Universidad Nacional de Quilmes

Departamento de Ciencia y Tecnología

Tecnicatura en Programación Informática

Trabajo de Inserción Profesional

Fronttier

Manejador de dependencias de la *capa de presentación* para plataformas basadas en lenguajes para la *JVM*

Alumno

Alan Rodas Bonjour

alanrodas@gmail.com

Directora

Dra. Gabriela B. Arévalo

garevalo@unq.edu.ar

noviembre 2014

Agradecimientos

Quiero agradecer a todos los profesores de la carrera de Tecnicatura en Programación Informática y a mis compañeros de clase quienes me acompañaron durante todo mi proceso de formación, me brindaron su apoyo, su compañía y cariño.

Quiero agradecer también a mi familia, por el aliento que me han brindado desde el principio de mis estudios, en especial a mis padres, quienes con mucho esfuerzo apostaron por mi educación.

A mi directora, Gabriela Arévalo, quiero agradecerle el apoyo y la paciencia durante el desarrollo de este trabajo.

Por último, no puedo dejar de agradecer a Gabriela Guibaud, quien no solo ha revisado y corregido este documento, sino que me ha brindado su apoyo y compañía incondicional durante todo este tiempo.

Índice

Resumen

En el desarrollo de sistemas web utilizando lenguajes que corren en la Java Virtual Machine (*JVM*), es generalmente necesario descargar *dependencias* para la *capa de presentación* y agregarlas al código del sistema de forma manual. Esta tarea lleva tiempo, es tediosa y propensa a errores. Actualmente no hay herramientas que corran enteramente sobre la *JVM* y que automaticen esta tarea. Este trabajo propone una solución al problema mediante el desarrollo de un *manejador de dependencias* hecho completamente en *Scala*, capaz de descargar automáticamente estos archivos, organizarlos en el código y dejarlos configurados para su posterior uso.

1. Introducción: Problemática de usar librerías web en la *JVM*

1.1. Desarrollo de sistemas web en la *JVM*

La creciente popularidad de los lenguajes que corren sobre la Máquina Virtual de Java (*JVM*)[1], y los cambios por los que ha transitado el desarrollo de soluciones empresariales a lo largo del tiempo hacen de esta plataforma una de las principales elecciones para los programadores.
El lenguaje *Java* fue el primero en correr sobre la plataforma de *Oracle*. Con el transcurso del tiempo, muchos otros lenguajes que compilan a *bytecode Java* han aparecido. *Scala*, *Clojure* y *Groovy* son algunos de los más populares y han dotado a la plataforma de muchas nuevas herramientas y posibilidades.
Por otro lado, en la industria del software para desarrollo empresarial las tecnologías web se han posicionado como la opción predilecta. Esto se ha debido a su facilidad de desarrollo, su bajo costo, y la facilidad de crecimiento.

[1] La *JVM* es una parte fundamental de la plataforma del lenguaje Java, creada originalmente por *Sun Microsystems*. La maquina virtual crea una capa de abstracción entre el sistema operativo y el código Java compilado (*bytecode*). De esta forma, el código *Java* puede ser compilado a *bytecode* una sola vez, y correrse en cualquier sistema que cuente con una *JVM*.

La aparición de *HTML5*[2] para la creación de aplicaciones web, que permite un alto grado de usabilidad, junto con las nuevas plataformas que brindan servicios de hosting de alto rendimiento y escalabilidad a precios accesibles potencian esta elección.

1.2. Estructura común de sistemas web

La mayoría de los sistemas web suelen dividirse en capas (también llamadas *layers* o *tiers*), donde cada una está encargada de funcionalidades muy puntuales de la lógica de la aplicación.
La mayoría de los autores suelen identificar tres capas, las cuales suelen recibir distintos nombres, aunque la estructura es siempre la misma. Estas capas son: la de interfaz o *presentación*, la de *lógica de negocios*, modelo o aplicación y la de *datos* o persistencia.
La capa de datos es la encargada de manejar los accesos a la base de datos, persistir la información, y todo lo relacionado a elementos de los que se deba guardar registro. La capa de lógica de negocios es la que contiene el código de la funcionalidad de la aplicación. La capa de interfaz es la que permite acceder a la funcionalidad y visualizar los datos a los usuarios o clientes [Baarish, 2002, pag 29-30]. La figura 1 muestra la arquitectura básica de una aplicación web según dicho esquema.

1.3. Reutilización de código mediante *dependencias*

Existen numerosos *frameworks*, *toolkits*, bibliotecas y porciones de código que los desarrolladores pueden utilizar en el código de su proyecto para obtener funcionalidades genéricas previamente desarrolladas. Estas porciones de código de terceros reciben el nombre de *dependencias*.
En éste trabajo se identificaran dos tipos de *dependencias*: las de la *capa de lógica de negocios*, dadas por *paquetes* de código *Java* compilado (archivos .jar[3] o .war[4]), y las de *capa de presentación* que consisten en (pero no se

[2] HyperText Markup Language versión 5. *HTML* es el lenguaje estándar para el desarrollo de sitios web. El termino *HTML5* suele hacer referencia no solo a la quinta versión de este lenguaje, sino también a las tecnologías *JavaScript* y *CSS* que complementan el mismo para generar aplicaciones ricas en contenido.

[3] .jar es una extensión de archivo utilizada por aplicaciones de la plataforma *Java*. Es un archivo comprimido que contiene en su interior código *bytecode Java* junto con metadata. Son, básicamente, programas que corren en la *JVM*.

[4] .war es una extensión de archivo utilizada por aplicaciones de la plataforma *Java* que corren en servidores web.

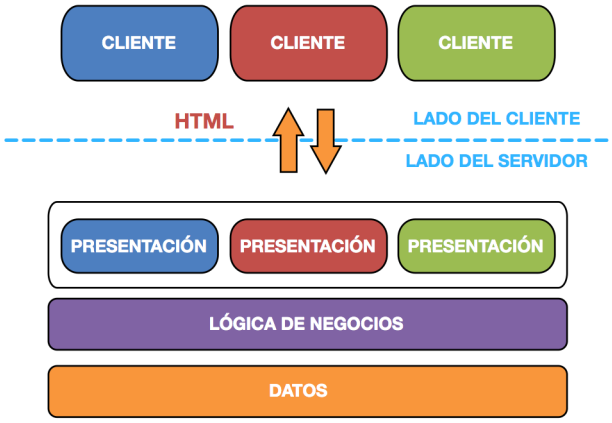


Figura 1: Arquitectura em capas básica de un sistema web.

limitan a) archivos *CSS* y *JavaScript*. A su vez, dichas *dependencias* pueden ser manejadas o no manejadas.
Las *dependencias* no manejadas son código que se agrega manualmente al proyecto mediante la clásica técnica de *"copiar y pegar"* o agregando los archivos correspondientes a esa *dependencia* en una carpeta del proyecto (proceso que se denominará *"instalar"*). Por su parte, las *dependencias* manejadas consisten en algún *archivo de configuración* que describe inequívocamente (generalmente mediante el nombre y la versión) las *dependencias* de terceros requeridas en el código a desarrollar. Un programa externo evaluará el *archivo de configuración* al momento de la compilación y se encargará de descargar e *instalar* las *dependencias* declaradas. Estos programas son conocidos como *manejadores de dependencias*. La figura 2 muestra los tipos de *dependencias*.

Las *dependencias* no manejadas implican una serie de pasos repetitivos y propensos a errores que el usuario debe realizar para correr su código. Por su parte, las *dependencias* manejadas son preferentes, ya que eliminan errores comunes y ahorran tiempo y trabajo a los desarrolladores. Además, los *manejadores de dependencias* suelen simplificar la complejidad asociada a instalar las *dependencias* anidadas[5] [Larman y Vodde, 2010].
Las *dependencias* de la *capa de presentación* en la plataforma *Java*, suelen

[5] Asuma el lector un proyecto *A* con una *dependencia B*. El paquete *B* es a su vez un proyecto con *dependencias*, por ejemplo *C*. Para que *A* funcione correctamente, requiere *B* y este a su vez requiere *C*, haciendo que de forma transitiva *A* requiera *C*.

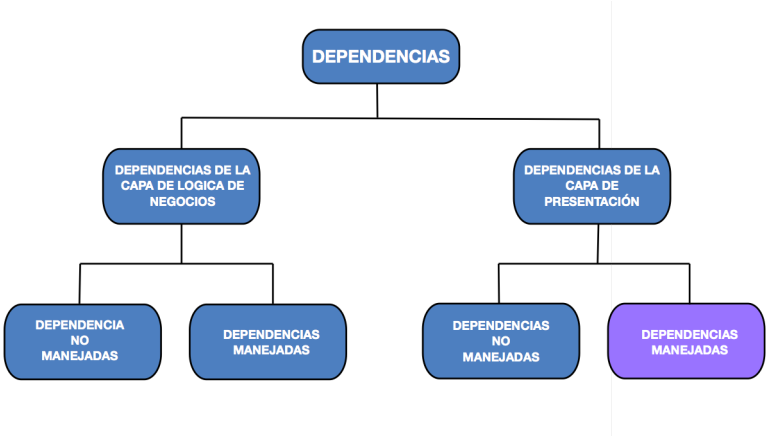


Figura 2: Distintos tipos de *dependencias*.

ser no manejadas. El presente trabajo se enfoca en transformar esas *dependencias*, en *dependencias* manejadas.

El presente trabajo tratará la implementación de un *manejadores de dependencias* para descargar librerías web (archivos *CSS*, *JavaScript*, y otros) en sistemas que compilan a *bytecode Java*. Para hacer un *manejador de dependencias* de la *capa de presentación* se utilizarán conceptos utilizados por herramientas existentes que trabajan sobre la *capa de lógica de negocios*, además de integrarse con dichas herramientas cuando sea posible.

2. Solución Propuesta: Manejador de *dependencias* web para la JVM

Basándonos en el problema de manejar *dependencias* en la *capa de presentación* presentado en el capítulo 1, es necesaria la implementación de una herramienta para los desarrolladores que corra sobre la *JVM* y permita manejar las mencionadas *dependencias*.
El presente trabajo propone como solución el desarrollo de un *manejador de dependencias* implementado en *Scala* llamado *Fronttier*. El mismo utilizará *archivos de configuración* con distintos formatos para configurar las *dependencias* requeridas. Adicionalmente, se integrará con herramientas externas para facilitar su uso con los flujos de trabajo ya establecidos.

2.1. Elección del nombre

La herramienta recibirá el nombre de *Fronttier*. El mismo deviene de un juego de palabras entre *front*, nombre que se le suele dar a la *capa de presentación*, y *tier*, denominación que recibe cada una de las capas de la aplicación.

2.2. Características deseadas

El objetivo del presente trabajo es la generación de una herramienta que se posicione como el estándar de los lenguajes de la *JVM* en el manejo de *dependencias* de la *capa de presentación*. Las siguientes características son deseables en la herramienta a desarrollar.

