## 4. Diseño e implementación de Aether

En éste capítulo se detallará el diseño e implementación del framework Aether. Este framework cuenta con tres módulos bien definidos, cada uno de los cuales posee un conjunto de responsabilidades específicas. Debido a esto y a la comunicación que mantienen los módulos se puede apreciar como estilo arquitectónico general al de capas. En la capa más baja se encuentra el Módulo Cargador de Adapters (aether-loader), el cual contiene las funcionalidades para detectar la carga de clases y modificarlas en tiempo de ejecución según sea necesario. En la capa intermedia se encuentra el Módulo de Adapters (aether-adapters), el cual provee las facilidades para convertir las llamadas de una aplicación desarrollada utilizando otro framework a llamadas equivalentes de Aether. Por último, en la capa superior se encuentra el Módulo de Abstracción de Servicios (aether-core), el cual se encarga de abstraer los servicios prestados por diferentes proveedores y dar acceso a estos mediante una única interface. A continuación en la figura 4.1 se presentan los módulos nombrados y la interacción entre ellos.

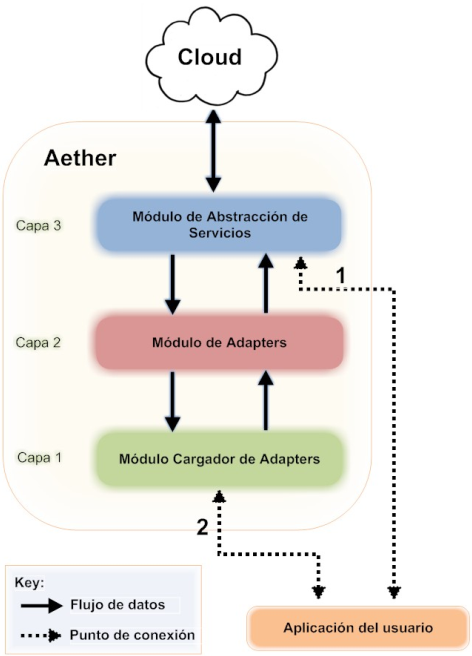


Figura 4.1 – Módulos de Aether y su interacción

El módulo de abstracción de servicios contiene la lógica necesaria para abstraer los servicios de cloud, configurar las conexiones y permitir las diferentes operaciones sobre cada uno de los servicios. La lógica presente en el módulo de adapters es la encargada 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. El último módulo corresponde al cargador de adapters. Este es el encargado de interceptar las llamadas a métodos de otros frameworks y realizar la invocación al correspondiente del adapter indicado. Por último, también puede apreciarse en la figura anterior las dos formas que tiene el usuario de utilizar el framework. Una de ellas es haciendo uso directamente del Módulo de Abstracción de Servicios (1) para los casos en que se desarrolle la aplicación utilizando este framework, mientras que la segunda opción es hacer uso de éste por medio del Módulo Cargador de Adapters (2). 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.

Con respecto a las herramientas utilizadas para el desarrollo del framework podemos mencionar que debido a que la implementación de cada uno de los módulos implica la utilización de diversas librerías desarrolladas por terceros, se tomó la decisión de utilizar un administrador de dependencias que facilite el manejo y actualización de las mismas. Para esto se utilizó una herramienta llamada Maven la cual posee un modelo de configuración de construcción simple basado en un formato XML permitiendo, entre otras cosas, una sencilla administración de dependencias. Adicional a esta herramienta fue necesario utilizar un IDE para el desarrollo del código. El entorno de desarrollo que se utilizó fue Eclipse ya que facilita la integración para el manejo de dependencias con Maven además de permitir el agregado de diversos complementos por medio plugins disponibles en la web. Cabe destacar también que este entorno es de código abierto y multiplataforma lo cual permite a los programadores utilizar para el desarrollo el sistema operativo de su preferencia.

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.

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

El Módulo de abstracción de servicios, como su nombre lo indica, tiene como función principal 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 y colas distribuidas. En la figura 4.2 se muestra de forma abstracta la organización de éste módulo.

