2compatible="true"/>
      </chain>
   </resolvers>
</ivysettings>
```

Fig. 60 - Configuración de repositorios para Ivy.

Mozo versión 0.3

Descriptor del proyecto (Mediciones Z1 a Z8):

- No necesario -

Ubicación de los repositorios:

trimatek.org/repository: Asheville, North Carolina, USA.

thenewrepository.000webhostapp.com: Ashburn, Virginia, USA.

anotherrepository.000webhostapp.com: Greenville, South Carolina, USA.

Cabe destacar que tanto Maven, Gradle como Ivy dependen de los metadatos de los archivos POM, build.gradle y build.xml+ivy.xml+ivysettings.xml para el nivel 0 y a su vez dependen de los metadatos leídos de los sucesivos archivos POM de las dependencias transitivas. Todos ellos estáticos.

Para todas las pruebas se utilizaron las últimas versiones estables. Para Ivy la útlima versión 2.4.0 es de diciembre de 2014, por lo que se estima que el proyecto próximamente sería desactivado.

9.5.2. Usabilidad

El modo de interactuar es, en esta versión inicial, únicamente a través de un cliente de línea de comandos (CLI). Con una serie acotada de comandos, hasta un programador principiante podría solicitar obtener el árbol de clausura y luego solicitar los módulos en él.

Toda la complejidad relacionada con la resolución de dependencias queda encapsulada en el servidor, es decir, tanto el análisis de requerimientos como la exploración de repositorios en busca de módulos, es transparente para el usuario.

El cliente también incluye un manual en línea que al recibir las palabras: man, help o ayuda, muestra el listado de comandos con un ejemplo de cada uno. (Fig. 61).

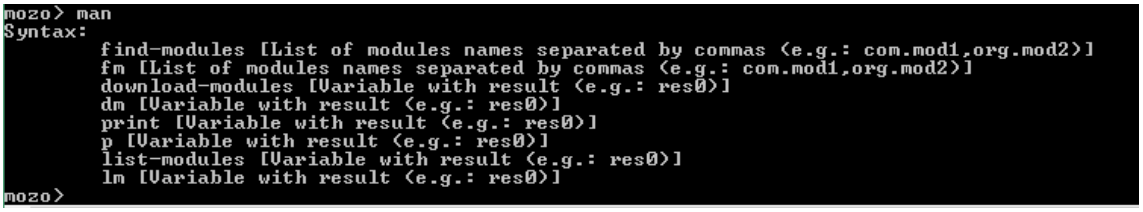


Fig. 61 - Ayuda en línea de la interfaz con el usuario[28].

De este modo, el programador orienta su atención y dedicación en crear software y no a configurar herramientas de soporte a la gestión de dependencias.

9.5.3. Mantenibilidad

Es una solución basada en arquitectura orientada a servicios (SOA) que concentra todas las funciones de servidor y las principales del cliente. Centralizando así la gestión de ajustes y ampliaciones de manera transparente para el usuario.

9.5.4. Escalabilidad

Como se demostró en la Sección 9.5.1.3, agregar repositorios no impacta significativamente en el desempeño de la resolución. Por otro lado, si el servicio requiriese atender a un mayor volumen de solicitudes, la primera alternativa es la de escalar verticalmente (incrementar velocidad de CPU, memoria RAM, ancho de banda). Un esquema de balanceador de carga, sería la alternativa más apropiada para un escalamiento horizontal.

9.5.5. Simplicidad

Es una solución libre de meta-datos. Con esta propuesta, para que un módulo pueda ser parte de un repositorio y el gestor de dependencias localice sus dependencias, no es necesario agregar archivos descriptores como sí ocurre con Maven, Gradle o Ivy. Por otro lado, el modo que se representan y transportan los datos, es con protocolos de integración libres como HTTP, REST y JSON.

9.6. Atributos comparados

A modo de resumen, se presenta en la Tabla 11 una comparación de las 4 herramientas para la gestión/resolución de dependencias, donde se establece una valoración numérica para cada atributo evaluado. Dando como resultado el siguiente total por tecnología:

[28] Originalmente el cliente estaba en castellano, luego se tradujo al inglés para una demostración en un congreso internacional.

Mozo: 32 puntos
Maven: 17 puntos
Gradle: 24 puntos
Ivy: 16 puntos

A continuación, un breve comentario para cada resultado.

Desempeño:
Los resultados de Mozo y Gradle son equivalentes.

Acoplamiento y mantenibilidad:
La ausencia de direcciones a repositorios en los descriptores junto con la propuesta de una arquitectura orientada a servicios y la carga remota de clases del cliente de Mozo marcan una considerable diferencia para esta tecnología, al igual que para mantenibilidad.

Usabilidad:
También Mozo se destaca dada su simplicidad y casi nula instalación.