- **Hecho enteramente sobre la *JVM***
  Deberá correr enteramente sobre la *JVM*, evitando depender de la instalación de software adicional en el equipo y haciéndolo independiente del sistema operativo. Por ésta razón es necesario que se encuentre desarrollada íntegramente en uno o varios lenguajes que compilen a *bytecode Java*.
- **Posibilidad de ser usado desde cualquier lenguaje de la *JVM***
  Deberá poder ser utilizada desde cualquier otro lenguaje de la *JVM* de forma natural. En caso de características incompatibles de los lenguajes, debería proveer una capa de abstracción de forma tal que el uso sea transparente y completo. Incluso es deseable que pueda ser ejecutada en forma de programa independiente desde la *línea de comandos*. Ésto último posibilitaría que pueda ser utilizado prácticamente desde cualquier lenguaje, aunque implica contar con acceso al sistema de archivos del sistema operativo.
- **Archivos de configuración personalizados, con varios formatos por defecto**
  La herramienta deberá ser capaz de descargar y acomodar los paquetes que el usuario determine como *dependencias* en un *archivo de configuración*. En lugar de forzar un formato específico para el *archivo de configuración*, el *manejador de dependencias* desarrollado deberá brindar al usuario la posibilidad de usar un formato que se adecúe a sus necesidades. Por ejemplo, los usuarios de *Apache Maven* podrían optar por un archivo de configuración en *XML* similar al que acostumbran y conocen, mientras que los usuarios de *SBT* podrían elegir un archivo de configuración cuya sintaxis asemeje los usados en esa herramienta.

Debería ser lo bastante flexible para permitir la generación de formatos de *archivo de configuración* por los mismos usuarios de la herramienta de forma sencilla.

- **Múltiples repositorios**
  Finalmente, sería ideal contar con un repositorio central, donde desarrolladores puedan subir sus paquetes, o al menos registrar los mismos, permitiendo así consultar la existencia de *dependencias*. Además, los usuarios podrían querer generar sus propios repositorios y hacer que la herramienta consulte en ellos, evitando la centralización del repositorio. La dualidad entre contar con un repositorio centralizado, y múltiples repositorios descentralizados, permite a organizaciones de distinto tamaño contar con posibilidades acorde a sus requerimientos, y se ha mostrado exitosa en proyectos como *Maven*.
- **Seguridad mediante autenticación en los repositorios**
  Algunos de éstos repositorios podrían ser privados, y por tanto, se requeriría acceso mediante un usuario y contraseña que la herramienta debe enviar de forma segura.
- **Integración con estándares existentes de *manejadores de dependencias***
  Al existir soluciones como *Bower* o *Component* la herramienta debería complementarlas, permitiendo, por ejemplo, utilizar sus archivos de configuración y sus repositorios. También es deseable que se integre con los *manejadores de dependencias* que corren sobre la *JVM*, de forma tal que el usuario se pueda despreocupar de tener que correr múltiples programas (el *manejador de dependencias* para la *capa de lógica de negocios* y *Fronttier* para las *dependencias* de la *capa de presentación*).
- **Integración con tecnologías web existentes en la *JVM***
  Finalmente, al existir numerosos *frameworks* para distintos lenguajes de la *JVM* que apuntan al desarrollo de aplicaciones web, sería deseable la integración con los mismos. Por ejemplo, múltiples *frameworks* utilizan sistemas de templates para generar el código *HTML* que finalmente será utilizado. Sería ideal poder brindar algún componente que integre los elementos descargados a los templates del usuario de forma automática.

Se puede agregar además que la mayoría de los *manejadores de dependencias* actuales se basan en archivos, es decir, un archivo por *dependencia*. En las *dependencias* de la *capa de presentación* esto no es necesariamente cierto, y a veces una *dependencia* consiste en múltiples archivos. La herramienta tiene que proveer una forma de contemplar los casos señalados para permitir bajar todos los archivos que sean necesarios.

2.3. Scala como lenguaje de implementación

Se utilizará *Scala* para desarrollar la aplicación. La elección se basa en que la combinación de *Programación Orientada a Objetos* y *Programación Funcional* que presenta el lenguaje permiten desarrollar complejas soluciones en cortos períodos de tiempo. Además, el código desarrollado es de fácil mantenimiento y gran expresibilidad, dado a la capacidad de *Scala* de crear sencillos DSLs.
Adicionalmente, el lenguaje elegido presenta una gran cantidad de bibliotecas estándar que permiten realizar rapidamente tareas complejas. Por ejemplo, su biblioteca de análisis sintáctico (*parsing*) resultará muy útil cuando se requiera leer archivos de configuración.
Scala será utilizado en combinación con *SBT* para manejar las *dependencias* del proyecto.
Al momento de desarrollar herramientas de integración con otros lenguajes o tecnologías existentes es posible que sea necesario el uso de otros lenguajes, como *Java*, *Clojure* o *Groovy*.

2.4. Alcance del presente trabajo

El alcance del presente trabajo no abarca la totalidad de las características deseadas expresadas en la sección 2.2. El mismo solamente se enfocará en las funcionalidades más importantes y dejará asentadas las bases para el desarrollo de las partes no implementadas en trabajos futuros. Así, el enfoque será puesto en la estructura del sistema, orientada siempre a la extensibilidad del mismo.
Se presenta como objetivo el desarrollo de las siguientes funcionalidades:

- Desarrollo del núcleo del sistema
- Descargar *dependencias* desde la web vía *HTTP*[6]
- Descargar desde *GitHub*[7] mediante *GIT*[8]
- Descargar desde un repositorio *SVN*[9]

[6] HTTP es el protocolo usado en cada transacción de Internet. HTTP define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (clientes, servidores, proxies) para comunicarse. Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor.

[7] GitHub es un popular sitio web que provee repositorios *GIT* gratuitos para proyectos de código abierto.

[8] GIT es un software de control de versiones distribuido, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente.

[9] Subversion (SVN) es una herramienta de control de versiones open source basada en un repositorio cuyo funcionamiento se asemeja enormemente al de un sistema de ficheros.

- Lectura de *archivos de configuración* en *XML*[10] estilo *Apache Maven*
- Lectura de *archivos de configuración* en *XML* estilo *Apache Ivy*
- Agregado de múltiples repositorios
- Usar la herramienta desde la *línea de comandos*
- Usar la herramienta desde *Scala*
- Integración de la herramienta con *SBT*
- Integración de la herramienta con Play! Framework

La herramienta será liberada como *Software Lîbre* con licencia *Apache* versión 2. Esto permitirá que la comunidad de desarrolladores la expanda para funcionar con todos los lenguajes de la *JVM*, en muchos más *frameworks*, y que se integre con una mayor cantidad de procesos.

3. Uso de la herramienta

La presente sección muestra el uso de la herramienta desarrollada. Inicialmente se verá su uso mediante la *línea de comandos*. Luego, se mostrará la forma en la que la herramienta puede ser usada y extendida desde otros lenguajes.

3.1. Uso desde la *línea de comandos*

Para correr la aplicación desde la *línea de comandos* es necesario ejecutar el archivo *JAR* de la herramienta, a través de la *JVM*. El siguiente código muestra como ejecutar *Fronttier* desde la interfaz de *línea de comandos*:

```
$ java -jar fronttier.jar
```

Adicionalmente, se puede pasar una serie de argumentos a la aplicación para configurarla a gusto y necesidad del usuario, tal como se muestra a continua-

[10] XML es un lenguaje de marcas utilizado para almacenar datos en forma legible. Se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas.

ción:

```
$ java -jar fronttier.jar <argumentos>
```

Con intención de simplificar el trabajo del usuario al momento de ejecutar la aplicación, *Fronttier* incluye un programa que puede ser colocado en cualquier lugar accesible para el usuario[11], permitiendo ejecutar la aplicación de forma directa, tal como se muestra en el siguiente ejemplo:

```
$ fronttier <argumentos>
```

A continuación se verán los argumentos posibles en más detalle, así como la forma de configuración de la herramienta.

3.1.1. Uso con los valores por defecto

Es posible llamar al programa sin ningún argumento y obtener funcionalidad del mismo mediante convenciones que *Fronttier* utiliza.
Al ser ejecutada sin ningún tipo de argumentos, *Fronttier* utilizará los valores por defecto de la aplicación. Éstos incluyen: utilizar la *cache global*, buscar *dependencias* en un archivo de configuración con el formato estándar de *Fronttier* con el nombre ***fronttier.ftt*** e instalar las *dependencias* descargadas en el directorio local.

3.1.2. Selección del archivo de configuración y su formato

Por defecto, *Fronttier* intenta localizar en el directorio actual un archivo con el nombre de `fronttier.ftt`. El mismo es donde se declaran las *dependencias* y repositorios del *proyecto*.
A su vez, el usuario puede especificar un archivo distinto en donde buscar *dependencias*. Ésto se logra mediante la opción ***--filename*** o ***-F*** pasada como argumento a la aplicación e incluyendo luego el nombre del archivo que se desea utilizar para declaración de *dependencias* y repositorios. A continuación se puede observar un ejemplo de cómo llamar a la aplicación para

[11] Un lugar accesible es, por ejemplo, cualquier carpeta que se encuentre dentro de la variable de entorno *"PATH"* en el sistema operativo del usuario.

que lea un archivo llamado ***dependencies.ftt***:

```
$ fronttier --filename dependencies.ftt
```

Los archivos mencionados pueden tener internamente distintos formatos, es decir, distintas formas de representar la información. Cada formato posee un nombre único por el cual el usuario puede identificarlo y, así, indicarle a la herramienta qué debería esperar de los contenidos del archivo de configuración. Esto se logra mediante el argumento ***--config*** o ***-c*** . El siguiente código muestra como elegir un formato con nombre ***xml***:

```
$ fronttier --config xml
```

Los formatos registrados por defecto en *Fronttier* son:

- fronttier
- xml
- attrxml

Sin embargo, el usuario puede registrar nuevos formatos mediante extensiones a la herramienta. En caso de no especificar el formato del archivo, el formato *fronttier* es el utilizado.
El formato elegido del archivo repercute sobre el nombre por defecto del mismo. Por ejemplo, el nombre de archivo por defecto para el formato ***xml*** es ***fronttier.xml***, al igual que para el formato *attrxml*. Por su parte, el formato ***fronttier*** buscará un archivo por el nombre de ***fronttier.ftt***.
Los argumentos ***--config*** y ***--filename*** se pueden usar al mismo tiempo dando lugar a cualquier combinación de nombre de archivo y formato. Por ejemplo, el siguiente código muestra cómo ejecutar *Fronttier* para que utilice un archivo ***dependencies.xml*** que posee formato ***xml***.

```
$ fronttier --config xml --filename dependencies.xml
```

La estructura interna de cada uno de los formatos, así como la forma de declarar *dependencias* y repositorios es especificada en la sección 3.4.