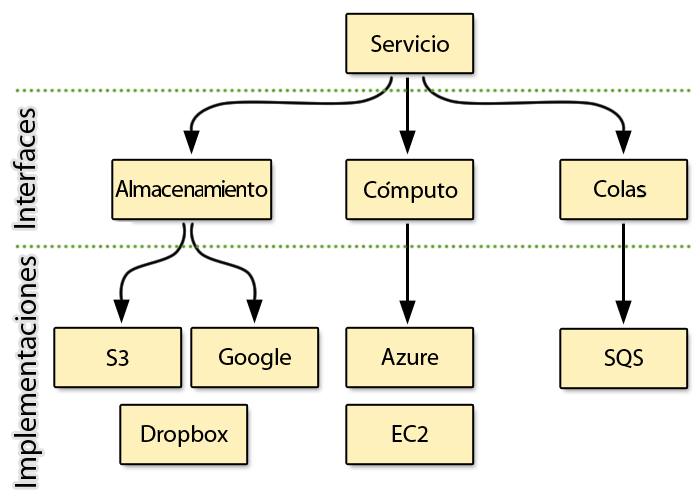


Figura 4.2 – Vista abstracta del Módulo de Abstracción de Servicios

En la figura anterior podemos apreciar que se definió una interfaz independiente para cada uno de los servicios, es decir, una interfaz para el Almacenamiento, otra para Cómputo y una última para Colas. Esto se debió a que las operaciones de cada uno de los servicios difieren de las del resto. 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; para un servicio de colas podría contener los métodos para encolar y obtener tareas y para un servicio de cómputo, la interface genérica podría contener un método para realizar algún cálculo. Esto facilita la inclusión de nuevos servicios concretos (Por ejemplo, S3 o SQS), por lo tanto, estas interfaces son implementadas para cada proveedor sobre el que se desee dar soporte. Estas implementaciones pueden verse en la parte inferior del gráfico en dónde se aprecian los servicios de S3, Google y Dropbox para el almacenamiento, Azure y EC2 para el cómputo y SQS para el servicio de colas.

En el mercado actual existe una gran diversidad de proveedores para cada uno de los servicios de Cloud. Estos proveedores poseen ciertas características particulares sobre los servicios que proporcionan como por ejemplo límites de almacenamiento, velocidad de subida y descarga de archivos, velocidad de procesamiento, costos de cada servicio, etc. lo cual lleva ocasionalmente a los usuarios a migrar su aplicación a un proveedor diferente del utilizado. Una migración de este estilo resulta incómoda y además en una pérdida de tiempo al surgir la necesidad de recodificar la aplicación para adaptarse a las nuevas interfaces provistas para su uso. Debido a estos problemas, se diseñó Aether para facilitar el cambio entre proveedores de forma dinámica y sin necesidad de alterar el código fuente de la aplicación. A continuación en la figura 4.3 se presenta el diseño de clases que hace posible esto:

