## 4. Diseño e implementación de Aether

En el siguiente capítulo se detallarán el diseño e implementación del framework Aether. Inicialmente se introducirán los requerimientos funcionales del sistema y detallará la tecnología utilizada. Posteriormente se describirán los principales conceptos de diseño como patrones y estilos arquitectónicos para luego pasar a definir la arquitectura de la herramienta desarrollada describiendo el diseño detallado.

### 4.1 Requerimientos funcionales

Para lograr la realización de este trabajo se desarrollaron una serie de requerimientos funcionales sobre los cuales se diseñó e implementó para llegar a obtener un framework de simple utilización y extensibilidad. A continuación se detalla cada uno de estos requerimientos:

1. Desarrollar un framework que permita abstraer la aplicación del usuario de las diferentes implementaciones de los servicios en cloud que utilice.
2. Permitir la migración al framework de aplicaciones ya desarrolladas utilizando otras tecnologías.
3. Posibilitar el cambio de proveedores de los servicios utilizados de forma dinámica sin modificar el código de la aplicación desarrollada.
4. Permitir la incorporación de nuevos servicios a medida que sea necesario.
5. Posibilitar la incorporación de nuevos proveedores para cada servicio provisto.

Estos requerimientos se extrajeron del análisis realizado en el capítulo 2 de este documento.

### 4.2 Tecnología utilizada

Entre las principales herramientas utilizadas para el desarrollo del framework se encuentran Maven[[1]](#footnote-1), Javassist[[2]](#footnote-2) y Eclipse[[3]](#footnote-3). Maven es una herramienta de software utilizada para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Posee un modelo de configuración de construcción simple basado en un formato XML permitiendo, entre otras cosas, una sencilla administración de dependencias. Por su parte, Javassist (asistente de programación Java) es una biblioteca de Java que proporciona un medio para manipular el bytecode Java de una aplicación. En este sentido, Javassist proporciona el soporte para la reflexión estructural, es decir, la capacidad de cambiar la implementación de una clase en tiempo de ejecución. Por último, Eclipse es un [entorno de desarrollo integrado](http://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado) de [código abierto](http://es.wikipedia.org/wiki/C%C3%B3digo_abierto) y multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Dispone de un [Editor de texto](http://es.wikipedia.org/wiki/Editor_de_texto) con resaltado de sintaxis y la compilación la realiza en tiempo real. Permite pruebas unitarias con [JUnit](http://es.wikipedia.org/wiki/JUnit), control de versiones con [CVS](http://es.wikipedia.org/wiki/CVS), integración con [Ant](http://es.wikipedia.org/wiki/Ant), asistentes (wizards) para creación de proyectos, clases, tests, etc. Asimismo, a través de "plugins" libremente disponibles es posible añadir una gran variedad de complementos.

### 4.3 Diseño

El diseño arquitectónico es una de las actividades más importantes involucradas en el desarrollo de sistemas. Como salida de esta actividad se obtiene la arquitectura de un sistema de software. Esta arquitectura es la estructura del sistema, la cual comprende elementos de software, las propiedades de los elementos visibles externamente y las relaciones entre ellos. Además, representa un conjunto inicial de decisiones de diseño.

El concepto de patrón representa uno de los enfoques más atractivos para la reutilización de conocimiento de diseño. Se utiliza el término patrón para describir abstracciones de software que son utilizadas por diseñadores y programadores expertos en el desarrollo de aplicaciones [Beck 1994]. Los patrones existen en diferentes niveles de abstracción:

* En el nivel más alto de abstracción se encuentran los denominados patrones arquitectónicos, los cuales caracterizan la organización global de un sistema. Estos patrones definen la división de una aplicación en subsistemas y establecen como estos subsistemas colaboran. Ejemplos típicos de esta categoría de patrones son los estilos arquitectónicos denominados client-server, pipes-and-filters y de capas, entre otras [Buschmann 1996].
* En un nivel intermedio, se encuentran los denominados patrones de diseño, los cuales prescriben soluciones a problemas de diseño específicos que surgen al refinar un diseño global.
* En el nivel más bajo de abstracción en el espectro de patrones existentes, se encuentran los idiomas (idioms) que describen como materializar diseños específicos en un determinado lenguaje de programación [Coplien 1991].

Los beneficios que provee el uso de patrones en el proceso de desarrollo de software incluyen: reutilización de diseño, reutilización potencial de código, mayor comprensión de la organización global de un sistema, y mejor interoperabilidad con otros sistemas por medio de la introducción de estándares.

Idealmente, si se cuenta con un rico conjunto de patrones bien definidos, el proceso de diseño puede transformarse en un proceso iterativo que involucra la búsqueda y selección de los patrones que resuelven cada uno de los problemas de diseño que debamos enfrentar en el desarrollo de un sistema, para luego aplicar a nuestro diseño la solución prescrita por aquellos patrones seleccionados, hasta que no hayan más problemas por resolver [Beck 1994].