3.1.3. Elección de carpeta de destino

Por defecto *Fronttier* trabaja sobre la carpeta actual, y es allí donde se *instalan* las *dependencias* descargadas.
No obstante, la carpeta sobre la que trabajará la herramienta puede ser especificada, de forma tal que las *dependencias* se instalen en alguna carpeta

específica o una subcarpeta de la carpeta local. Para ello se utiliza el comando ***--destination*** o ***-d*** seguido de la ubicación en donde se desean descargar las *dependencias*.

```
$ fronttier --destination <folder>
```

El valor por defecto es **.** (punto), es decir, la carpeta actual. Por tanto, los siguientes dos comandos son equivalentes:

```
$ fronttier
```

```
$ fronttier --destination .
```

La elección de la carpeta de destino resulta útil en proyectos donde el código compilado termina en una carpeta específica que no forma parte del proyecto y que se elimina cada vez que se recompila el mismo. Así, o más habitual es elegir un subdirectorio de la carpeta de trabajo donde se deben colocar las *dependencias* previo a ejecutar el sistema. El siguiente ejemplo muestra cómo indicarle a *Fronttier* que utilice como carpeta de descarga un subdirectorio con el nombre de *"WEB-INF/html"*:

```
$ fronttier --destination ./WEB-INF/html
```

Si bien no es algo común, se pueden especificar también direcciones absolutas en lugar de relativas. En caso de que las direcciones contengan espacios se debe utilizar comillas simples o dobles para especificar la dirección completa, como se muestra en el siguiente ejemplo:

```
$ fronttier --destination "C:\Web Dependencies"
```

Se debe tener en cuenta que *Fronttier* no realiza ninguna acción para limpiar la carpeta de destino, por lo que el usuario deberá garantizar su limpieza previa a futuras ejecuciones.

3.1.4. Cache local y cache global

Fronttier mantiene en la máquina del usuario una cache con todos los archivos que va descargando a lo largo de su uso. Ésto permite acelerar el proceso de *instalación* de las *dependencias* en un proyecto particular. Así, *Fronttier*

mantiene dos lugares que utiliza como cache, la *cache local* y la *cache global*. La *cache local* tiene alcance sólo sobre el proyecto de código sobre el cual se está trabajando (es decir, el *archivo de configuración* que se está leyendo y el directorio sobre el cual se están instalando las *dependencias*). Su uso se limita a evitar tener que descargar las *dependencias* en subsecuentes ejecuciones del programa en el mismo proyecto.
Por su parte, la *cache global* contiene las *dependencias* descargadas por todos los proyectos. Ésta cache se utiliza, no sólo para evitar la descarga de *dependencias* en futuras ejecuciones del programa sobre un proyecto determinado, sino además para evitar la descarga en caso de que dos proyectos independientes requieran eventualmente la misma *dependencia*.
Las caches permiten adicionalmente trabajar sin la necesidad de una conexión a Internet, ya que no se deben descargar las *dependencias*, pues ya se encuentran el equipo.
La *cache global* se encuentra en una carpeta creada por *Fronttier* en el directorio personal del usuario. Por su parte, la *cache local* se encuentra en una carpeta similar, pero localizada en el directorio del proyecto sobre el que aplica dicha cache. Las caches se ubican en los siguientes directorios.

Cache global
~/.ftt/Repository

Caches local
%PROJECT_FOLDER%/.ftt/Repository

Donde *"%PROJECT_FOLDER%"* es el directorio del proyecto sobre el cual se está trabajando.
Para comprender mejor el sistema de caches es necesario comprender cómo *Fronttier* descarga *dependencias*. En primer lugar, *Fronttier* leerá el *archivo de configuración* y analizará las *dependencias* a instalar. Para cada *dependencia*, se evaluará si existe en la *cache local*. Si se encuentra, se copiará desde la misma a la carpeta donde se requiera instalar la *dependencia*. En caso de no encontrarse, se recurrirá a buscarla en la *cache global* para copiarla desde allí. Finalmente, si no se encuentra en las caches, se descargará de alguno de los *repositorios en línea* disponibles. *Fronttier* indicará un error al usuario si no encuentra en éstos últimos la *dependencia*.
Fronttier copiará por defecto las *dependencias* que descargue desde *repositorios en línea* a la *cache global* para futuros usos. Éste comportamiento está dado mediante el *flag* ***--global*** o ***-g*** . Así, los siguientes dos códigos son

equivalentes:

```
$ fronttier
```

```
$ fronttier -g
```

Mediante el *flag* ***--local*** o ***-l*** se puede indicar que no se desea copiar las *dependencias* a la *cache global* sino a la local. Esto puede resultar útil en algunos proyectos. El *flag* ***--nocache*** evita que se guarden las *dependencias* descargadas en cualquiera de las caches.
Por último, se puede utilizar el *flag* ***-f*** o ***--force*** para forzar la descarga de una *dependencia* desde los *repositorios en línea* incluso si se encuentran en alguna cache. Esta acción reemplazará la versión actual en la cache por la recién descargada.
Todos estos *flags* pueden combinarse para dar lugar a acciones determinadas. Por ejemplo, el siguiente código muestra cómo descargar *dependencias* sin guardarlas en la cache, y utilizando siempre la versión disponible en los *repositorios en línea*.

```
$ fronttier --nocache -f
```

También es posible descargar las *dependencias* en la *cache local*, independientemente de si se encuentran en alguna de las caches, como muestra el siguiente código.

```
$ fronttier -lf
```

Cabe destacar que los *flags* en su formato corto pueden agruparse utilizando un guión solo (-) y pasando todos los *flags* juntos. Así ***-lf*** es equivalente a ***-l*** ***-f*** .

3.1.5. Modo verbose

Por defecto *Fronttier* brinda muy poca información al usuario con respecto a las acciones que realiza en cada momento, indicando solamente los errores que se puedan llegar a producir. Sin embargo, es posible cambiar dicho comportamiento activando el modo ***verbose***.
Precisamente el *flag* ***--verbose*** o ***-v*** activa este modo permitiendo ver información de todos los pasos que la herramienta va realizando. Esto resulta útil en caso de errores inesperados para poder descubrir el origen de los mismos.

3.1.6. Agregar archivos de expansión

La herramienta *Fronttier* puede ser extendida por código del usuario. Esto permite, entre otras cosas, agregar nuevos formatos de *archivo de configuración*.
Para extender el programa, basta cargar los archivos *JAR* (código compilado a *bytecode Java*) que contienen las extensiones, a los cuales llamaremos *plugins*. Ésto se realiza mediante el argumento ***--load*** o ***-L*** el cual recibe el o los *plugins* a cargar. Por ejemplo, para cargar los *plugins "first.jar"* y *"second.jar"* se debería ejecutar la aplicación de la siguiente forma:

```
$ fronttier -L first second
```

La extensión del archivo no es necesaria pues se asume que siempre será .jar. Los archivos de extensión, en este caso, se encuentran en la carpeta actual desde donde se ejecuta la aplicación. También se pueden utilizar rutas absolutas o relativas para llamar a los archivos como se muestra a continuación:

```
$ fronttier --load "C:\Fronttier Extensions\first "
```

Finalmente, cabe destacar la posibilidad de *instalar* extensiones en una carpeta global o local para que puedan ser utilizados en cualquier momento. Esta carpeta actúa de forma similar a la cache para los archivos de *dependencia*. De hecho, su ubicación es precisamente en un directorio hermano a la carpeta del repositorio de cache. A continuación se detalla su ubicación:

Cache de plugins global
~/.ftt/Plugins

Caches de plugins local
%PROJECT_FOLDER%/.ftt/Plugins

Donde *"PROJECT_FOLDER"* es el directorio del proyecto sobre el cual se está trabajando.
Los archivos *JAR* que se encuentren en una carpeta de *cache de plugins* pueden ser utilizados como si se encontraran en la carpeta actual, es decir, mediante su nombre.
Cuando *Fronttier* recibe un *plugin* a cargar que no contiene ruta, primero evaluará su existencia en la carpeta actual, luego en la *cache de plugins local*, y finalmente en la *cache de plugins global*. Utilizará entonces aquel que

encuentre primero, indicando un error en caso de no encontrar coincidencias.

3.2. Comandos de manejo de la aplicación

Fronttier incluye una serie de comandos adicionales para facilitar el manejo de la herramienta y realizar tareas básicas sin necesidad de utilizar un archivo de configuración o realizar dichas tareas manualmente. Esto incluye cosas como instalar *dependencias*, copiar *dependencias* a la cache, borrar *dependencias* de la cache, instalar o desinstalar *plugins*, etc.

3.2.1. Instalación de *dependencias*

Es posible instalar *dependencias* sin la necesidad de declararlas en un *archivo de configuración* mediante el uso del comando ***install***. Este comando tomará como argumento el nombre de la *dependencia* a instalar. El siguiente código muestra cómo funciona el comando:

```
$ fronttier install {company}:{package}:[{version}]
```

Donde ***company*** y ***package*** son cadenas de texto que hacen referencia inequívoca a la *dependencia* a instalar. Opcionalmente ***version*** puede ser especificado para indicar una versión puntual del mismo (si no se indica, se descargará la más reciente). El siguiente código muestra un ejemplo puntual para descargar la biblioteca ***jQuery*** en su versión ***1.9.0***:

```
$ fronttier install org.jquery:jquery:1.9.0
```

La mayoría de los argumentos descriptos en la sección 3.1 pueden ser utilizados, quedando exceptuados aquellos que refieren a los *archivos de configuración*. Por ejemplo, el argumento ***--destination*** o ***-d*** , especificados en la sección 3.1.3, puede ser utilizado como se muestra a continuación:

```
$ fronttier install org.jquery:jquery:2.0 -l -d ./WEB-INF/html
```

La búsqueda del archivo en las caches es exactamente igual a cuando se trabaja con un *archivo de configuración*.

3.2.2. Cachear *dependencias*

El comando ***cache*** y el comando ***uncache*** permiten guardar y borrar *depen-*

dencias en la cache de *dependencias* de *Fronttier*. Se puede entonces pedir que un determinado archivo se descargue a la cache de la siguiente forma:

```
$ fronttier cache org.jquery:jquery:1.9.0
```

Los argumentos ***--global*** o ***-g*** y ***--local*** o ***-l*** , presentados en la sección 3.1.4, pueden ser utilizados para indicar exactamente sobre qué cache se va a trabajar. Por ejemplo, el siguiente código remueve un archivo de la *cache local*:

```
$ fronttier uncache org.jquery:jquery:1.9.0 --local
```

3.2.3. Instalar *plugins*

Los *plugins* pueden ser instalados a las *caches de plugins* mediante los comandos ***plug*** y desinstalados con el comando ***unplug***. Los *flags* ***--global*** o ***-g*** y ***--local*** o ***-l*** , presentados en la sección 3.1.4, pueden ser utilizados. A continuación se muestra cómo instalar un *plugin* desde la carpeta local a la *cache de plugins global*:

```
$ fronttier plug myCoolPlugin
```

Y aquí cómo desinstalar un *plugin* desde la *cache de plugins local*:

```
$ fronttier unplug myCoolPlugin -l
```

3.3. Archivos de defaults de la herramienta

Fronttier trabaja con varios valores predeterminados, como el formato del *archivo de configuración* a utilizar o qué cache usar. Éstos valores pueden ser modificados mediante los llamados archivos de defaults de *Fronttier*.
Así, pueden existir una serie de archivos que deben tener por nombre *defaults* y que modifican los valores por defecto de la herramienta cuando ésta es utilizada desde la *línea de comandos*. Éstos archivos se encuentran en:

Archivo de *defaults* global
~/.ftt/defaults

Archivo de *defaults* local
%PROJECT_FOLDER%/.ftt/defaults

Donde *"%PROJECT_FOLDER%"* es el directorio del proyecto sobre el cual se está trabajando.
Cada archivo posee configuraciones para la ejecución por defecto de *Fronttier*. En primer lugar, se cargarán los valores por defecto generales, luego se leerán los valores del archivo *defaults* global. En este momento, los valores definidos en ese archivo *"sobreescribirán"*[12] a los valores por defecto. Finalmente, se leerá el archivo de *defaults* local, *"sobreecribiendo"* los valores que hayan resultado de leer el archivo de *defaults* global.
Fronttier se distribuye con un manual de uso que provee las indicaciones necesarias para crear y modificar éstos archivos, así como los contenidos esperados en los mismos.