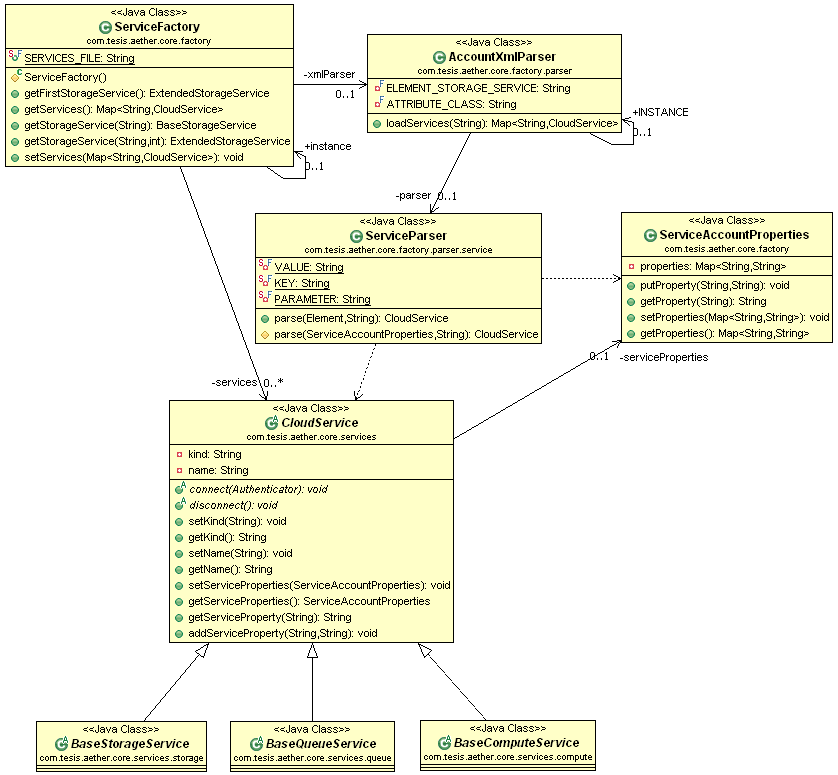


Figura 4.3– Diagrama de clases para la abstracción de servicios

Como puede apreciarse en la figura anterior, esto corresponde a un problema de configuración el cual se atacó utilizando dos fábricas (o factories) para crear y mantener disponibles cada uno de los servicios. Estas son “ServiceFactory” y “ServiceParser”. Básicamente la función de ServiceFactory es forzar la carga de los servicios que se encuentren especificados en un archivo de configuración y mantenerlos disponibles para cuando se soliciten, mientras que la principal función de ServiceParser es crear las instancias de cada uno de los servicios y retornarlos según sea necesario. A continuación se describe más en detalle las responsabilidades de cada una de las clases presentes en el diagrama anterior.

La clase *ServiceFactory* es el punto de acceso para el ususario. Esta clase se encuentra implementada respetando el patrón de diseño Singleton. El objetivo de este patrón es garantizar que sólo exista una instancia y de ésta manera proporcionar un punto de acceso global a ella. 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 este factory son únicas, por lo que diferentes 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.

La funcionalidad de las tres clases restantes (CloudService, AccountXmlParser y ServiceParser) es sencilla. CloudService contiene el comportamiento genérico de un servicio en la nube tales como facilidades para conexión y desconexión, las propiedades del servicio que fueron cargadas desde un archivo XML de configuración definido por el usuario, el tipo de servicio y su nombre. AccountXmlParser se encarga de cargar y analizar el archivo XML de configuración. Para esto, AccountXmlParser, lee cada nodo del XML e invoca a ServiceParser para que construya la instancia efectiva del servicio. Debido a que ServiceParser no conoce a priori los objetos que puede crear, no es posible realizar la instanciación de estos de la forma tradicional. Para resolver este problema se utilizó una API de java llamada “Reflection”. Entre otras utilidades que provee esta API, permite crear instancias de objetos indicando un nombre de paquete y clase. Por medio de este mecanismo, Aether puede crear instancias específicas de “CloudService” y posterior a esto agregarle a dicha instancia las propiedades indicadas en el archivo de configuración. Debido al uso de reflexión para la carga de clases es de suma importancia que la configuración presente en el archivo sea la correcta.

Los servicios que se crean deben pertenecer a algún subtipo de CloudService, los cuales se aprecian en la figura con los nombres BaseStorageService, BaseQueueService y BaseComputeService y corresponden a los servicios de cloud de almacenamiento, colas y cómputo respectivamente. A su vez, cada uno de estos tres servicios posee implementaciones específicas ya que las clases anteriores sólo describen la estructura básica que debe respetar cada implementación particular. A continuación en la figura 4.4 se presentan como ejemplo algunas de estas implementaciones particulares para el servicio de almacenamiento y las superclases involucradas.