#### 4.3.1 Estilos arquitecturales

Los estilos arquitectónicos, o también llamados patrones arquitecturales, representan esquemas para organizar estructuralmente sistemas de software. Proveen un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen reglas y guías para organizar sus relaciones [Buschmann 1996].

Un estilo arquitectónico está determinado por [Bass 2003]:

* Un conjunto de tipos de componentes (ejemplo: repositorio de datos, proceso, procedimiento) que realizan alguna función durante la ejecución del sistema.
* Un layout de los componentes que indica sus relaciones en tiempo de ejecución.
* Un conjunto de restricciones semánticas (ejemplo: no se permite que el repositorio de datos modifique los datos almacenados en él).
* Un conjunto de conectores (ejemplo: llamada a subrutina, llamada a procedimiento remoto, sockets) que median la comunicación, coordinación y cooperación entre los componentes.

Habiendo introducido los principales conceptos relacionados al diseño arquitectónico de un sistema, en la siguiente sección se detallará la arquitectura del framework desarrollado.

#### 4.3.2 Arquitectura del framework Aether

En este apartado se detallará la arquitectura del framework desarrollado describiendo los componentes de cada módulo que lo forman.

#### 4.3.2.1 Diseño detallado de Aether

El framework se encuentra organizado en tres proyectos java independientes: aether-core, aether-adapters y aether-loader. En el primero se encuentran las clases correspondientes al núcleo del framework encargadas de abstraer los servicios de cloud, configurar las conexiones y permitir las diferentes operaciones sobre cada uno de los servicios. Aether-adapters corresponde al grupo de clases encargadas de proporcionar al usuario las facilidades para migrar a Aether una aplicación ya desarrollada utilizando algún otro framework; básicamente está formado por sub-proyectos independientes (uno para cada herramienta o framework desde el cuál se desee realizar la migración). Por último, aether-loader es el encargado de interceptar las llamadas a métodos de otros frameworks y realizar la invocación al método correspondiente del adapter indicado. En la figura 4.1 se presentan los tres módulos nombrados y la interacción entre estos.

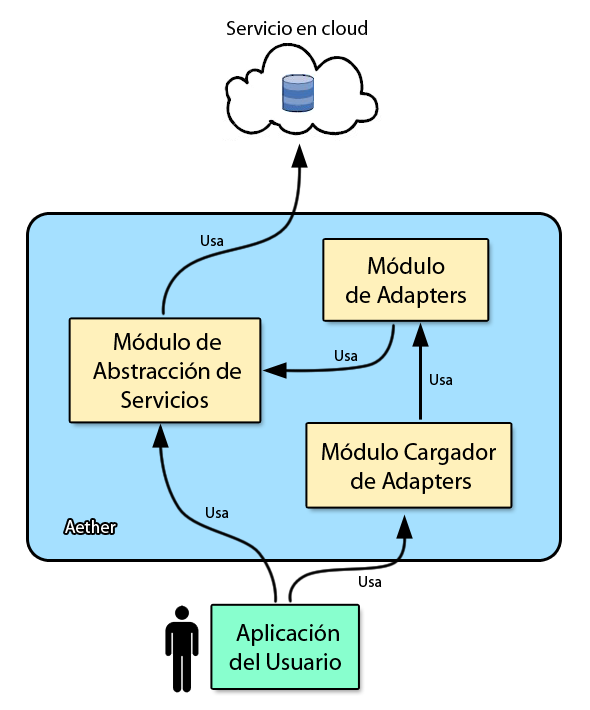


Figura 4.1 – Módulos de Aether y su interacción

En la figura anterior puede apreciarse tanto la interacción entre los tres módulos del framework como así también las dos maneras que tiene el usuario de utilizarlo. Una de ellas es haciendo uso directamente del Módulo de Abstracción de Servicios para los casos en que se desarrolle la aplicación utilizando este framework. La segunda opción es hacer uso de éste por medio del Módulo Cargador de Adapters (Aether-Loader). Esta última opción es práctica para ser utilizada en aplicaciones desarrolladas sobre algún otro framework ya que el Cargador de Adapters intercepta las llamadas al framework original y las redirige al adapter correspondiente presente en el Módulo de Adapters. Por último, éste módulo es el que se encarga de interactuar con el de Abstracción de Servicios para poder llevar a cabo las operaciones solicitadas.

A continuación, se detallarán las principales decisiones referidas al diseño de Aether. Cada una de éstas se tomó a partir de un problema específico de diseño.