Eficiencia:
Gradle obtiene el mejor puntaje dado que gestiona un repositorio local y los tiempos de descarga son siempre muy buenos.

Rendimiento:
Prácticamente no hay diferencia con ninguna de las tecnologías probadas, en todos los casos el uso de recursos es moderado a bajo.

Simplicidad:
Aquí también Mozo obtiene el mayor puntaje dado que tanto la complejidad de la herramienta como de parametrización de una solicitud de resolución de dependencias queda encapsulado en el intermediario, siendo esto, siempre transparente para el usuario.

A continuación una representación gráfica de estas dimensiones comparadas.

9.6.1. Visualización

Los atributos cuantificados en la sección anterior son representados y comparados en el siguiente diagrama radial:

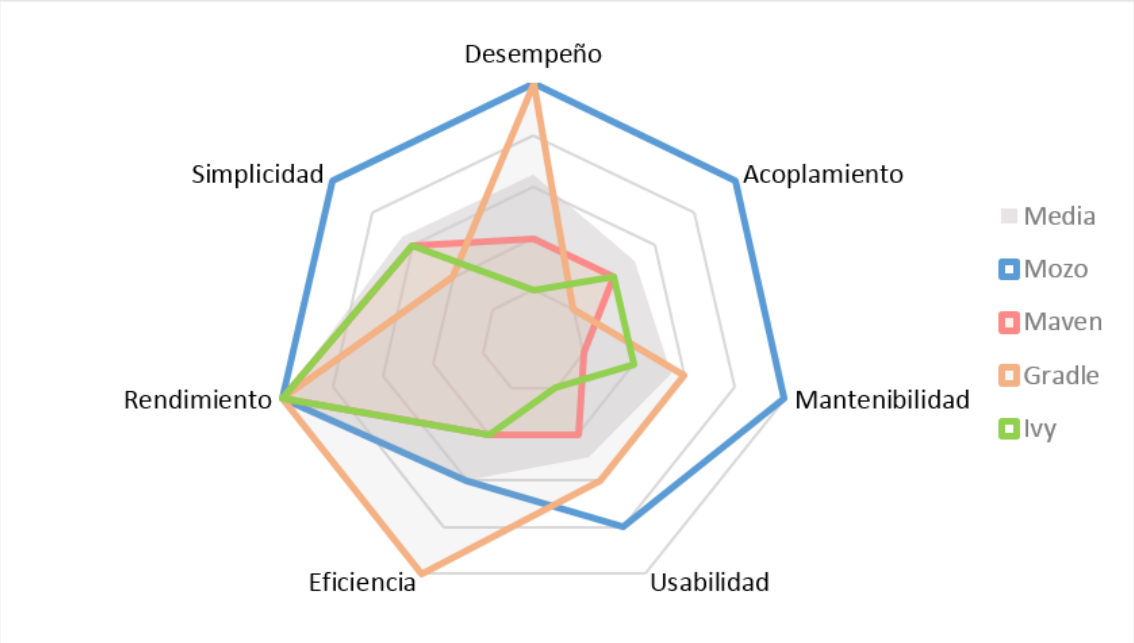


Fig. 62 - Atributos comparados

Donde se observa que la propuesta de esta tesis supera y es más uniforme en comparación con las otras herramientas, excepto para la dimensión eficiencia, donde Gradle se destaca por contar con un repositorio local en el ambiente de desarrollo.

Tabla 11 – Resumen comparativo de atributos

	Mozo	Maven	Gradle	Ivy
Desempeño	Los resultados de la sección 9.5 lo ubican como la tecnología con las mejores marcas de desempeño. **Puntaje: 5**	Quedó en el 3er puesto de la comparativa, detrás de Mozo y Gradle. **Puntaje: 2**	Junto con Mozo, esta tecnología obtuvo las mejores marcas de desempeño. **Puntaje: 5**	En la comparativa de la sección 9.5 obtuvo los peores resultados de tiempo. **Puntaje: 1**
Acoplamiento	No existe ninguna relación estática entre el proyecto y los repositorios. **Puntaje: 5**	Requiere indicar las rutas a los repositorios en el descriptor del proyecto. **Puntaje: 2**	Requiere indicar las rutas a los repositorios e influye el orden en que son listadas. **Puntaje: 1**	Requiere indicar las rutas a los repositorios en el descriptor del proyecto. **Puntaje: 2**
Mantenibilidad	El cliente carga remotamente y a demanda las clases con la funcionalidad requerida. **Puntaje: 5**	El cliente se instala en el ambiente de ejecución, sin actualizaciones automáticas. **Puntaje: 1**	El cliente se actualiza parcialmente durante la primera ejecución y las sucesivas. **Puntaje: 3**	El cliente es un Jar que se acopla a Ant. No contempla actualización dinámica. **Puntaje: 2**
Usabilidad	Si bien es acotado el listado de comando de la CLI, no es un ambiente totalmente intuitivo. **Puntaje: 4**	En los casos donde se deben indicar las rutas a repositorios se incrementa la complejidad. **Puntaje: 2**	Desde el DSL es más fácil que Maven o Ivy configurar, no obstante, requiere su aprendizaje. **Puntaje: 3**	Requiere configurar hasta 3 archivos descriptores al indicar rutas a repositorios. **Puntaje: 1**
Eficiencia	Remote Zip y el caché de descriptores, lo optimizan. No ofrece repositorio local. **Puntaje: 3**	La presencia del repositorio local lo optimiza. No se destaca en el proceso de descarga. **Puntaje: 2**	El repositorio local y el desempeño al descargar descriptores y bibliotecas lo destacan. **Puntaje: 5**	La presencia del repositorio local lo optimiza. No se destaca en el proceso de descarga. **Puntaje: 2**
Rendimiento	En ningún momento se detectó un significativo consumo de recursos. **Puntaje: 5**	En ningún momento se detectó un significativo consumo de recursos. **Puntaje: 5**	En ningún momento se detectó un significativo consumo de recursos. **Puntaje: 5**	En ningún momento se detectó un significativo consumo de recursos. **Puntaje: 5**
Simplicidad	Tanto su instalación como uso son muy sencillos. Una futura GUI la incrementaría. **Puntaje: 5**	Su instalación requiere configurar variables locales y el uso es a través de una CLI. **Puntaje: 3**	Su instalación requiere configurar variables locales y conocimiento previo. **Puntaje: 2**	Una vez instalado y funcionando Ant es fácil de instalar y usar. **Puntaje: 3**