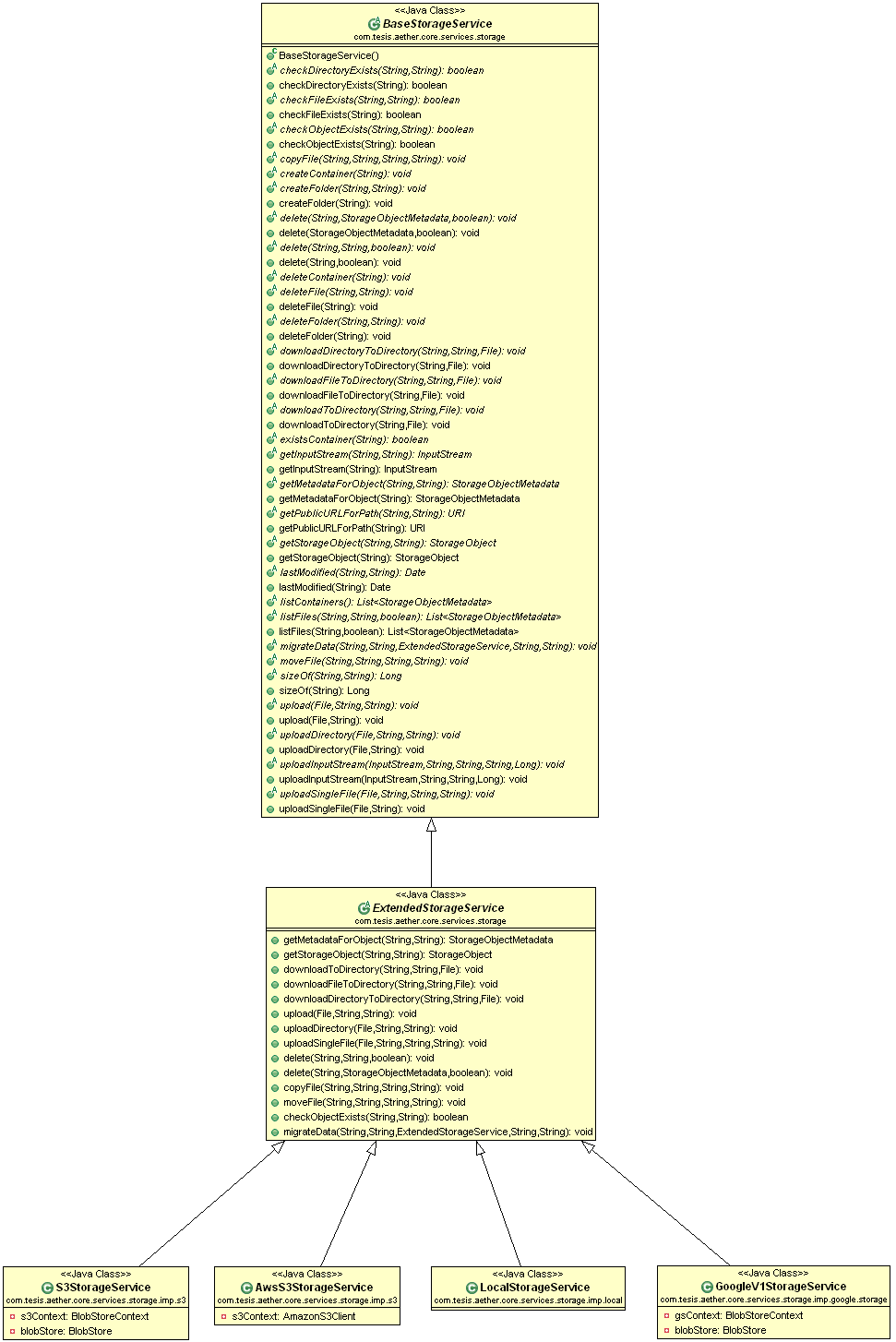


Figura 4.4 – Especificaciones del servicio de almacenamiento.

En esta figura puede verse que ExtendedStorageService extiende la clase básica del servicio de almacenamiento (BaseStorageService) agregando comportamiento extra, luego S3StotageService, AwsS3StorageService, LocalStorageService y GoogleV1StorageService extienden a esta última para implementar el comportamiento correspondiente a cada servicioparticular. En estos casos, las implementaciones particulares que pueden apreciarse corresponden a los servicios de almacenamiento S3 y AWS de Amazon, un servicio de almacenamiento local (por medio del uso de File) y por último un servicio de almacenamiento en Cloud de Google.

El archivo XML de configuración que se mencionó anteriormente permite a Aether determinar el proveedor que debe utilizar y los parámetros de los que debe hacer uso para interactuar con éste. El mismo posee una estructura sencilla en dónde se distinguen sólo tres elementos. A continuación en la figura 4.5 se presenta la estructura de este archivo definido para el servicio de almacenamiento de Google.