#### 4.3.2.1.1 Módulo de Abstracción de Servicios (Aether-core)

El Módulo de abstracción de servicios tiene como función abstraer las interfaces de diferentes proveedores para diversos tipos de servicios. En este contexto se debe considerar la existencia de servicios de distintas naturalezas como pueden ser almacenamiento, cómputo o colas distribuidas. Debido a que las operaciones de cada servicio difieren de las del resto se definió una interfaz independiente para cada uno de ellos. Estas interfaces contienen todos los métodos requeridos para la utilización del tipo de servicio que se está tratando. Por ejemplo, para implementar un servicio de almacenamiento la interface genérica podría contener métodos para subir, bajar, eliminar y copiar archivos. Esto ayuda a facilitar la inclusión de nuevos servicios concretos (Por ejemplo, S3 o SQS), por lo tanto, estas interfaces son implementadas para cada proveedor sobre los que se desee dar soporte. En la figura 4.2 se presenta de forma gráfica lo indicado hasta el momento:

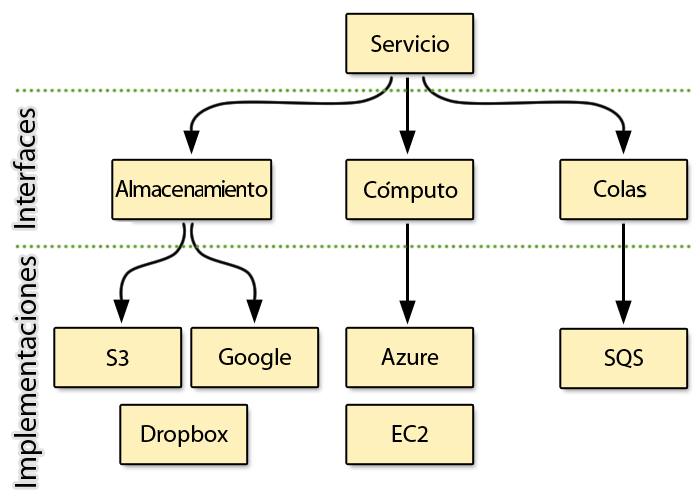


Figura 4.2 – Módulo de abstracción de servicios

En la figura anterior se puede apreciar cómo se trata cada servicio en particular. En parte media de la figura (“Interfaces”) se puede apreciar cada una de las interfaces correspondientes según la naturaleza del servicio, es decir, la interface correspondiente al almacenamiento, al cómputo y la utilizada para las colas. También puede apreciarse, en la parte inferior de la figura, las implementaciones concretas de cada interface para cada tipo de servicio.

Como se especificó anteriormente en el tercer requerimiento funcional, se debe posibilitar el cambio de proveedor para cada servicio de forma dinámica y sin necesidad de modificar el código fuente de la aplicación. Esto corresponde a un problema de configuración, el cual se atacó utilizando los patrones de diseño “Builder” y “Abstract factory”. El primero abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes. El segundo (Abstract Factory) permite crear objetos diferentes pero todos pertenecientes a la misma familia. A su vez, este patrón también permite la inclusión de manera sencilla de nuevas familias de objetos. La combinación de estos dos patrones de diseño provee al usuario un punto de acceso único a los servicios. Idealmente el usuario del framework deberá realizar llamadas del tipo “*Factory.getService (tipo, implementación)*”, donde “tipo” se refiere al tipo de servicio e “implementación” se refiere al servicio concreto que lo implementa. En la figura 4.3 se presenta la instanciación y configuración de los servicios por parte de la aplicación.

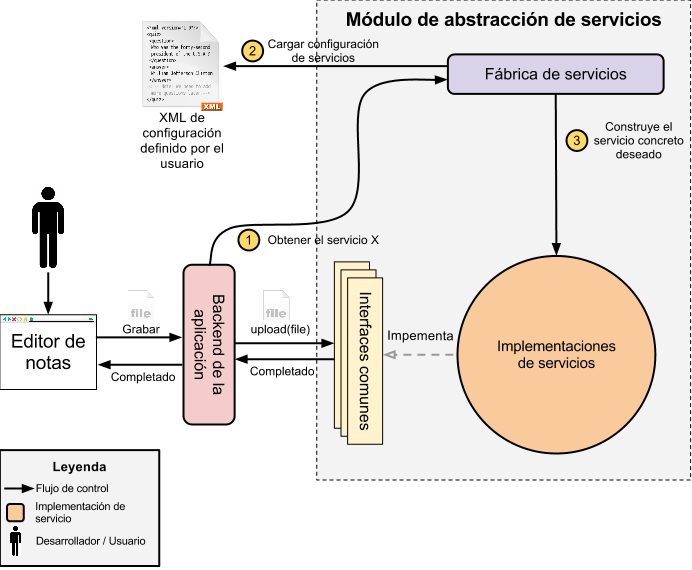
****

Figura 4.3 – Instanciación y configuración de servicios

Como puede apreciarse, el desarrollador obtiene los servicios utilizando la “Fabrica de Servicios”. Para obtener un servicio concreto el backend de la aplicación invoca al factory haciendo uso de una interface sencilla del tipo “Factory.getService (tipo, implementación)” como se indicó anteriormente. De este modo, cuando se recibe una llamada, el factory selecciona el builder correspondiente y retorna un servicio listo para utilizar. La configuración de este factory debe realizarse por medio de un archivo que detalla uno a uno los servicios disponibles del usuario. En el siguiente diagrama de secuencia (figura 4.4) se presenta la idea de este punto de acceso mediante un ejemplo de construcción para el servicio de S3.