9.7. Conclusiones del Capítulo

En este Capítulo se diseña un caso de prueba con el propósito de evaluar la propuesta en una situación real.

Este caso de prueba, al igual que en el Capítulo 3, también consistió en resolver las 60 dependencias (2 directas y 58 transitivas) del programa Quickstart que utiliza el software Geotools 14.3. Como al momento de las pruebas todavía no existían módulos Java 9 de esas dependencias, se crearon los 60 módulos a partir de las bibliotecas originales, agregando un archivo descriptor a cada una. El caso de prueba también abarcó una comparación con las 3 herramientas de software más utilizadas para resolver dependencias Java: Maven, Gradle e Ivy. Para ello se crearon ambientes de equivalentes condiciones, tanto en el objetivo (obtener las 60 dependencias de Quickstart) como en la configuración de los casos (infraestructura y cantidad de repositorios).

Durante las pruebas se realizaron al menos 3 ensayos con cada herramienta y se registraron todas las mediciones de los resultados. Para el caso de Mozo (es el nombre del prototipo presentado en esta Tesis), se ejecutaron un total de 16 mediciones, combinando diferente cantidad de repositorios donde buscar los módulos. Una vez obtenidos los resultados, se procesaron los datos y representaron en gráficos comparativos, agrupados por: valores mínimos, media y máximos (peor resultado). Mozo obtuvo las mejores marcas en los casos del mínimo y la media. Gradle consiguió la mejor marca para el caso del máximo con una diferencia a favor de 5 segundos frente a Mozo.

Para cuantificar la escalabilidad de la solución, se validó comparando el impacto en el tiempo de respuesta que produce agregar repositorios, es decir, resolviendo dependencias ubicadas en 1 repositorio, 2 y 3 repositorios distintos y remotos. Esto no produjo variaciones significativas en los resultados, siendo de 91 segundos cuando Mozo sólo debe acceder a 1 repositorio y de 99 en el peor caso cuando los módulos están repartidos entre 3 repositorios.

Por último, y también para cuantificar el desempeño en un caso habitual, se midió y comparó la variación del tiempo de respuesta contando con todos los descriptores en memoria caché, una situación que no debería ser de excepción en condiciones de producción ya que el lote de bibliotecas más frecuentemente referenciada por la mayoría del software industrial habitualmente es pequeño. En este caso, para resolver y descargar las 60 bibliotecas/módulos con Mozo, sólo tomó 35.5 segundos de promedio.

Se completa el Capítulo describiendo la configuración de las 4 herramientas comparadas y destacando otros atributos particulares de la propuesta: Usabilidad por lo fácil que es descargar, ejecutar y operar el cliente de interfaz de comandos. Mantenibilidad, por ser una solución basada en una arquitectura orientada a servicios que concentra tanto el servidor como la mayor parte del cliente en la nube, haciendo transparente para el usuario todo cambio preventivo o correctivo. Simplicidad también es un atributo destacado, dado que es una solución libre de meta-datos propietarios, es decir, no requiere agregar ningún recurso externo a los módulos ni a los repositorios.

Finalmente se establece una valoración numérica para cada atributo evaluado, tanto para Mozo como las otras 3 herramientas. Otorgando un puntaje individual para cada una desde esas perspectivas y explicando la justificación de cada valoración. En esta comparativa las 2 mejores marcas fueron para Mozo y Gradle con 32 y 24 puntos respectivamente.