3.4. Distintos formatos del archivo configuración

Como ya se ha mencionado en el presente documento, existen múltiples formatos de archivos de configuración posibles. El formato elegido usando el método expresado en la sección 3.1.2 influye así en la forma en la que deben estar declaradas las *dependencias* en el archivo. En las secciones siguientes se muestran los formatos incluidos por defecto con la herramienta.

3.4.1. Formato de configuración estándar

El formato de configuración estándar o formato *Fronttier* es el método por defecto para declarar las *dependencias* y repositorios.
En este formato se deben declarar dos secciones: ***repositories*** y ***dependencies***. La sección ***repositories***, la cual es opcional, indica los repositorios adicionales que se desean utilizar, uno por línea, o separados por punto y coma. La sección ***dependencies*** es obligatoria e indica las *dependencias* requeridas por el proyecto. Las secciones se encuentran delimitadas por llaves, y sólo puede haber una sección ***dependencies*** y una sección ***repositories***.
En la sección ***dependencies*** cada *dependencia* es declarada en una nueva línea (o separándolas por punto y coma) indicando la ***company*** seguida de dos puntos y el nombre (***name***). De forma opcional, se puede agregar dos puntos luego del nombre e indicar la versión puntual que desea instalar.
En este formato de archivo, los espacios en blanco no son tenidos en cuenta

[12] Es decir, se dejará el valor por defecto para todas aquellos atributos a los cuales no se les haya asignado un nuevo valor en el archivo *defaults*, pero en caso de que se haya declarado un valor, se utilizará este último, descartando el valor anterior.

y se pueden agregar comentarios de línea mediante el caracter #.
El código 1 muestra un ejemplo de una declaración de este formato de archivo.

```
repositories {
  http://myrepository.com
  https://some-other-repo.org/repository
}

dependencies {
  org.jquery:jquery:2.1.1; org.jquery:jquery-ui:1.11.0
  # Ésto es un comentario
  com.twitter:bootstrap:3.2.0
}
```

Código 1: Ejemplo del formato *fronttier*

El formato Backus-Naur Extendido para éste archivo se muestra de forma simplificada en el código 2.

```
string          = "\s", {"\s"}
repository      = [string "::"], string
repositories    = "repositories {" {repository} "}"
dependency      = string, "::", string, ["::", string]
dependencies    = "dependencies {" {dependency} "}"
depsfirst       = dependencies, [repositories]
depslast        = [repositories], dependencies
grammar         = depsfirst | depslast
```

Código 2: EBNF de *fronttier*

Éste formato se encuentra registrado bajo el identificador *fronttier*.

3.4.2. Formato de configuración estilo *Maven*

El formato *XML* permite configurar las *dependencias* y repositorios utilizando *XML*, que puede ser más cómodo para los usuarios de *Maven*. En este formato los repositorios, *dependencias* y todas las propiedades de cada una de las *dependencias* se expresan como texto entre tags *XML* anidados.
Los tags posibles y sus anidaciones están dadas por un archivo DTD, el cual se puede apreciar en el código 3.

```
<!ELEMENT fronttier (repositories?, dependencies)>
<!ELEMENT repositories (repository*)>
<!ELEMENT repository (type?, url)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT dependencies (dependency*)>
<!ELEMENT dependency (group, name, version?)>
<!ELEMENT group (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT version (#PCDATA)>
```

Código 3: DTD del formato *xml*

Los nombres y las convenciones utilizadas en el formato se corresponden completamente con aquellos usados por *Apache Maven* para una más fácil adopción por parte de los usuarios de esta herramienta. En el código 4 se muestra un ejemplo del formato ***xml***.

3.4.3. Formato de configuración estilo *Ivy*

Además del formato ***xml***, el formato ***attrxml*** también utiliza *XML* para declarar *dependencias*. A diferencia del anterior, éste es más compacto, ya que los valores se declaran como atributos de los tags *XML*.
Dicho formato respeta las convenciones impuestas por *Apache Ivy*, pudiendo ser de interés para los usuarios de la herramienta.
La estructura de éstos archivos está dada por el DTD que se muestra en el código 5

El código en 6 muestra un ejemplo del formato descripto.

3.5. Integración en sistemas

Fronttier no es solamente una herramienta que corre desde la *línea de comandos*, sino que, por ser un archivo *JAR*, puede ser utilizado como una biblioteca de la plataforma *Java*. Ésto permite que se utilice desde otros programas escritos para cualquiera de los lenguajes que corren sobre la *JVM*.
Para cada acción que se puede ejecutar desde la *línea de comandos*, *Fronttier* provee objetos y métodos que pueden ser utilizados desde el código del usuario. Esta es la forma en que se pueden crear extensiones para la herramienta,

```xml
<?xml version="1.0"?>
<!DOCTYPE fronttier PUBLIC "-/Fronttier/xml.dtd"
    "http://fronttier.alanrodas.com/dtd/xml.dtd">
<fronttier>
  <repositories>
    <repository>
      <url>http://myrepository.com</url>
    </repository>
    <repository>
      <type>git</type>
      <url>https://some-other-repo.org/repository</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <group>org.jquery</group>
      <name>jquery</name>
      <version>2.11.0</version>
    </dependency>
    <dependency>
      <group>org.jquery</group>
      <name>jquery-ui</name>
    </dependency>
  </dependencies>
</fronttier>
```

Código 4: Ejemplo del formato *xml*

como por ejemplo nuevos formatos de *archivos de configuración*.
El código de *Fronttier* es compatible por defecto con *Scala*, por estar hecho en dicho lenguaje. También se proveen bindings para *Java* que permiten utilizar todas las funcionalidades.
La mayoría de los lenguajes que corren desde la *JVM* permiten importar código *Java* y utilizarlo. Sin embargo, el código *Java* no necesariamente se adapta a las técnicas de programación que se utilizan en ese lenguaje. Por eso, nuevos bindings que permitan usar la herramienta de forma transparente en estos lenguajes pueden ser creados a futuro.
En particular, el paquete `com.alanrodas.fronttier` provee una serie de traits, objetos y clases para extender la aplicación o usar sus características. A su vez, el paquete `com.alanrodas.fronttier.java` provee los bindings

```
<!ELEMENT fronttier (repositories?, dependencies)>
<!ELEMENT repositories (repository*)>
<!ELEMENT repository EMPTY>
<!ATTLIST repository type CDATA #IMPLIED>
<!ATTLIST repository url CDATA #REQUIRED>
<!ELEMENT dependencies (dependency*)>
<!ELEMENT dependency EMPTY>
<!ATTLIST dependency group CDATA #REQUIRED>
<!ATTLIST dependency name CDATA #REQUIRED>
<!ATTLIST dependency ver CDATA #IMPLIED>
```

Código 5: DTD del formato *attrxml*

```
<?xml version="1.0"?>
<!DOCTYPE fronttier PUBLIC "-/Fronttier/xmlattr.dtd"
    "http://fronttier.alanrodas.com/dtd/xmlattr.dtd">
<fronttier>
  <repositories>
    <repository url="http://myrepository.com"/>
    <repository url="https://some-other-repo.org/repository"/>
  </repositories>
  <dependencies>
    <dependency group="org.jquery" name="jquery" ver="2.11.0"/>
    <dependency group="org.jquery" name="jquery-ui"/>
  </dependencies>
</fronttier>
```

Código 6: Ejemplo del formato *attrxml*

correspondientes para realizar las mismas acciones desde *Java*. La documentación completa se distribuirá oportunamente junto con la aplicación y podrá ir variando con el tiempo en base a nuevas versiones del programa.

4. Desarrollo de *Fronttier*

El sistema está compuesto por dos módulos independientes, *Scaliapp* y *ScalaVCS*, y por el código principal de *Fronttier*. *Scaliapp* se centra en permitir desarrollar aplicaciones que corren desde la *línea de comandos* de forma rá-

pida y sencilla, mientras que *ScalaVCS* brinda una capa de abstracción a los repositorios de control de versiones. Ambas bibliotecas son utilizadas desde *Fronttier*, pero pueden también ser utilizadas en otros proyectos.
El código desarrollado es lo más simple posible para una mayor escalabilidad y mejor mantenimiento a futuro. Todas las características avanzadas de *Scala* fueron usadas en la implementación.
El código y la documentación detallada para cada uno de los objetos, clases y métodos se encuentra disponible para su análisis en el sitio web de *Fronttier* (*http://fronttier.alanrodas.com*).