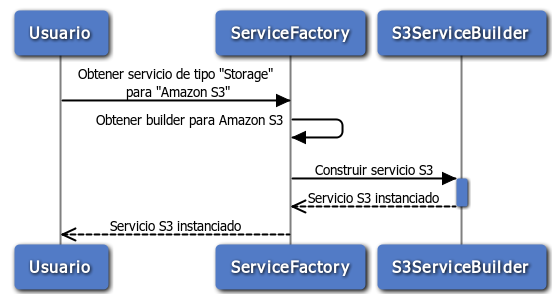


Figura 4.4 - Construcción de un servicio

Como se puede apreciar en la figura anterior, el código del usuario realiza un pedido al factory especificando el tipo de servicio y el proveedor del mismo. El factory construye entonces el servicio concreto con el builder indicado y lo devuelve al llamador. Es válido destacar que el servicio retornado debe ser del tipo de una de las interfaces de servicios planteadas anteriormente. El diagrama de clases que se presenta a en la figura 4.5 muestra la abstracción mencionada correspondiente a los servicios.

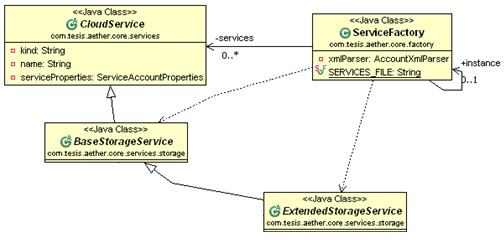


Figura 4.5 – Abstracción de servicios

Los servicios en Aether son instanciados y accedidos por medio de la clase ServiceFactory. Por medio de esta clase el usuario obtiene instancias listas para utilizarse de algún CloudService concreto. Un CloudService contiene el comportamiento genérico de un servicio en la nube del tipo que sea, esto incluye facilidades para conexión y desconexión, las propiedades del servicio que fueron cargadas en la configuración inicial del framework, el tipo de servicio y su nombre. No contiene gran carga de lógica ya que eso queda relegado a las subclases de tipos más específicos (storage, compute, queue, etc.).

Las especializaciones de CloudService son tipos de servicios abstractos, como es *BaseStorageService y ExtendedStorageService*, que contienen los métodos específicos para el tipo de servicio de storage contra el que se interactúa.

Ya definida la estructura principal para la instanciación de los servicios continuaremos explicando cómo se realiza la carga y configuración de estos. Una de las guías principales para el desarrollo de Aether es permitirle al usuario configurar el framework de manera simple. Idealmente una aplicación de usuario no tendría que tener conocimiento de que servicio concreto se utiliza, sino solo su tipo. Se debe proveer información acerca del servicio puntual de forma externa al código de la aplicación para facilitar la re-configuración y migración entre proveedores.

El modelo que se plantea en Aether está centrado en un único XML de configuración en el que el usuario tiene la posibilidad de ingresar las credenciales y cualquier otro atributo necesario para el correcto funcionamiento del servicio. A su vez, cada servicio posee un set de elementos de configuración mínimos que debe contener el XML.. Por ejemplo, si queremos utilizar Amazon S3 tenemos que asegurarnos de proveer la clave de acceso (Access key) y la clave secreta (secret key).

La base de esto es que luego de tener una aplicación funcional, el usuario solo necesita modificar el XML de configuración si desea migrar entre diferentes servicios, cambiar la información de sus cuentas, etc.

El único XML de configuración de servicios que se plantea tiene la forma que se presenta a continuación:

<storageServices>

<storageService class=*"com.tesis.aether.core.services.storage.imp.s3.S3StorageService"*>

<parameter key=*"s3AccessKey"* value=*"myAccessKey"*/>

<parameter key=*"s3SecretKey"* value=*"myPassword"*/>

</storageService>

</storageServices>

El servicio deseado se define mediante el atributo “class”. En este atributo se debe colocar el paquete y nombre de la clase que implementa el servicio que se desea utilizar. Adicionalmente se pueden colocar tantos elementos “parameter” como sea necesario para ingresar los datos del servicio. En el ejemplo se puede ver que el usuario desea utilizar Amazon S3 con la clave de acceso “myAccessKey” y la clave secreta “myPassword”.

Aether obtiene este XML y mediante las clases ServiceParser y AccountXmlParser configura todo lo necesario para que se puedan realizar los llamados al servicio deseado mediante la clase ServiceFactory.

De esta manera, con un simple cambio de XML, el usuario puede migrar entre diferentes servicios o cuentas sin costo adicional.

Cabe destacar también que este modelo soporta más de un servicio en el XML. Por ejemplo, si queremos agregar una cuenta de Google Storage a la configuración anterior tendríamos que incluir lo siguiente:

<storageService class=*"com.tesis.aether.core.services.storage.imp.google.storage.GoogleStorageService"*>

<parameter key=*"googleStorageAccessKey"* value=*"myGoogleAccessKey"*/>

<parameter key=*"googleStorageSecretKey"* value=*"myGoogleSecretKey"*/>

</storageService>

Utilizando la misma línea, si queremos migrar un servicio existente o simplemente cambiar las credenciales, debemos modificar su sección en el XML para proveer los parámetros necesarios. El código no se verá alterado ya que ServiceFactory comenzará a trabajar con la nueva configuración suministrándole al usuario una instancia de manera transparente.