Capítulo 10

Conclusiones y trabajos futuros

El desarrollo de software a escala industrial requiere de infraestructura acorde a los requerimientos de cada proyecto. La comunidad de software se retroalimenta de forma permanente, a través de la reutilización de componentes distribuidos en el formato de bibliotecas o paquetes. Actualmente, los proyectos de software tienden a ser diseñados como una composición de recursos de funcionalidad específica, promoviendo la reutilización y siendo, en muchos casos, un factor clave de éxito, ya sea por calidad probada o integración inmediata de una nueva prestación.

Desde principios de los años ’90 han surgido diferentes herramientas de soporte a la integración y actualización de sistemas operativos a través de paquetes. Esas herramientas denominadas Gestores de Paquetes permiten agregar y quitar, de forma atómica, paquetes de software proveniente de repositorios externos. Estas unidades autocontenidas de software, en la mayoría de los casos, poseen dependencias. Las dependencias son otras piezas de software que deben formar parte de la plataforma destino y son referenciadas por el software a agregar. Esas dependencias también están organizadas en forma de paquetes y, a su vez, pueden requerir la presencia, o entrar en conflicto con otros paquetes. Esta sucesión de dependencias en muchos casos deriva en complejos grafos que deben evitar anomalías, como por ejemplo ciclos o repeticiones de paquetes.

Cada paquete posee un manifiesto de metadatos que definen su nombre, versión, dependencias y conflictos. Habitualmente, una dependencia determina un rango de compatibilidad con versiones, por lo tanto, un Gestor de Paquetes puede contar con diferentes estrategias de resolución de dependencias y estar vinculado a más de un repositorio de paquetes.

La industria del software también adoptó el modelo de distribución por paquetes, definiendo a la biblioteca como un conjunto de elementos de software reutilizables, indivisibles y de alta cohesión. Estas bibliotecas también están referenciadas a otras a través de un manifiesto y, habitualmente, están disponibles en repositorios públicos. Para el caso de la tecnología Java, las bibliotecas son archivos comprimidos en formato Zip, pero con extensión Jar, donde el manifiesto no es obligatorio, delegando en soluciones de terceros la declaración formal de sus dependencias.

Tomando como base a los Gestores de Paquetes, la industria de software desarrolló Gestores de Dependencias. Estas herramientas interactúan con los repositorios de bibliotecas, asistiendo a los ambientes de desarrollo de software en los procesos de recuperación y clausura de dependencias.

Si bien la adaptación del modelo de Gestor de Paquetes al entorno de desarrollo ha sido exitosa, padece de los siguientes inconvenientes:

Subutilización de dependencias (SD): Posiblemente el modelo de vuelco total de bibliotecas a un entorno local haya sido apropiado para los años '90 y principios de los 2000, cuando el acceso a Internet era limitado. No obstante, requerir de tantas piezas de software inutilizado, limita la portabilidad a ambientes de producción de recursos limitados como dispositivos IoT o cómputo en la nube.

Alto acoplamiento (AA): El modelo de resolución de dependencias más difundido de la actualidad establece que sea el cliente, a través del descriptor de proyecto, quien localice las bibliotecas en las fuentes declaradas, delegando en el ambiente de programación contar con las direcciones a fuentes de bibliotecas actualizadas. Estudios revelan que casi 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.

Desempeño (DE): En casos donde los recursos son limitados, por ejemplo: dispositivos IoT o servidores virtuales, es costoso en términos de ciclos de procesador enlazar clases en tiempo de ejecución. Esto se incrementa si las bibliotecas contienen un gran número de clases no referenciadas.

Conflictos entre bibliotecas (CB): Es habitual que existan problemas durante la resolución de dependencias. El sistema de gestión de dependencias debe contar con suficiente capacidad analítica para evitar estos conflictos. Además, es imprescindible que los metadatos estén completos, incluyendo casos especiales de incompatibilidad entre bibliotecas.

Este trabajo de tesis se dividió en tres fases. La primera fase (antecedentes) se orientó a cuantificar la subutilización de bibliotecas. La segunda, se enfocó en diseñar y crear un prototipo donde, partiendo de las cualidades de enlace dinámico entre clases de la tecnología Java, fuese posible personalizar el suministro de recursos de bibliotecas a partir de una resolución de dependencias a nivel de clases (propuesta inicial). Y finalmente una tercera fase, donde se propuso un nuevo prototipo para la resolución de dependencias a nivel de módulos (propuesta actualizada).

En la primera fase y previamente a crear una herramienta para medir la tasa de referencias entre bibliotecas, se evaluaron otras de terceros, llegando a la conclusión que no cubrían los requisitos del estudio. Luego de validar su precisión a través de un estudio comparativo, se midió una muestra de productos de software desarrollados con tecnología Java y sus dependencias, llegando a verificar esa subutilización con un resultado promedio inferior al 10% de recursos referenciados sobre el total disponible.