4.1. Patrones de diseño y desarrollo en *Scala*

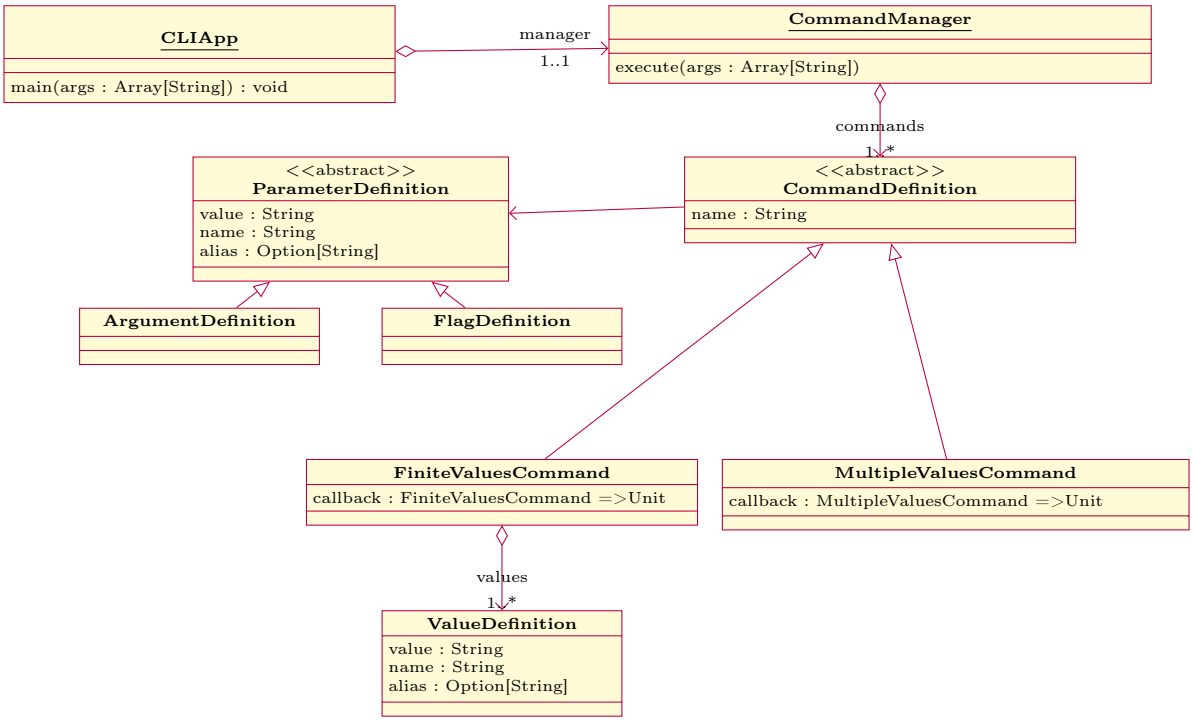
Al desarrollar sistemas se utilizan generalmente una serie de soluciones reutilizables para problemas comunes, las cuales se denominan *patrones de diseño*. En algunos casos, los *patrones de diseño* son una señal de una característica faltante en el lenguaje de programación usado. El diseño del lenguaje *Scala* soluciona por sí mismo muchos de los patrones clásicos, tanto de la *Programación Orientada a Objetos* [Gamma y col., 1995], como de la *Programación Funcional* [Yorgey, 2009].
Así, el patrón *singleton* se utilizó en varias oportunidades mediante la creación de objetos con ***object*** y *factory methods* fueron utilizados mediante el uso de ***apply*** como constructores. Las conversiones implícitas en el lenguaje permitieron implementar el patrón *adapter*, mientras que utilizar el patrón *value object* fue trivial gracias al uso de ***case classes***. Los patrones *command*, *strategy* y *chain of responsibility* se lograron mediante la posibilidad de usar funciones como elementos de primer nivel que pueden pasarse por parámetro, o la posibilidad de tener funciones parciales. El patrón *null object* se logró mediante el uso de ***Option***, aunque la funcionalidad no es exactamente igual al patrón tradicional. Por último, el patrón *dependency injection* se logra mediante el uso de ***traits*** y la capacidad de asociarlos a un objeto particular en lugar de a una clase, algo conocido como *cake pattern* [Bonér, 2008].
En cuanto a los patrones de diseño de lenguajes funcionales, *Scala* provee *functores* y *monadas* en clases como ***Option*** y ***Either*** o ***Error***. Estas clases fueron ampliamente utilizadas en el código de *Fronttier*, de *ScalaVCS* y *Scaliapp*. Además, *Scala* provee *pattern matching*, *inferencia de tipos* y otras características que suelen ser comunes de lenguajes funcionales.
Finalmente, *Scala* da la posibilidad de utilizar métodos que toman un solo argumento de forma infija o llamar métodos sin utilizar paréntesis. Ésto, junto con las caracteristicas de ***package objects*** y el patrón *builder*, permite crear lenguajes de dominio específico (DSLs). Esta técnica es ampliamente

utilizada para proveer una interfaz más clara y concisa a los desarrolladores o incluso a clientes como explica Venners en *The feel of scala*, y fue usada en *Fronttier* y las bibliotecas para dar una mayor expresibilidad al código a usar.

4.2. Modelo del parser de *línea de comandos*

El módulo *Scaliapp* es una biblioteca para crear aplicaciones de *línea de comandos* y que funciona de forma completamente independiente a *Fronttier*. La biblioteca permite extender la clase ***CLIApp*** para crear aplicaciones capaces de ser ejecutadas desde la *línea de comandos* recibiendo argumentos, flags y comandos. Para ello, *Scaliapp* provee un DSL que ayuda a especificar los comandos que serán válidos para la aplicación, que argumentos podrá recibir, y qué acción se ejecutará una vez recibido dicho comando.
El código principal se encuentra en el paquete ***com.alanrodas.scaliapp***, el cual, al importarlo, provee todas las clases y funciones necesarias para utilizar el DSL. Además provee la clase ***CLIApp***. La misma debe ser extendida por el usuario en el objeto que declarará en su cuerpo los comandos válidos de la aplicación. ***CLIApp*** se encargará de ejecutar el código correspondiente al ***callback*** del comando ejecutado al invocarse la aplicación.
El UML en la figura 1 provee una vista simplificada del modelo de definiciones de *Scaliapp*. ***CLIApp*** no es más que una clase que provee las mismas funcionalidades que ***App*** en *Scala*, pero evaluando todos los comandos declarados y delegando su procesamiento en ***CommandManager***. Para cada ***CommandDefinition*** existe un ***callback***, una función que será ejecutada cuando se invoque el comando correspondiente, y que recibirá los datos que se hayan pasado al comando. Estas funciones no tienen valor de retorno y se espera que ejecuten alguna acción concreta.

Al ser una biblioteca completamente independiente del proyecto *Fronttier*, *Scaliapp* tiene sus propios objetivos, trabajos a futuro, etc. En particular, la idea general es que la herramienta sea lo suficientemente flexible como para soportar cualquier aplicación que corra desde la *línea de comandos*, incluyendo aplicaciones interactivas. La documentación puntual, así como la información de tareas a realizar y el código actualizado puede encontrarse en la web de *Scaliapp* (http://scaliapp.alanrodas.com).
Según nuestro punto de vista el trabajo realizado para *Scaliapp* no se considera terminado pues solamente se limitó al desarrollo de la parte que fuera imprescindible para *Fronttier*. Como trabajo a futuro en este ámbito, se plantea el soporte de aplicaciones interactivas, mayor formato de comandos,



UML 1: Diagrama general de *Scaliapp*.

soporte multilenguaje y uso y generación de archivos de ayuda.
Como aplicación que corre desde la *línea de comandos*, *Fronttier* hace uso de *Scaliapp* para determinar todos los comandos que serán válidos para la aplicación.

4.3. Modelo de *archivos de configuración*

Los *archivos de configuración* de *Fronttier* se leen en base a una serie de objetos registrados en el sistema. Cada uno de dichos objetos es instancia de una clase que extiende el trait ***ConfigParser*** y reimplementa el método ***parse***, así como el valor ***id***, entre otras propiedades.
Todas las clases que implementan el trait ***ConfigParser*** son identificadas automáticamente utilizando *reflection* al comenzar la ejecución del programa. Para cada clase, una nueva instancia de la misma es creada y registrada en el sistema como encargada de analizar el *archivo de configuración* que le compete.
Fronttier permite cargar paquetes JAR creados por terceros. Esto posibilita cargar nuevos *parsers* que son luego registrados automáticamente. Así, los

desarrolladores que quieran extender la herramienta de esta forma solamente requerirán crear una clase que implemente ***ConfigParser***.
A lo largo de *Fronttier*, las instancias creadas al analizar los *parsers* son pasadas en un parámetro implícito. Además, estas instancias son creadas en forma *lazy*, de forma tal que se creen solamente objetos que van a ser efectivamente utilizados, mejorando la performance de la aplicación.

4.4. Modelo de sistema de descargas

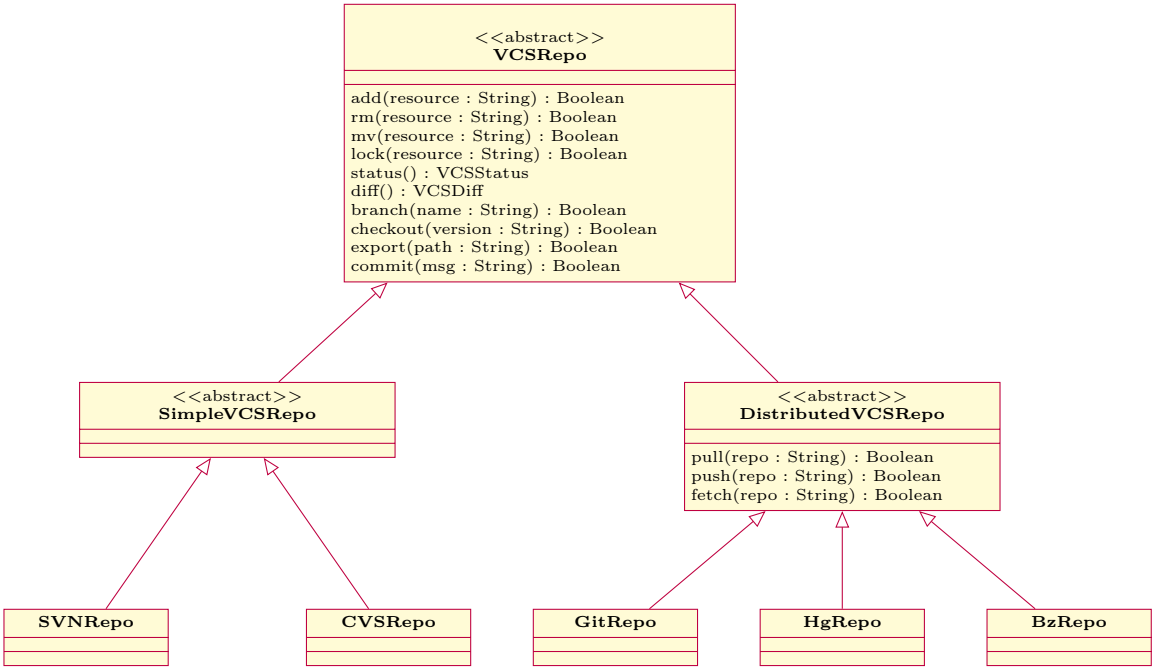
El modelo del sistema de descargas cuenta con dos partes: la parte de descargas mediante *HTTP* y que se encuentra dentro del modulo de *Fronttier*; y la parte de descargas desde sistemas de control de versiones, el cual hace uso del modulo *ScalaVCS*.
Para descargar archivos mediante *HTTP*, *Fronttier* utiliza la biblioteca ***RaptureIO***, la cual provee una base de abstracción sobre el proceso de entrada y salida en *Scala*. De esta forma, se puede descargar un archivo de *Internet* y guardarlo en el disco duro como si el mismo estuviera en la máquina local, sin preocuparse por particularidades del sistema operativo o el sistema de archivos.
Por otro lado, *Fronttier* utiliza *ScalaVCS* para descargar archivos desde sistemas de control de versiones (VCSs) como *GIT*, *SVN*, *CVS*,*Bazaar* y *Mercurial*.
ScalaVCS provee una capa de abstracción sobre los distintos sistemas de control de versiones, partiendo de la base que existen una serie de acciones que el usuario puede desear realizar sobre un VCSs. Tales operaciones incluyen añadir un archivo al sistema de control, remover un archivo, listar los archivos agregados, salvar la versión actual, volver a una versión anterior, ver las versiones salvadas o comparar dos versiones salvadas, entre otras.
Así, *ScalaVCS* provee clases que representan a cada uno de los posibles sistemas de control de versiones, las cuales poseen métodos polimórficos para dichas operaciones. Todo esto se acompaña de un DSL que permite ejecutar código como el mostrado en el ejemplo 7. El mismo asemeja en gran medida los comandos que este tipo de herramientas proveen desde la *línea de comandos*.