El diseño que hace posible lo descrito anteriormente se puede ver en la figura 4.6.

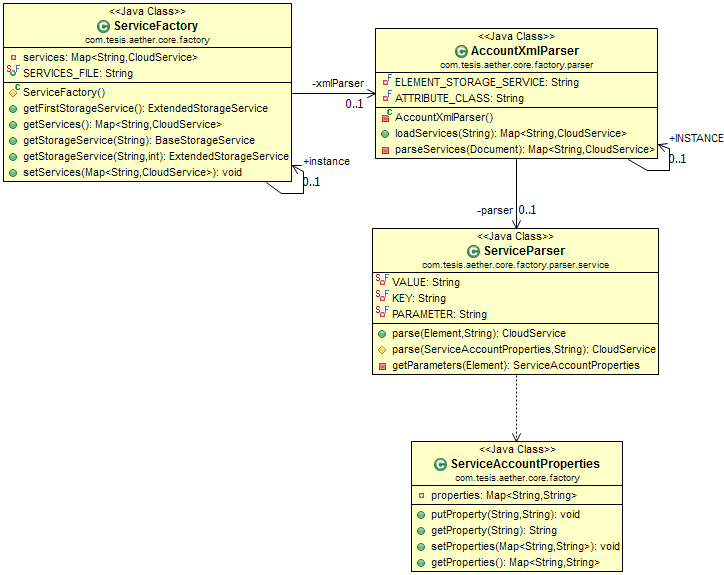


Figura 4.6 – Modelo de instanciación de servicios.

El punto de acceso para el usuario es la clase *ServiceFactory*, la cual se encuentra implementada respetando el patrón de diseño Singleton. El objetivo de este patrón es garantizar que la clase sólo tenga una instancia y de ésta manera proporcionar un punto de acceso global a ella. Esta clase (ServiceFactory) se encarga de la construcción de los servicios basándose en el XML de configuración proveyendo al usuario un objeto listo para utilizarse contra el servicio deseado. Las instancias de los servicios retornados por el factory son únicas, por lo que varios llamados a los métodos "*get*" con los mismos parámetros retornan las mismas instancias. Esto es útil para poder gestionar la concurrencia en los servicios provistos.

El proceso de carga de servicios puede apreciarse en la figura 4.7

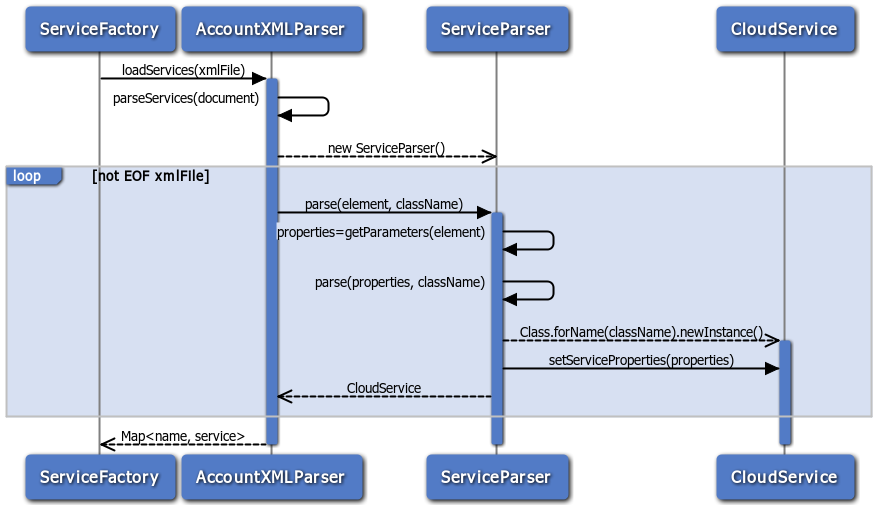


Figura 4.7 – Secuencia de instanciación de un servicio.

Como se puede observar, al momento de crear el Singleton de “*ServiceFactory*”, se cargan los servicios desde el archivo *xml* de configuración. Para esto se realiza una llamada al método “*loadServices*” correspondiente a la clase “*AccountXmlParser*” pasándole como parámetro la ruta al archivo de configuración. El parseo del XML es sencillo, simplemente debe leerse cada nodo del XML e invocar a ServiceParser para que construya la instancia efectiva del servicio. La clase ServiceParser lee todos los elementos ingresados por el usuario para un servicio particular en el XML de configuración y en base al atributo “class” del XML construye una nueva instancia del servicio por medio de mecanismos de reflexión. Adicionalmente, cada elemento “parameter” del XML es leído e inyectado en la instancia del servicio recién creada.