La segunda fase tuvo como objetivo estudiar el código intermedio generado por el compilador Java y desarrollar un prototipo basado en tecnología OSGi, que primero prepare subrogantes de bibliotecas para conseguir la compilación del programa de usuario, para luego suministrar sólo las clases que formaran parte de la clausura. Mediante este prototipo (inicial) y a partir de pruebas comparativas con gestores de dependencias a nivel biblioteca, se llegó a la conclusión preliminar que el empaquetado en bibliotecas no ofrecería el nivel de granularidad adecuado para construir el software Java que la industria estaría necesitando.

En simultáneo con el desarrollo de este proyecto (2015-2016), la Java Community Process propuso y aprobó la especificación JSR-376 con el objetivo de modularizar tanto la plataforma (JDK), como el software creado a partir de allí. Este grupo de expertos también acordó incluir la implementación de esta especificación en la próxima versión de Java.

La última fase de este trabajo consistió en estudiar el alcance de los objetivos específicos formalizados a través de las JEP (JDK Enhancement Proposal) que definen con mayor nivel de detalle la nueva versión de Java (JDK 9) lanzada en septiembre de este año. Con esta documentación, más los borradores de las próximas especificaciones del lenguaje/máquina virtual y versiones preliminares de la plataforma (early access), se diseñó e implementó el prototipo de un servicio especializado en resolver dependencias de módulos Java. Este servicio recibe solicitudes de módulos, y retorna un árbol conformado por los módulos y las dependencias que determinan la clausura.

En este nuevo sistema de módulos para Java (proyecto Jigsaw), las bibliotecas se dividen en módulos, los módulos están interrelacionados a través de un descriptor obligatorio donde se define: nombre del módulo, módulos requeridos, paquetes que exporta y los servicios que publica y consume. En estos módulos, al igual que las bibliotecas Java, las clases compiladas están empaquetadas como archivos Zip, pero con extensión Jar. Además, el sistema de módulos también fragmenta el espacio desde donde son cargadas las clases, propone un class-path modular, que agilizaría el tiempo de respuesta para la carga de clases. Esta nueva versión también incluye un enlazador estático de módulos, de modo que sea posible crear plataformas de ejecución personalizadas y optimizadas para la arquitectura anfitriona.

10.1. Conclusión parcial

Los problemas de subutilización de dependencias (SD) y desempeño (DE), serían abordados por la próxima versión de la plataforma Java. El sistema de módulos fragmenta en una proporción de 1:20, por lo que reduciría significativamente esta subutilización de bibliotecas. JDK 9 también daría solución para los problemas de desempeño (DE) derivados de la arquitectura monolítica de Java hasta la versión 8. Con un class-path modular y la posibilidad de crear imágenes a medida, la plataforma estaría corrigiendo estos problemas. Por último, para dar solución a los problemas restantes: alto acoplamiento (AA) y conflictos entre bibliotecas (CB), este estudio se orientó en el sentido que se explica a continuación.

Tomando como base la experiencia de la primera versión del prototipo y en función de las características del sistema de módulos que ofrece Java 9, se actualizó la propuesta pasando de una resolución de dependencias a nivel clase, a una resolución a nivel módulo. Esta nueva versión también es un intermediario entre el ambiente de desarrollo y los repositorios de bibliotecas/módulos, pero con la diferencia que construye el grafo de dependencias sin hacer copias locales de los módulos. Sólo recupera la porción del archivo Jar que contiene el descriptor, y lo descompila para determinar si deberá formar parte del resultado. De esta

manera, el cliente (ambiente de desarrollo) recibe un grafo de clausura consistente y con las rutas a los módulos para conformar su plataforma de desarrollo, es decir, el conjunto de bibliotecas consolidado.

10.2. Conclusión final

Con esta propuesta se busca dar solución a los problemas de alto acoplamiento (AA) y conflictos entre bibliotecas (CB), para ello y a modo de prueba de concepto, se diseñó e implementó un prototipo con el propósito de presentar una alternativa para:

1. Desacoplar la resolución de dependencias entre un ambiente de desarrollo y los repositorios de bibliotecas/módulos.
2. Migrar la resolución de dependencias de un programa local configurado por un archivo de configuración de proyecto, a un servicio especializado.
3. Concentrar y optimizar el proceso analítico de resolución de dependencias en un servicio en la nube, que a su vez pueda ser ampliado y mejorado de forma transparente sin requerir nuevas instalaciones.
4. Que los sistemas de gestión y construcción de proyectos de software puedan delegar la resolución y localización de dependencias en un servicio específico.
5. Ofrecer una solución que no requiera agregar o duplicar metadatos, sino que sólo se base en la información nativa del descriptor de módulo.

En resumen, este trabajo tiene por objetivo actualizar el modo como se resuelven las dependencias de software Java, pasando de un modelo centrado en: metadatos, estación de trabajo y repositorios, a un servicio dedicado y específico que pueda dar soporte a las demandas actuales y futuras de la industria del software. Asimismo, este proyecto 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 en menor medida en las herramientas de soporte.