A su vez, *ScalaVCS* distingue dos tipos de sistemas de control de versiones: aquellos que funcionan con un solo servidor (*CVS*, *SVN*) y los distribuidos (*GIT*, *Mercurial*, *Bazaar*). Los sistemas distribuidos tienen acciones adicionales, como sincronizar de forma distribuida, o bajar una versión específica. El diagrama UML 2 muestra la estructura básica de los repositorios de *Sca-*

```
git = GIT clone "http://myrepo/mycoolproject"
git rm "someFile"
git add "*"
git commit "Some message"
```

Código 7: Ejemplo del formato *fronttier*

laVCS. Existe, a su vez, un objeto para cada tipo de sistema de control de versiones que actúa a modo de *factory*.



UML 2: Diagrama general de los repositorios de *ScalaVCS*.

ScalaVCS utiliza parámetros implícitos en todos los objetos que actúan de *factory*. Estos parámetros pueden ser importados desde el paquete ***com.alanrodas.scalavcs.connectors*** y sus subpaquetes, pasándose automáticamente a los objetos constructores. Tienen por objetivo proveer el *conector* que se utilizará como cliente para un determinado sistema de VCS. Así, es posible usar un conector que utilice la aplicación *SVN* mediante llamadas a la misma a través del sistema operativo, o utilizar una implementación desarrollada completamente en *Java*, como *SVNKit*.
ScalaVCS también es un proyecto independiente de *Fronttier*. El desarrollo

realizado en esta herramienta es mínimo, por lo cual, no se ha lanzado ninguna versión al momento de publicar el presente documento, quedando como trabajo a futuro terminar la implementación para todos los VCSs. Puede seguirse el desarrollo y planes a futuro del proyecto en el sitio web de *ScalaVCS* (http://scalavcs.alanrodas.com).

4.5. Modelo general

Las partes anteriores se combinan finalmente en *Fronttier* dando lugar a la aplicación prácticamente completa. Las clases simples que representan los conceptos básicos de *repositorios*, *dependencias* y otros, complementan algunos patrones que cierran la aplicación. A esto, se suma al desarrollo de la aplicación el agregado de *logging*, el *testing*, y los *Scala Docs* necesarios para una correcta documentación.
Cabe mencionar que se utilizó *GIT* como sistema de control de versiones en el proyecto, *SBT* como manejador de dependencias del mismo e *IntelliJ IDEA* como IDE de desarrollo. El sitio web de la herramienta fue desarrollado utilizando *HTML5*, con *JavaScript* y *CSS*.
Por último, este documento, así como otros archivos de documentación en el sitio, fueron creados con *XeLaTeX*, *Biber* y la herramienta *arara*[13] para su compilación. La versión de *TeX* utilizada corresponde a *TeXLive 2014*, con el paquete *minted* en su versión de desarrollo, y la herramienta de *Python pygmentize* en su versión 1.6. *TeX Studio* fue usado como IDE para los documentos.

5. Conclusiones y trabajos a futuro

Dentro del contexto de los *manejadores de dependencias* para la *JVM*, *Fronttier* innova por ser la primera en encargarse de los recursos web de un sistema, así como también lo hace en contexto de los *manejadores de dependencias* para la *capa de presentación* por ser el primero pensado para la plataforma *Java*.
Fronttier permite al usuario descargar *dependencias* web de forma sencilla, desde la *línea de comandos*, y desde aplicaciones que corren sobre la *JVM*. Se ha integrado a procesos existentes de desarrollo de software para facilitar

[13] *arara* es una herramienta programada en Perl que permite simplificar el proceso de compilación de documentos de *LaTeX*, *XeLaTeX* y otros derivados de *TeX* mediante comentarios en el documento.

su adopción.
Se ha logrado construir una herramienta que es altamente extensible, y que pone al desarrollador en una posición de pocas restricciones con respecto a las extensiones posibles a la misma.
Si bien la herramienta puede no lograr convertirse en un estándar, no es difícil pensar que futuras modificaciones como, mayor cantidad de interacción con otras herramientas y la creación de repositorios libres y comunes puedan llevar a *Fronttier* a convertirse en un referente en cuanto a manejo de dependencias de la *capa de presentación* se refiere.
Durante el proceso de programación de la herramienta, se ha podido observar como el desarrollo de módulos pensados como una API son fundamentales para una fácil y rápida extensión del sistema.

Como se observa en la sección 2.4, el presente trabajo tiene un alcance limitado con respecto a los deseos expresados en la sección 2.2. En particular, sería interesante desarrollar las siguientes extensiones:

- El soporte de otras formas de descarga, por ejemplo, con otros sistemas de control de versiones como *Mercurial*, *Bazaar*, *CVS*, etc.
- El uso de una mayor cantidad de formatos de *archivos de configuración*
- La seguridad en los repositorios mediante autenticación
- El funcionamiento desde otros lenguajes de la *JVM* como *Clojure* y *Groovy*
- La integración con más cantidad de procesos, como *Grape*, *Gradle*, etc.
- La integración con mayor cantidad de *frameworks* web
- La integración con interfaces de desarrollo integradas (IDEs)

También resulta interesante como extensión la generación de un sitio que funcione como repositorio de *dependencias* centralizado para que se constituya como la referencia en cuanto a paquetes para usar en la *capa de presentación*. El sitio debería permitir a los desarrolladores subir y manejar ellos mismos los paquetes que desean distribuir en el repositorio. Este es realmente uno de los trabajos pendientes más importantes para lograr una buena aceptación de la herramienta y posicionarla como un estándar en la industria.
El hecho de que exista una enorme cantidad de lenguajes para la *JVM*, y una enorme cantidad de *frameworks* especializados en el desarrollo de aplicaciones web que corren en los mismos, hace que sea extremadamente sencillo encontrar constantemente nuevas posibilidades a implementar. La adaptación a más lenguajes de forma *"nativa"*, y la integración con más procesos y

frameworks es una de las formas de expansión de este trabajo más deseables. Finalmente, si bien la plataforma elegida fue la *JVM*, existen numerosos lenguajes que no corren sobre ésta y que se enfocan también en el desarrollo de sistemas web, como la plataforma *.NET* o *PHP*. Se podrían entonces crear adaptaciones (*ports*) a dichos lenguajes y plataformas como una opción de trabajo a futuro.
Por último, es de suma importancia remarcar que la herramienta es liberada como *Software Líbre* haciendo que el trabajo pendiente expresado en esta sección pueda ser realizado por la comunidad. Incluso es posible que existan desarrollos que expandan el sistema en formas que no han sido planteadas en esta sección.

Referencias

[1] Alvin Alexander. *Scala Cookbook*. Sebastopol, CA, United States of America: O'Reilly Media, ago. de 2013 (vid. pág. 46).

[2] Roy A. Allan. *A history of the personal computer: the people and the technology*. London, Ontario, Canada: Allan Publishing, 2001. URL: http://www.retrocomputing.net/info/allan/ (vid. pág. 43).

[3] Greg Baarish. *Building Scalable and High-performance Java Web Applications using J2EE Technology*. Indianapolis, Indiana: Pearson Education, 2002 (vid. pág. 8).

[4] Jonas Bonér. *Real-World Scala: Dependency Injection (DI)*. 2008. URL: http://jonasboner.com/2008/10/06/real-world-scala-dependency-injection-di/ (vid. pág. 28).

[5] Pierre Carbonelle. *PYPL PopularitY of Programming Language index*. Jun. de 2014. URL: https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language (vid. pág. 41).

[6] CDN.js. *CDN.js: The missing CDN for JavaScript and CSS*. URL: http://cdnjs.com/ (vid. pág. 49).

[7] Matthew David. *HTML5: Designing Rich Internet Applications*. New York; London: Focal Press, 2013 (vid. pág. 45).

[8] Doris G. Duncan y Sateesh B. Lele. "Converting From Mainframe to Client/Server at Telogy Inc." En: *Journal of Software: Evolution and Process* 8 (5 1996). DOI: 10.1002/(sici)1096-908x(199609)8:5<321::aid-smr135>3.0.co;2-4. URL: http://libgen.org/scimag/index.php?s=10.1002/(sici)1096-908x(199609)8:5%3C321::aid-smr135%3E3.0.co;2-4 (vid. págs. 43, 44).

[9] Margaret S. Elliott y Kenneth L. Kraemer. *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*. ASIS&T monograph series. Information Today Inc. 143 Old Merlton Pike, Medford, New Jersey: American Society for Information Science y Technology, 2008 (vid. págs. 43, 45).

[10] Greg Franko. *Dependency Management with RequireJS How-to*. 35 Livery Street, Birmingham, UK: Packt Publishing, mayo de 2013 (vid. pág. 50).

[11] Erich Gamma y col. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (vid. pág. 28).

[12] Google. *Google Hosted Libraries*. URL: https://developers.google.com/speed/libraries/ (vid. pág. 48).

[13] Jason Hunter y William Crawford. *Java Servlet Programming*. The Java Series. Sebastopol, CA, United States of America: O'Reilly Media, abr. de 2001 (vid. pág. 44).

[14] jsDelivr. *jsDelivr: A free super-fast CDN for developers and webmasters*. URL: http://www.jsdelivr.com/ (vid. pág. 49).

[15] Gerber Kunst. *Programming Language Popularity Chart*. Jun. de 2014. URL: http://langpop.corger.nl/ (vid. pág. 41).

[16] Craig Larman y Bas Vodde. *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*. The Java Series. Boston, MA, United States of America: Pearson Education, Inc, ene. de 2010 (vid. pág. 9).

[17] Microsoft. *Microsoft Ajax Content Delivery Network*. URL: http://www.asp.net/ajaxlibrary/cdn.ashx (vid. pág. 49).

[18] Gautier de Montmollin. *The Transparent Language Popularity Index*. Jul. de 2013. URL: http://lang-index.sourceforge.net/ (vid. pág. 41).

[19] TIOBE Software. *TIOBE Index for June 2014*. Jun. de 2014. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (vid. pág. 41).

[20] Sonatype. *Maven: The Definitive Guide*. Sebastopol, CA, United States of America: O'Reilly Media, ago. de 2008 (vid. pág. 46).

[21] David Stephens. *What On Earth is a Mainframe?* Longpela Expertise, oct. de 2008 (vid. págs. 42, 43).

[22] Bower by Twitter. *Bower: A package manager for the web*. 2014. URL: http://bower.io/ (vid. pág. 50).