Para entrar un poco más en detalle, la figura 4.6 presenta un diagrama de secuencia correspondiente a la etapa de inicialización de un adapter y como éste interactúa con *ServiceFactory* para obtener la configuración deseada. Como ejemplo se detallará el corresponde al servicio JetS3.

Para éste caso particular partiremos desde el adapter correspondiente (*JetS3tAetherFrameworkAdapter*):

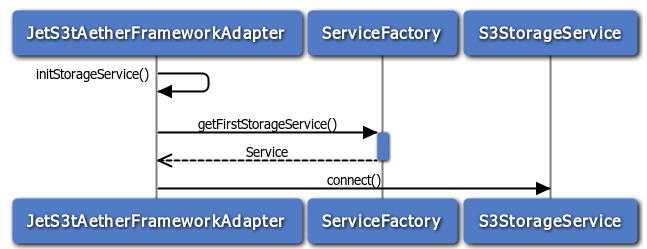


Figura 4.8 – Secuencia de inicialización de un adapter.

En la figura anterior puede verse claramente la secuencia de invocaciones entre los objetos al momento de inicialización de un adapter. “*AetherFrameworkAdapter*” es el encargado de obtener, mediante *ServiceFactory*, el servicio de storage y realizar la conexión correspondiente. La carga de los servicios es realizada al momento de inicializar el Singleton de *ServiceFactory,* de la misma forma que se hizo en el ejemplo anterior.

#### 4.3.2.1.2 Módulo de adapters para frameworks de terceros - Aether-adapters

Una vez desarrollada la aplicación del usuario utilizando el módulo de abstracción de servicios de Aether, es sencillo realizar el cambio de proveedor, pero ¿qué sucedería si el usuario ya posee una aplicación funcional que utiliza tecnología ajena a ésta plataforma y desea migrar a Aether? Por ejemplo, supongamos que un desarrollador X tiene implementada una aplicación utilizando el framework jClouds para acceder a Amazon S3 y desea migrarla a Google Storage con nuestra plataforma. Al usuario difícilmente le caería bien la idea de volverla a codificar utilizando el módulo de abstracción de servicios ya que perdería una base de código estable y testeado. Es por este motivo que se agrega al framework una capa de adapters para tecnologías ajenas a la plataforma. Con esto se logra que un usuario pueda utilizar Aether de manera transparente, manteniendo su base de código actual desarrollada para otro producto atacando de esta forma el segundo requerimiento funcional detallado al comienzo del capítulo: “Permitir la migración al framework desde aplicaciones ya desarrolladas utilizando otras tecnologías”.

Cada uno de los adapters del módulo realiza traducciones entre una tecnología X y Aether, haciendo uso de las interfaces del módulo de abstracción de servicios. Siguiendo el ejemplo, la plataforma proveerá un adapter para el framework jClouds de tal manera que la invocación a un método de jClouds será traducido a un set de llamados equivalentes de Aether. Es importante destacar que cada adapter debe tener en cuenta la traducción de objetos desde y hacia las tecnologías de terceros, es decir, realizar la serie de llamadas correspondientes al framework y luego transformar los resultados para brindar la salida correspondiente al framework original.

Debido a que todos los adapters poseen características comunes, se tomó la decisión de crear una clase abstracta sobre Aether-core que agrupe estas funcionalidades. Esta clase debe ser extendida por cada uno de los adapters que se implementen para poder mantener la consistencia entre ellos. A continuación se presenta un diagrama de clases simplificado en el cuál se muestra la clase abstracta y las implementaciones particulares de algunos adapters.

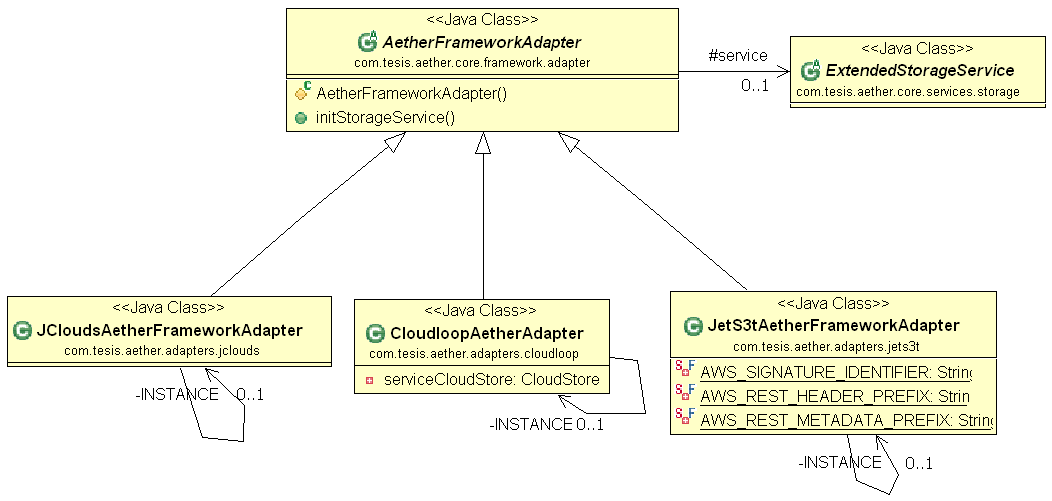


Figura 4.9 – Abstracción de adapters para frameworks de terceros.