10.3. Trabajos futuros

La siguiente fase del proyecto, y a modo de trabajo futuro para esta tesis, 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.

Tomando como referencia el proyecto Spack, el siguiente paso apuntaría a crear un validador del grafo de dependencias, con el fin de garantizar la exactitud del resultado.

Otra ampliación consistiría en 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 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.

Para permitir integración con sistemas gestores de proyectos Java, se proponte también como objetivo publicar una API, de modo tal que estas herramientas puedan delegar en este servicio la resolución de dependencias.

Una característica que no está disponible en la primera versión de Java 9, es el manejo de versiones de módulos. Según la documentación actual, está previsto que sea incorporado en las próximas versiones. Por este motivo, este servicio también deberá sumar ese parámetro a la búsqueda de bibliotecas, al igual que la condición 'uses' presente en descriptor de módulo.

También será necesario analizar la técnica de autocarga (autoload) de clases, que está disponible para el gestor de dependencias Composer (PHP), el cual se basa en principios creados en los años '70 y que posiblemente, ahora con Internet, también sean de utilidad para la tecnología Java.

Una vez disponible la versión estable del servicio propuesto, el siguiente paso estaría enfocado en medir precisión y tiempos de respuesta. Con esos resultados, se prepararía un documento para su publicación en una revista o congreso específicos del área.

Si bien las ideas presentadas en esta tesis están fuertemente ligadas al éxito que logre el sistema de módulos de Java 9, dar continuidad al desarrollo de este servicio, puede ser una oportunidad para estar a la vanguardia en el ámbito del software orientado a componentes.

Nomenclatura

Binario	En Java a los archivos generados durante el proceso de compilación se los denomina binarios. No son exactamente archivos binarios, sino que poseen las instrucciones y una tabla de constantes que luego se ejecutarán en la Máquina Virtual de Java.
Bytecode	Es el conjunto de instrucciones que interpreta la Máquina Virtual de Java y están agrupados en métodos.
Class-path	Define para la Máquina virtual de Java o el compilador, el lugar donde se ubican las clases de usuario o bibliotecas externas.
Constant pool	Es la sección de un archivo binario Java donde se lista toda la información simbólica, precedida por una etiqueta que indica el tipo.
Descriptor	Es un archivo que explica las características de una biblioteca, proyecto o módulo. Normalmente contiene datos como nombre, requerimientos externos, servicios provistos y otros.
Jar	Es la extensión que poseen las bibliotecas de clases Java. En Java 9, los módulos también son archivos Jar pero deben incluir su archivo descriptor.
JCP	Java Community Process, es el proceso creado por Sun Microsystems para desarrollar estándares de Java. Actualmente es operado por Oracle. La JCP produce solicitudes de especificaciones Java (Java Specification Request o JSR).
JDK	Java Development Kit. Es el conjunto de recursos necesarios para crear software Java.
JEP	JDK Enhancement Proposals, es el listado de propuestas de ampliación para la JVM.
JIT	Just In Time Compiler. Es el proceso de traducción del código JVM en tiempo de carga o ejecución al conjunto de instrucciones nativas del CPU anfitrión.
JLS	Java Language Specification, es la especificación formal del lenguaje de programación Java.
JNLP	Java Network Launch Protocol, permite ejecutar aplicaciones de escritorio a partir de recursos almacenados en un servidor web remoto.

JPA	Java Persistence API, es una especificación para acceder, persistir y gestionar datos entre objetos Java y bases de datos relacionales.
JPMS	Java Platform Module System, es la implementación de la especificación JSR 376 que será parte de Java 9.
JSR	Java Specification Request, son documentos formales que describen propuestas de especificación para agregar a la plataforma Java.
JVM	Máquina Virtual de Java. Procesa archivos binarios Java.
JVMS	Java Virtual Machine Specification, es el documento que especifica una máquina virtual para Java.
Meta-datos	Son los datos o información que proveen información acerca de otros datos. Para el caso de los archivos Jar, los metadatos pueden ser el archivo manifiesto MANIFEST.MF o para un proyecto Maven el archivo descriptor de extensión POM.
OSGi	Originalmente significaba Open Service Gateway initative, actualmente corresponde a Dynamic Module System for Java.
PDE	Eclipse Plug-in Development Environment, son un conjunto de herramientas para crear, probar y desplegar elementos para el IDE Eclipse.
POM	Es el archivo descriptor de proyectos de Maven.
REST	Representational State Transfer, son servicios web para proveer interoperabilidad entre sistemas.
RFC2616	Request For Comments, es una especificación que define el protocolo HTTP/1.1
R-OSGi	Es tecnología que permite acceder de forma transparente a servicios OSGi remotos.