[23] Bill Venners. *The feel of scala*. 2009. URL: https://www.parleys.com/play/514892250364bc17fc56b9a5/chapter0/about (vid. pág. 29).

[24] Wikipedia. *List of Java Virtual Machines*. Abr. de 2012. URL: https://en.wikipedia.org/wiki/List_of_Java_virtual_machines (vid. pág. 41).

[25] Wikipedia. *List of JVM Languages*. Sep. de 2014. URL: http://en.wikipedia.org/wiki/List_of_JVM_languages (vid. pág. 40).

[26] Brent Yorgey. *Typeclassopedia*. 2009. URL: https://www.haskell.org/haskellwiki/Typeclassopedia (vid. pág. 28).

Anexos

A. JVM como plataforma: Popularidad de la JVM

En esta sección se explican brevemente los factores que motivan la elección de la *JVM* como la plataforma de trabajo. En particular son dos los motivos por los cuales se ha elegido esta plataforma, su popularidad y la gran cantidad de lenguajes que corren sobre la misma.

A.1. Acerca de los lenguajes para la Java Virtual Machine (*JVM*)

Con la popularidad de *Java*, comenzaron a surgir una serie de lenguajes que compilan a *bytecode* para la Maquina Virtual de Java. Estos lenguajes se presentan como alternativas para los programadores de la plataforma que buscan una opción "superior", en algún aspecto, a *Java*. *Groovy*, por ejemplo, es un lenguaje dinámico, similar a Ruby; *Clojure* es una extensión de Lisp apuntado a la Programación Concurrente; *Scala* es un lenguaje que posee características tanto de Programación Funcional como de Programación Orientada a Objectos. También existe otra numerosa cantidad de lenguajes, algunos experimentales, otros con una base creciente de usuarios, que corren sobre la plataforma. A esto se suma una amplia cantidad de adaptaciones de lenguajes populares para que corran en la *JVM*, como JRuby y Jython, versiones de Ruby y Python que compilan a *bytecode Java* [Wikipedia, 2014]. Una característica interesante de estos lenguajes es que el código compilado suele ser compatible entre si, es decir, código escrito en *Java* puede ser ejecutado en *Scala*, código *Groovy* puede ser usado desde *Clojure*, etc.[14].
Finalmente, se destaca el hecho de que hoy en día existen numerosas implementaciones de la *JVM*, capaces de correr dichos lenguajes en distintas

[14] Si bien mayoritariamente el código de un lenguaje para la *JVM* es compatible con otros lenguajes de la *JVM*, ciertas características especiales de estos suelen no poder ser empleadas desde otros lenguajes, presentando así ciertas limitaciones de compatibilidad.

plataformas [Wikipedia, 2012].

A.2. Popularidad de la *JVM* como plataforma

Las estadísticas de análisis de popularidad de lenguajes como la conocida TIOBE demuestran que *Java* es uno de los lenguajes más populares [TIOBE Software, 2014]. Por su parte, el informe *"transparente"* creado por Gautier de Montmollin, que publica el código de forma *Open Source* y sus reglas online, exhibe al lenguaje de Oracle en la misma posición [Montmollin, 2013], mientras que la otra herramienta *Open Source* que permite medir la popularidad de los lenguajes, PyPL, de Pierre Carbonelle, lo presenta en primer lugar [Carbonelle, 2014].
Además, Gerber Kunst ha desarrollado un gráfico en donde proyecta la cantidad de líneas de código en *GitHub* por sobre las menciones en *Stack Overflow*[15], mostrando a *Java* por sobre los más populares, pero también a los lenguajes que corren sobre la *JVM*[16] como de alto interés.

[15] StackOverflow es un popular sitio sobre programación, en donde usuarios suelen realizar preguntas a problemas puntuales de programación y otros usuarios las responden.

[16] Entre los lenguajes para la *JVM* se encuentran *Scala*, *Clojure*, *Groovy*, *AspectJ*, *Ceylon*, *Fantom*, *Fortress*, *Frege*, *Gosu*, *Ioke*, *Jelly*, *Kotlin*, *Mirah*, *Processing*, *X10*, *Xtend*, *Rhino*, *JRuby*, *Jython*, entre otros. Aunque no todos se encuentran mencionados en la gráfica.

B. Motivación de enfocarse en los sistemas web: Historia de los sistemas empresariales

Este apartado dará una breve muestra de las transformaciones que ha sufrido la industria del desarrollo de sistemas en los últimos tiempos. Se verá también cómo estas transformaciones han llevado a que los sistemas web sean hoy en día una de las opciones más utilizadas en el ámbito empresarial.

B.1. Estado anterior a los sistemas web

En las décadas de 1960 y 1970, comienza en el mundo un proceso lento pero incremental de computarización de la información. Las computadoras dejan de ser *"juguetes"* científicos para pasar a ser complejas maquinarias con verdadera utilidad práctica en distintas industrias.
Durante esos años varias empresas comienzan a *informatizarse*. Corporaciones compran grandes computadoras capaces de almacenar toda su información o procesar complejos cálculos matemáticos en poco tiempo.
En este contexto, múltiples empresas como *IBM*, *RCA*, y *General Electric*, comenzaron a fabricar gigantescas computadoras cada vez más económicas. Estas maquinas evolucionarían poco a poco hasta convertirse en lo que hoy se conocen como *mainframes*.
En este sentido, David Stephens en su libro *What On Earth is a Mainframe?* [2008] describe a un *mainframe* como una computadora muy grande, con una base de datos de alto rendimiento y a la cual se accede desde una terminal remota (una máquina *"tonta"*, sin mucha otra finalidad más que la de conectarse al servidor central para realizar peticiones de datos y mostrarle los resultados al usuario).
Stephens también destaca los beneficios de los *mainframes* por sobre una computadora personal a nivel empresarial. Las características que menciona son:

- **Integridad de datos:** Los datos DEBEN ser correctos.
- **Rendimiento:** Se debe procesar gran cantidad de datos.
- **Respuesta:** El procesamiento debe ser inmediato.
- **Recuperación ante desastres:** En caso de un fallo, se debe volver a estar operativo inmediatamente.
- **Usabilidad:** Debe hacer lo que se requiere, cuando se requiere.
- **Confiabilidad:** No debe fallar y siempre debe estar disponible.

- **Auditoria:** Ser capaz de saber quien realizó qué acción en el equipo.
- **Seguridad:** Solo aquellos que pueden realizar una acción pueden hacerlo.

Estas características son todavía buscadas en los grandes *mainframes* de la actualidad, y resultan fundamentales en emprendimientos críticos.
Margaret S. Elliott y Kenneth L. Kraemer han descripto que las tendencias de la tecnología utilizada no obedecen solamente a motivaciones económicas y operativas, sino a un complejo entramado de relaciones entre empresas que comparten una visión utópica sobre la tecnología en cuestión [Elliott y Kraemer, 2008, pag 3]. Así, las empresas comercializadoras de *mainframes* enarbolando como bandera principal los beneficios planteados por Stephens, lograron generar una visión utópica en las empresas usuarias para posicionar la tecnología durante largos años. Es por esto que los *mainframes* siguen teniendo hoy en día un lugar en el mercado.

B.2. Aparición de las computadoras personales

A partir de fines de los 70's y comienzos de los 80's, las *PCs* comienzan a ser cada vez más accesibles [Allan, 2001, cap 4]. La liberación de *ARPANET* por parte de *DARPA*[17] en 1983 da nacimiento a *Internet*. Estos dos cambios suponen un quiebre en las tecnologías de uso empresarial.
El *PC*, sumado a la conectividad y nuevas tecnologías tanto en sistemas como en lenguajes de programación llevaron a que los *mainframes* comenzaran a perder terreno.
La posibilidad de ejecutar programas en cada *PC* dio lugar a nuevas arquitecturas *cliente-servidor*. Ahora los usuarios corrían los programas en su propio equipo, permitiendo aumentar la velocidad de respuesta. El programa se conectaba a *Internet* y sincronizaba información con un servidor central solo cuando fuera necesario.
Esta arquitectura permitiría una más rápida evolución del software por sobre la opción de tener el sistema en un *mainframe*, como explican Doris G. Duncan y Sateesh B. Lele en su artículo "Converting From Mainframe to Client/Server at Telogy Inc.". Como contrapartida estos cambios suponen un alto costo de mantenimiento, y en caso de poseer equipos con distintos sistemas operativos, un gasto extra en el desarrollo. Independientemente de

[17] Agencia de Proyectos de Investigación Avanzados de Defensa. En está agencia se concibió el concepto de *Internet*.

los costos, el factor más incidente en el cambio es la necesidad de mantener el sistema al día con las necesidades de la empresa, señalan Duncan y Lele.

B.3. Navegadores y primeros sistemas web

Tras la aparición de los navegadores web (como *Netscape* e *Internet Explorer*) a comienzos de 1990 se comenzaron a desarrollar los primeros sistemas web.
Los mismos consistían básicamente en documentos dinámicos, generados con información tomada de una base de datos. Gran influencia tuvieron en este tipo de soluciones la creación de lenguajes de programación y tecnologías pensadas para generar documentos *HTML* dinámicamente, como *PHP*, *ASP* y los *Servlets* de *Java* [Hunter y Crawford, 2001, pag 2]. La posibilidad de generar *aplicaciones* que corran en el navegador solucionó las problemáticas asociadas a el mantenimiento de los sistemas y al desarrollo para múltiples sistemas operativos. Sin embargo, la naturaleza de *HTML* limitaba la capacidad de las aplicaciones a sencillos formularios, botones y enlaces.
Estas limitaciones hicieron que no fuera sino hasta con la aparición de las primeras tecnologías complementarias a *HTML* que las aplicaciones web comenzarían a cobrar relevancia, dando lugar a las *RIAs*.

B.4. Rich Internet Applications

Las Aplicaciones Ricas de Internet (*RIAs*) surgen para compensar aquellas faltantes que presentaban las aplicaciones web frente a las tradicionales de *cliente-servidor*.
Una *RIA* es una aplicación que corre en el navegador del usuario y que permite emular complejos comportamientos y funcionalidades que antes solo eran posibles en sistemas de escritorio, tales como *"arrastrar y soltar"*, ordenar elementos o crear complejos gráficos.
En un primer lugar software en forma de complementos (*plugins*) que permitían correr algún lenguaje especial en los navegadores fueron creados. De esta forma, se obtenía una mayor funcionalidad en la *capa de presentación* que usando sólo *HTML*.
Las *RIAs* coparon los mercados, ya que permitían ahorrar en mantenimiento y desarrollo a la vez que brindaban la usabilidad que se demandaba de un sistema empresarial.
Surgieron cada vez más soluciones y los *plugins* privativos comenzaron a ser un problema para los desarrolladores. Esto llevaría a que se fomentaran con

afán *CSS*[18] y *JavaScript*[19], tecnologías libres que permitían crear *RIAs*, disponibles en cualquier navegador sin necesidad de instalar software adicional. Así, en los últimos años, y gracias también al auge de los smartphones y tablets incapaces de correr los anteriormente mencionados *plugins*, el llamado *HTML5* pasaría a transformarse en el estándar para el desarrollo de *RIAs* [David, 2013].