<storageServices>

<storageService class=*"aether.core.services.storage.imp.GoogleV1StorageService"*>

<parameter key=*"container"* value=*"container1"*/>

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

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

</storageService>

</storageServices>

Figura 4.5 – Estructura del archivo XML de configuración para el servicio de almacenamiento

El primer elemento define la sección de servicios disponibles correspondientes a un tipo específico, por ejemplo para el servicio de almacenamiento se definió con el nombre de storageServices. El segundo elemento, llamado storageService, posee un atributo “class” por medio del cual se indica el objeto que se deberá crear y configurar al instanciar el servicio. Este elemento tiene además subelementos llamados “parameter” los cuales deben tener obligatoriamente dos atributos: key y value. El valor del atributo “key” corresponde al nombre del parámetro que reconocerá el framework, mientras que el texto del atributo “value” indica su valor.

Como puede apreciarse, en el archivo de configuración del framework se encuentra definida la clase y parámetros que se deben utilizar para interactuar con el proveedor del servicio, con lo cual una migración a un proveedor diferente sólo recae en la modificación de los datos del archivo estableciendo la nueva clase que permitirá manejar las llamadas a los servicios del nuevo proveedor y los parámetros necesarios para poder interactuar con él. A continuación en la figura 4.6 se muestra un diagrama de secuencia en donde por medio de un ejemplo de instanciación de un servicio se puede ver de forma gráfica la interacción entre las clases mencionadas hasta el momento.

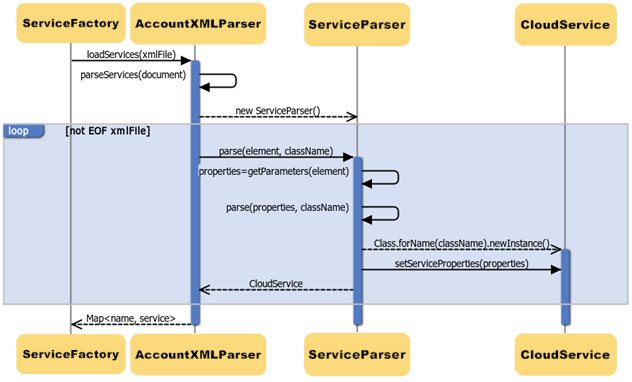


Figura 4.6 – Secuencia de instanciación de un servicio.

En la figura anterior puede verse que 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, AccountXmlParser simplemente lee cada nodo del XML e invoca a ServiceParser mediante el método parse(element, className) 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 haciendo uso de sus métodos getParameters(element) y parse(properties, className) para luego, en base al atributo “class” del XML, construir 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.

## 4.2 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 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 agregó 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.

Cada uno de los adapters del módulo realiza traducciones entre una tecnología específica 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 llamadas 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 Aether y luego transformar los resultados para brindar la salida correspondiente al framework original. Para lograr esto se utilizó el patrón de diseño “Adapter” (también conocido como “Wrapper”).

Debido a que todos los adapters poseen ciertas 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 en la figura 4.7 se presenta un diagrama de clases simplificado en el cuál se muestra la clase abstracta y las implementaciones particulares de algunos adapters.