Referencias

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

[2] Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S., “A modular package manager architecture”, en *Information and Software Technology*, 55(2):459 -- 474, 2013. 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’?", en MIT Tecnology Review, 2012. En línea: https://www.technologyreview.com/s/425970/who-coined-cloud-computing/. Consultado: 03/01/2017.

[5] 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.

[6] Wang, P., Yang, J., Tan, L., Kroeger, R., Morgenthaler, D., ”Generating precise dependencies for large software”, en *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", en Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, 1998.

[8] Compiler Group, “Project Jigsaw”, en OpenJDK, Oracle Corporation, 2017. En línea: http://openjdk.java.net/projects/jigsaw/. Consultado: 03/01/2017.

[9] Reinhold, M., “JEP 220: Modular Run-Time Images”, en OpenJDK, Oracla Corporation, 2016. En línea: http://openjdk.java.net/jeps/220/. Consultado: 03/01/2017.

[10] Flotyński J., Krysztofiak K., Wilusz D., “Building Modular Middlewares for the Internet of Things with OSGi”, en *Galis A., 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”, en *The Server Side*, 2011. En línea: http://www.theserverside.com/news/thread.tss?thread_id=62523. Consultado: 04/01/2017.

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

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

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

[15]Smith, J., Ravi, N., “The architecture of virtual machines”, Computer, 2005.

[16]Presser, L., White, J., ”Linkers and Loaders”, en *ACM Computing Surveys,* 4, 3, 1972.

[17]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.

[18]Schildt, H., "Java: The Complete Reference", *Oracle Press. 9th Edition.* 2014.

[19]“Java Community Process”, Oracle Corporation, 2017. En línea: https://www.jcp.org. Consultado: 06/01/2017.

[20]Curran, P., Undheim, T., "The Java Community Process standardization, interoperability, transparency," en *7th International Conference on Standardization and Innovation in Information Technology,* Berlin, 2011.

[21]Reinhold, M., “JSR 376 – Java Platform Module System”, Oracle Corporation. En línea: https://www.jcp.org/en/jsr/detail?id=376. Consultado: 06/01/2017.

[22]Compiler Group, “Project Jigsaw: Context & History”, en OpenJDK, Oracle Corporation, 2017. En línea: http://openjdk.java.net/projects/jigsaw/history. Consultado: 06/01/2017.

[23]“JDK 9 Early Access with Jigsaw Project”, Oracle Corporation. En línea: https://jdk9.java.net/jigsaw/. Consultado: 06/01/2017.

[24]Goslin, J., Joy, B., Steele, G., Bracha, G., Bucley, A., “The Java Language Specification”, 8th Edition, Oracle America, 2015.

[25]Buckley, A., “JPMS: Modules in the Java Language and JVM”, Oracle Corporation. En línea: http://cr.openjdk.java.net/~mr/jigsaw/spec/lang-vm.html. Consultado: 06/01/2017.

[26]Reinhold, M., “JEP 220: Modular Run-Time Images”, OpenJDK. En línea: http://openjdk.java.net/jeps/220. Consultado: 07/01/2017.

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

[28]Rellermeyer, J., “Modularity as a Systems Design Principle”, Thesis for ETH Zurich, 2011.

[29]Seider, D., Schreiber, A., Marquardt, T., Brüggemann, M., "Visualizing Modules and Dependencies of OSGi-Based Applications," *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, 2016.

[30]Vogel, L., “Eclipse Rich Client Platform”, Vogella Series, 2015.

[31]Mancinelli, F., Boender, J., di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R., “Managing the Complexity of Large Free and Open Source Package-Based Software Distributions”, en *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering* (ASE '06). IEEE Computer Society. 2006.

[32]Ignatiev, A., Janota, M., Marques-Silva, J., “Towards efficient optimization in package management systems” en *Proceedings of the 36th International Conference on Software Engineering* (ICSE 2014). ACM. 2014.

[33]Varanasi, B., Belida, S., “Introducing Maven”, Apress, 1era Edición. 2014.

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

[35]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.

[36]Martin, R., "OO Design quality metrics: An analysis of dependencies". Object Mentor. 1994.

[37]Mausolf, J., Austin, Kimberly, Ann., “Classpath optimization in Java runtime environments”, Patente: US 9.069.582 B2. 2015.

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

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

[40]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.

[41]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.

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

[43]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: 09/01/2017.

[44]Bateman, A., Buckley, A., Gibbons, J., Reinhold, M., “JEP 261: Module System”, OpenJDK. En línea: http://openjdk.java.net/jeps/261. Consultado: 10/01/2017.

[45]Agüero, M., Ballejos, L., Pons, C., "Resolución de dependencias Java a demanda", en *4to Congreso Nacional de Ingeniería Informática / Sistemas de Información (CoNaIISI)*, 2016.