B.5. Conclusiones

Los sistemas web actuales permiten la robustez esperada de los primeros sistemas *mainframe*, pero sin los altos costos de adquisición y mantenimiento. A su vez, las *RIAs* ofrecen la capacidad de tener aplicaciones con un alto grado de usabilidad. Además, la gran cantidad de *frameworks* capaces de generar rápidamente sistemas web con una interfaz completamente realizada en *HTML5* que han aparecido en los últimos años, permiten reducir los tiempos de desarrollo. Junto con las plataformas como servicio, como *Heroku* o *Rackspace*[20] reducen significativamente la puesta en producción de los sistemas.
Todas estas características han llevado a que las aplicaciones web con una *capa de presentación* hecha en *HTML5* se hayan transformado en la nueva visión utópica señalada por Elliott y Kraemer. Así, este tipo de desarrollos seguirán siendo la norma durante los próximos años.

[18] CSS es un lenguaje utilizado para describir el aspecto de un documento escrito en lenguaje HTML o similar. Actualmente es un estandar web y es soportado por prácticamente todos los navegadores.

[19] JavaScript (JS) es un lenguaje de programación interpretado, orientado a objetos y basado en prototipos, utilizado principalmente del lado del cliente. Este lenguaje es un estándar web ya que es implementado como parte de prácticamente todos los navegadores web, permitiendo interactividad del usuario con la interfaz sin necesidad de comunicarse constantemente con el servidor.

[20] Rackspace y Heroku son servicios de hosting para aplicaciones pensados para escalar en capacidad en poco tiempo.

C. Estado del arte: Manejo de distintos tipos de *dependencias*

En esta sección se explicará como se manejan las *dependencias* en las distintas capas de software. Se mostrará como la solución propuesta en el presente documento se adapta a técnicas ya probadas.

C.1. Dependencias manejadas de la *capa de lógica de negocios*

En el ámbito del manejo de *dependencias* en la *capa de lógica de negocios* la herramienta *Maven* de la fundación *Apache* se ha posicionado como el estándar de facto[21] [Sonatype, 2008]. Otro de los proyectos que se han posicionado en esta área es *Apache Ivy*, que se concentra solamente en el manejo de *dependencias*.
Maven hace uso de un *archivo de configuración* escrito en *XML*, comúnmente denominado POM, ya que su nombre es pom.xml. El código 8 muestra un ejemplo de dicho tipo de *archivo de configuración*. *Ivy* utiliza también archivos *XML*, pero estos son más compactos que los de *Maven*. El código 9 es un ejemplo del mismo.

Con la aparición de nuevos lenguajes para la *JVM* comenzaron a surgir nuevos *manejadores de dependencias*, como Grape o Gradle para *Groovy*, *SBT* para *Scala*, Leiningen para *Clojure*, y *Apache* Buildr como propuesta multilenguaje. Cada uno de ellos utiliza un *archivo de configuración* con distinta sintaxis [Alexander, 2013, pág. 569]. El código 10 muestra un ejemplo de la declaración de una *dependencia* en el *archivo de configuración* de *SBT*.

A pesar de la gran cantidad de formatos de *archivo de configuración*, todos los *manejadores de dependencias* trabajan bajo el estándar pre-impuesto por *Maven*. Esto llega a un punto tal que el sitio *MVNRepo*[22] es el centro de referencia al momento de buscar dependencias. Se puede entonces apreciar

[21] Si bien *Apache Maven* es en realidad no solo un *manejador de dependencias* sino un administrador de proyectos, provee la funcionalidad antes mencionada y la misma se ha vuelto un estándar.

[22] MVNRepo o Maven Repository es un sitio web alojado en *http://mavenrepository.com* que permite buscar una amplia cantidad de paquetes *.jar* y *.war* que podemos importar

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-a</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
</project>
```

Código 8: Configuración de *dependencias* en *Maven* mediante archivo POM

```
<ivy-module version="2.0">
  <info organisation="org.apache" module="hello-ivy"/>
  <dependencies>
    <dependency org="group-a" name="artifact-a" rev="1.0"/>
    <dependency org="group-b" name="artifact-b" rev="1.0"/>
  </dependencies>
</ivy-module>
```

Código 9: Archivo de configuración de Ivy

como todos los *archivos de configuración* especifican inequívocamente un paquete de código mediante tres campos clave para su búsqueda: el nombre de la compañía, el nombre del módulo y la versión del programa.

C.2. Dependencias no manejadas de la *capa de presentación*

La mayoría de las *dependencias* de la *capa de presentación* consisten en archi-

en nuestro proyecto, brindando el código necesario para agregarlo al mismo utilizando distintos *manejadores de dependencias*.

```
libraryDependencies += "org.springframework.data" %
    "spring-data-neo4j" % "3.1.1.RELEASE"
```

Código 10: Dependencia de SpringFramework 3.1.1 para *SBT*

vos principalmente *CSS* y *JavaScript* (aunque también podrían ser fuentes web u otros recursos), los cuales son descargados manualmente a través de la página del creador del código (no existen repositorios centralizados como en el caso de las *dependencias* de la *capa de lógica de negocios*). Una vez descargados, el desarrollador debe mover los archivos a una carpeta de su proyecto para *instalarlos* de forma que se encuentren disponibles para su uso en el código (la ubicación exacta depende de la estructura del proyecto y por tanto debe ser recordada por el programador).
Los errores en esta etapa son muy comunes dado la variada cantidad de tipos de *dependencia*, donde cada uno debe ser copiado en una ubicación puntual. Además, una misma *dependencia* no necesariamente consiste en un único archivo (como sucede en la *capa de lógica de negocios*), sino en múltiples archivos de distinto formato que deben ser agregados al proyecto de una forma específica. La falta de un lugar que agrupe éstos contenidos y estandarice sus nombres, versiones, formas de instalación, etc. hace que sea muy difícil manejar éste tipo de *dependencias*.

C.3. Redes de distribución de contenido

Las *Redes de Distribución de Contenido*, (*CDN*) consisten en una serie de servidores en *Internet* distribuidos en distintos puntos geográficos. Los usuarios solicitan contenido a éstos servidores y es siempre el más próximo a la ubicación del usuario el que entrega el mismo, disminuyendo los tiempos de carga.
Así, las *CDNs* permiten agrupar el contenido disponible online en un solo lugar[23], permitiendo al usuario insertar en el código *HTML* de su proyecto una cláusula que importa automáticamente las dependencias desde el servidor de la *CDN* más cercano.

El código 11 utiliza la *CDN* de Google (`https://developers.google.com/speed/libraries/`) una de las más populares y confiables junto con la

[23] Su creación no responde necesariamente a procesos de manejo de dependencias en proyectos de desarrollo de software, sin embargo es uno de los usos posibles de este tipo de redes y ha sido uno de los usos más comunes.

```
<html>
  <head>
    ...
    <!-- JQuery -->
    <script src="//ajax.googleapis.com/
        ajax/libs/jquery/1.11.1/jquery.min.js">
    </script>
    <!-- JQuery Mobile -->
    <link rel="stylesheet" href="//ajax.googleapis.com/ajax/libs/
        jquerymobile/1.4.3/jquery.mobile.min.css"/>
    <script src="//ajax.googleapis.com/ajax/libs/
        jquerymobile/1.4.3/jquery.mobile.min.js">
    </script>
    <!-- JQuery UI -->
    <link rel="stylesheet" href="//ajax.googleapis.com/ajax/libs/
        jqueryui/1.11.0/themes/smoothness/jquery-ui.css"/>
    <script src="//ajax.googleapis.com/ajax/libs/
        jqueryui/1.11.0/jquery-ui.min.js">
    </script>
  </head>
  <body>
    ...
  </body>
</html>
```

Código 11: Dependencias agregadas mediante *CDN*

de Microsoft (`http://www.asp.net/ajaxlibrary/cdn.ashx`) y los sitios `http://www.jsdelivr.com/` y `http://cdnjs.com/`.
Las *CDNs* permiten adicionalmente aumentar la velocidad de la aplicación que se desarrolla ya que los archivos de *dependencia* pueden ser descargados desde servidores más cercanos a la ubicación del usuario, y además ahorran carga de trabajo al servidor de la aplicación.
Como contrapartida, las *CDNs* suelen tener una gestión centralizada desde alguna entidad proveedora, la cual no necesariamente responde a los pedidos o requerimientos del usuario. Ésto hace que sea difícil contar con las últimas versiones, o con bibliotecas que los administradores del mismo decidan no soportar. A esto se le agrega el alto costo de mantener una *CDN* por cualquier empresa que no posea grandes servidores a lo largo del globo.

C.4. Dependencias manejadas de la *capa de presentación*

Otra forma de manejar *dependencias* en la *capa de presentación* consiste en utilizar el mismo mecanismo de *manejadores de dependencias* que se utiliza en la *capa de lógica de negocios*. Esto pone solución a las problemáticas que acarrean las *CDNs*.
Bower y *Component* son los *manejadores de dependencias* más populares para la *capa de presentación*. Ambos son módulos de la plataforma *Node.js*[24] y manejan las *dependencias* mediante un *archivo de configuración*.
Estas herramientas no se empaquetan de forma auto contenida, sino que requieren contar con herramientas externas como *GIT* o *SVN* en la máquina en donde se va a utilizar [Twitter, 2014]. Más aún, los usuarios de lenguajes que corren sobre la plataforma *Java* muchas veces carecen de acceso a la máquina física, por fuera de la *JVM*, haciendo imposible utilizar estas opciones.
Adicionalmente, existen soluciones que se enfocan en un tipo especifico de dependencia, como *RequireJS* y *Browserify* que permiten manejar solamente archivos *JavaScript*, sin utilizar *archivos de configuración*, sino mediante una línea de código en el programa del usuario [Franko, 2013].
Todas éstas soluciones se presentan bastante incompletas en comparación a las disponibles en la *capa de presentación*. Además, al requerir herramientas que no son independientes del sistema operativo, generan muchas complicaciones al momento de configurar el entorno de desarrollo o de puesta en producción del sistema. Finalmente, fuerzan al programador a aprender una nueva tecnología, proceso que lleva tiempo y esfuerzo.

C.5. Funcionamiento de *Fronttier*

La herramienta propuesta funciona mediante un *archivo de configuración* de forma similar a las opciones disponibles para la *capa de lógica de negocios*. Además, los *archivo de configuración* que utiliza, pueden poseer formatos distintos, adaptándose a los que los desarrolladores ya conocen de otras herramientas.
A diferencia de las opciones de manejo de *dependencias* que funcionan en la *capa de presentación*, *Fronttier* no requiere herramientas externas, y todo corre sobre la *JVM*.

[24] Node.js es un entorno de programación en la capa del servidor basado en el lenguaje de programación *JavaScript*, con I/O de datos en una arquitectura orientada a eventos y basado en el motor *JavaScript* V8.