En la imagen anterior se puede apreciar la clase abstracta perteneciente a Aether-core denominada “AetherFrameworkAdapter” y las implementaciones particulares de los adapters para los frameworks JClouds (JCloudsAetherFrameworkAdapter), Cloudloop (CloudloopAetherFrameworkAdapter) y JetS3t (JetS3tAetherFrameworkAdapter).

#### 4.3.2.1.3 Reemplazo dinámico de llamadas – Aether-loader

Una vez detallada la creación de los adapters se describirá la mecánica utilizada para lograr la detección de los llamados de la aplicación del usuario al framework utilizado y el reemplazo dinámico de la clase correspondiente por la del adapter.

Luego de analizar diferentes alternativas para atacar el problema de interceptar las llamadas y redirigirlas al adapter correspondiente determinamos que la mejor solución es detectar la carga de clases mediante un classloader personalizado, modificar la clase en tiempo de ejecución insertando las llamadas al adapter correspondiente y retornarla a la aplicación del usuario para que ésta no note la alteración y trabaje de manera normal.

Para lograr esto se recurrió a la utilización de una herramienta llamada Javassist, la cual provee, entre otras utilidades de importancia, la posibilidad de modificar el código de los métodos y atributos presentes en las clases ya “compiladas”. El poder de esta herramienta se combinó con el uso de reflexión que provee java y la utilización de un classloader personalizado para Aether, el cual por medio de la lectura de datos desde un archivo de configuración xml reconoce las clases que debe modificar y compilar nuevamente para ser utilizadas por la aplicación del usuario.

Cómo se indicó en el párrafo anterior, las clases especificadas en el archivo de configuración del classloader sufren modificaciones en el código. Estas modificaciones constan de agregar al comienzo de cada método de la clase a modificar, una llamada a uno con igual signatura presente en el adapter correspondiente y pasarle los parámetros recibidos. De esta forma, al momento de la nueva compilación en tiempo de ejecución, se deja sin efecto el resto de código presente en el método modificado. Las llamadas agregadas en este paso tienen la forma:

"return " + clase adapter + ".getInstance()." + nombre del método + "(parametros);"

El método “getInstance()” del cual depende la inyección de código debe haber sido implementado por cada adapter concreto que se vaya a utilizar.

A continuación se presenta un ejemplo sencillo en el cuál se describe la clase original, la clase correspondiente al adapter y el resultado final de la clase original modificada por el classloader y Javassist:

Se tiene la clase original “Clase1” con un método “metodo1” el cuál recibe como parámetro un valor entero “param1” y retorna un valor de tipo String como se muestra a continuación:

**public** **class** Clase1 {

**public** String metodo1(**int** param1) {

**return** "Hola " + param1;

}

}

El adapter correspondiente deberá poseer un método con la misma signatura que el declarado en la Clase1, por lo tanto el adapter quedaría como se muestra a continuación:

**public** **class** ClaseAdapter **extends** AetherFrameworkAdapter {

**private** **static** ClaseAdapter *INSTANCE* = **null**;

**protected** ClaseAdapter() {

**super**();

}

**public** **static** ClaseAdapter getInstance() {

**if** (*INSTANCE* == **null**) {

*INSTANCE* = **new** ClaseAdapter();

}

**return** *INSTANCE*;

}

**public** String metodo1(**int** parametro) {

**return** "Llamada modificada: Hola " + parametro;

}

}

En el archivo xml de configuración para Aether-loader se debería indicar el mapeo de clases con la estructura y datos siguientes:

<ClassLoaderConfig>

<classException>

<srcClass>Clase1</srcClass>

<dstClass>ClaseAdapter</dstClass>

</classException>

</ClassLoaderConfig>

Al correr la aplicación con estos elementos, el classloader irá cargando las clases correspondientes a medida que se soliciten verificando que sean diferentes a la especificada en el archivo de configuración (“Clase1”). En caso de detectar que la clase que se solicitó coincide con la especificada, procederá a cargarla y modificar la misma resultando ésta en el código que se muestra a continuación:

**public** **class** Clase1 {

**public** String metodo1(**int** param1) {

**return** ClaseAdapter.*getInstance*().metodo1(param1);

}

}

Gracias a este procedimiento de detección y modificación de código en tiempo de ejecución se libra al usuario en gran medida de tener que realizar cambios en la aplicación ya desarrollada. Los casos que suelen necesitar la intervención del usuario para la modificación del código ya desarrollado corresponden a la utilización de classloaders personalizados, es decir, que si la aplicación del usuario utiliza un classloader personalizado para ella es probable que se deba modificar el código del mismo para delegar la carga de las clases al cargador de clases de Aether.