[46]Denise, J., “JEP 282: The Java Linker”, OpenJDK. En línea: http://openjdk.java.net/jeps/282. Consultado: 09/01/2017.

[47]Bartlett, N., “OSGi and Java 9 Modules Working Together”, En línea: http://njbartlett.name/2015/11/13/osgi-jigsaw.html. Consultado: 10/01/2017.

[48]Reinhold, M., “The State of the Module System”, OpenJDK. En línea: http://openjdk.java.net/projects/jigsaw/spec/sotms/. Consultado: 10/01/2017.

[49]Cervantes, H., Kazman, R., "Designing Software Architectures: A Practical Approach", Addison-Wesley. 2016.

[50]Clemens, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J., "Documenting Software Architectures: Views and Beyond", Addison-Wesley, 2011.

[51]Hohpe, G., Woolf, B., "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", Addison-Wesley Professional, 2003.

[52] “Apollo Spotify”, Spotify, 2017. En línea: http://spotify.github.io/apollo/. Consultado: 16/01/2017.

[53]“Java 9 Early Access with Project Jigsaw”, Oracle Corporation, 2016. En línea: https://jdk9.java.net/jigsaw/. Consultado: 11/01/2017.

[54]“NetBeans JDK 9 Support”, NetBeans, 2017. En línea: http://wiki.netbeans.org/JDK9Support. Consultado: 11/01/2017.

[55]“Java Network Launch Protocol”, Oracle Corporation, 2017. En línea: http://www.oracle.com/technetwork/java/javase/jnlp-136707.html. Consultado: 19/01/2017.

[56]Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T., "RFC2616 - 206 Partial Content", Sección 10.2.7, The Internet Society. 1999. En Línea: https://www.ietf.org/rfc/rfc2616.txt. Consultado: 19/01/2017.

[57]“RPM Package Manager”, RedHat. En línea: http://rpm.org/. Consultado: 21/01/2017.

[58]“Advanced Package Tool”, Debian. En línea: https://wiki.debian.org/Apt. Consultado: 21/01/2017.

[59]“PackageManagement”, Microsoft TechNet. Microsoft Corporation. 2015. En línea: https://blogs.technet.microsoft.com/packagemanagement/2015/04/28/introducing-packagemanagement-in-windows-10/. Consultado: 21/01/2017.

[60]“Bundler”, 2017. En línea: http://bundler.io/rationale.html. Consultado: 21/01/2017.

[61]“PIP”, 2017. En línea: https://pypi.python.org/pypi/pip. Consultado: 21/01/2017.

[62]“NuGet”, .NET Foundation. 2017. En línea: http://www.nuget.org. Consultado: 21/01/2017.

[63]O'Keefe, R., “Building object oriented programs out of pieces” en *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (ICOOOLPS '16). ACM, 2016.

[64]Adermann, N., Boggiano, J., “Composer”, 2017. En línea: https://getcomposer.org/. Consultado: 21/01/2017.

[65]“PHP Stardard Recommendation-04”, The PHP Framework Interop Group. En línea: http://www.php-fig.org/psr/psr-4/. Consultado: 21/01/2017.

[66] “Apache Ivy”, The Apache Software Foundation. 2014. En línea: http://ant.apache.org/ivy/. Consultado: 21/01/2017.

[67]“Scala sbt”, Lightbend, 2016. En línea: http://www.scala-sbt.org/. Consultado: 21/01/2017.

[68]“Apache Ant”, The Apache Software Foundation. 2017. En línea: http://ant.apache.org. Consultado: 21/01/2017.

[69]“Apache Maven”, The Apache Software Foundation. 2017. En línea: https://maven.apache.org/. Consultado: 21/01/2017.

[70]“Gradle”, Gradle Inc. 2016. En línea: https://gradle.org/. Consultado: 21/01/2017.

[71]Feldman, S., “Make --- A Program for Maintaining Computer Programs” en Journal of Software: Practice and Experience. Wiley. 1979.

[72]“Managing complexity. Most software projects fail to meet their goals. Can this be fixed by giving developers better tools?”. The Economist. En línea: http://www.economist.com/node/3423238. Consultado: 21/01/2017.

[73]Muschko, B. “Why Build Your Java Projects with Gradle Rather than Ant or Maven?”. Dr. Dobb’s. En línea: http://www.drdobbs.com/jvm/why-build-your-java-projects-with-gradle/240168608. Consultado: 21/01/2017.

[74]“Bower”, En línea: https://bower.io. Consultado: 21/01/2017.

[75]Robaldo, L., “Dependency Tree Semantics: Branching Quantification in Underspecification” en *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA 2007: Artificial Intelligence and Human-Oriented Computing* (AI*IA '07), Springer-Verlag, 2007.

[76]“GNU GUIX”, Free Software Foundation. 2017. En línea: https://www.gnu.org/software/guix/. Consultado: 21/01/2017.

[77]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” en *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '15). ACM, 2015.