Con respecto al diseño de este módulo, no fue necesario utilizar ningún patrón de diseño específico ya que la funcionalidad es muy sencilla y concreta. A continuación en la figura 4.10 se presenta el diagrama de clases correspondiente al cargador de clases de Aether..

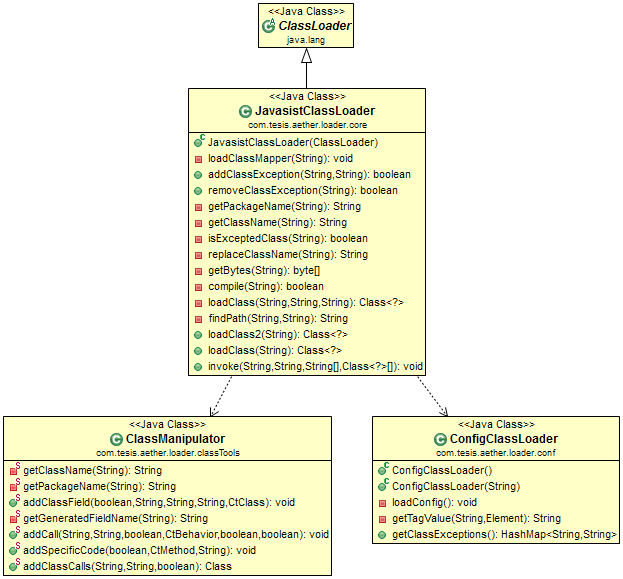


Figura 4.10 – Diagrama de clases de Aether-loader.

Este módulo consta de tres clases, la principal “JavasistClassLoader” provee las funcionalidades de carga de clases comunes de un classloader pero buscando coincidencias de la clase a cargar con las especificadas en el archivo de configuración. Otra de las clases de éste módulo es “ClassManipulator”, la cual es la encargada de modificar la clase que se indique agregando las llamadas al adapter correspondiente. Por último se encuentra la clase encargada de configurar el classloader leyendo los datos desde un archivo de configuración xml, ésta con el nombre ConfigClassLoader.

Como último punto, cabe destacar que para activar el uso del classloader de Aether se debe utilizar un switch adicional para la máquina virtual de Java. Esto permite definir como classloader del sistema al creado para Aether y de esta forma poder interceptar y redireccionar las llamadas hacia los adapters. El parámetro que debe especificarse para ejecutar la aplicación es el siguiente:

*-Djava.system.class.loader=com.tesis.aether.loader.core.JavasistClassLoader*

En la figura 4.11 se presenta la utilización del módulo cargador de adapters por parte de una aplicación ya desarrollada con una tecnología diferente.

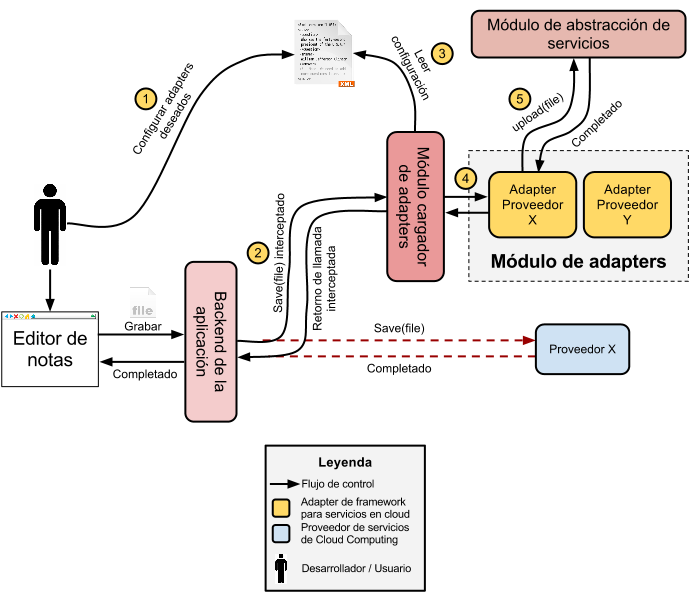


Figura 4.11 Utilización del módulo cargador de adapters de Aether.

Como se puede apreciar, el cargador de adapters se encuentra entre la aplicación del usuario y el framework utilizado. Una vez activado, el nuevo módulo comienza a capturar las invocaciones de la aplicación del usuario a la herramienta objetivo (jClouds). Cuando Aether detecta un método de interés carga los mapeos necesarios del archivo de configuración y reemplaza la llamada original por una llamada a su método homónimo en el adapter. Nótese también que el flujo original de la aplicación (líneas punteadas) se mantiene intacto con respecto a la nueva implementación. Tal es el caso que desde el punto de vista del desarrollador el método que se ejecuta sigue siendo el original. Al trabajar de este modo, el usuario del framework sólo debe tener conocimiento del módulo cargador y cómo configurarlo, lo que se traduce en simpleza a la hora de utilizar la plataforma.

1. http://maven.apache.org/ [↑](#footnote-ref-1)
2. http://www.jboss.org/javassist [↑](#footnote-ref-2)
3. http://www.eclipse.org/ [↑](#footnote-ref-3)