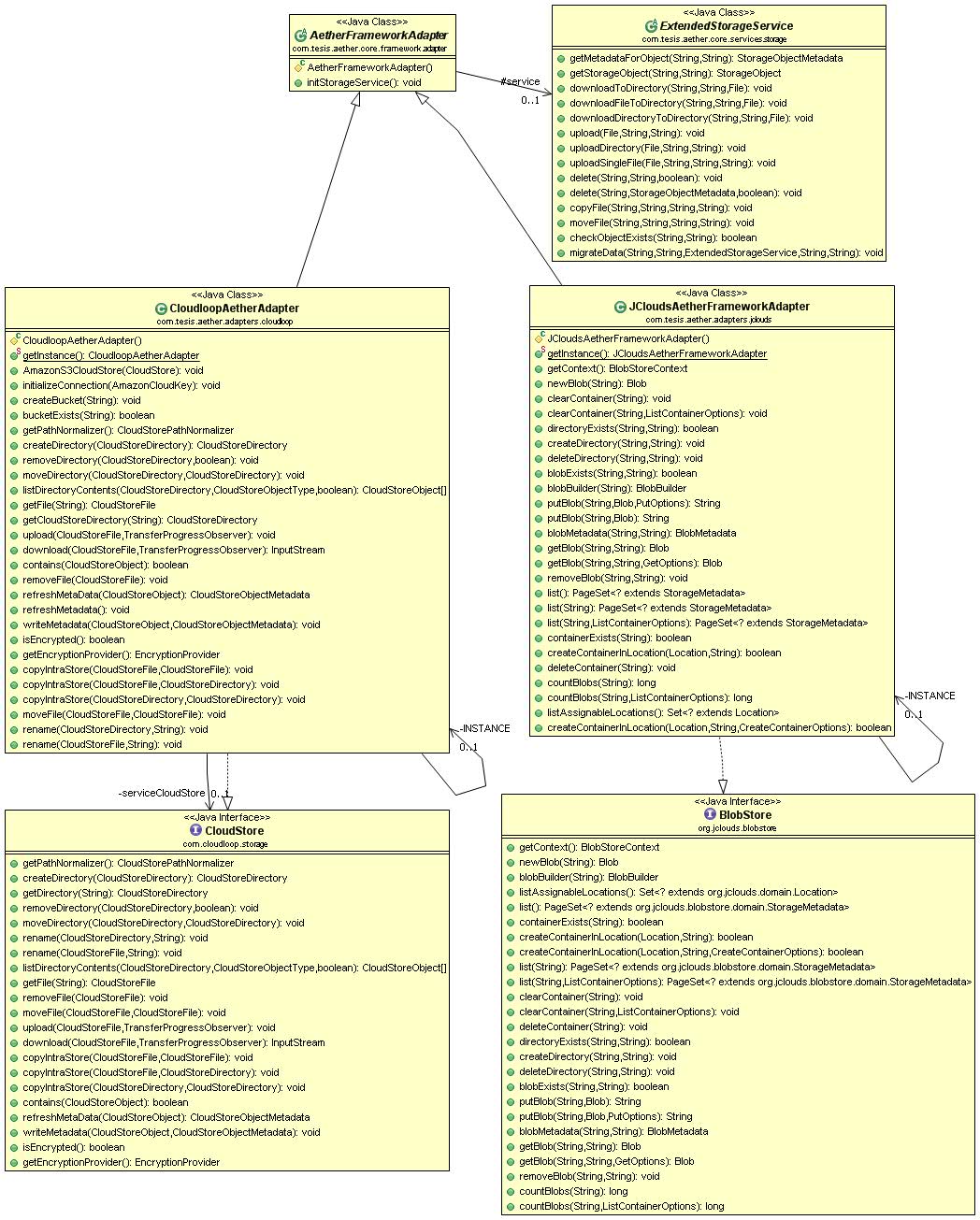


Figura 4.7 – Abstracción de adapters para frameworks de terceros.

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 Cloudloop (CloudloopAetherFrameworkAdapter) y JClouds (JCloudsAetherFrameworkAdapter) los cuales extienden la funcionalidad de la primera y permiten llevar a cabo las traducciones mencionadas. Con respecto a los roles que cumple cada clase de la figura en relación al patrón “Adapter” podemos ver que AetherFrameworkAdapter corresponde al target u objetivo, CloudStore y BlobStore corresponden al adaptee o interface existente que se necesita adaptar y por último CloudLoopAetherAdapter y JCloudsAetherFrameworkAdapter corresponden a las clases adapters en sí.

Para comprender mejor la funcionalidad de un adapter y su interacción con Aether se presenta a continuación en la figura 4.8 un diagrama de secuencia en dónde se puede ver la inicialización del adapter y una solicitud de almacenamiento de un archivo por parte de la aplicación del usuario. Como ejemplo se presenta el caso para una aplicación de usuario desarrollada utilizando JetS3t y Aether configurado para utilizar el almacenamiento de Google.

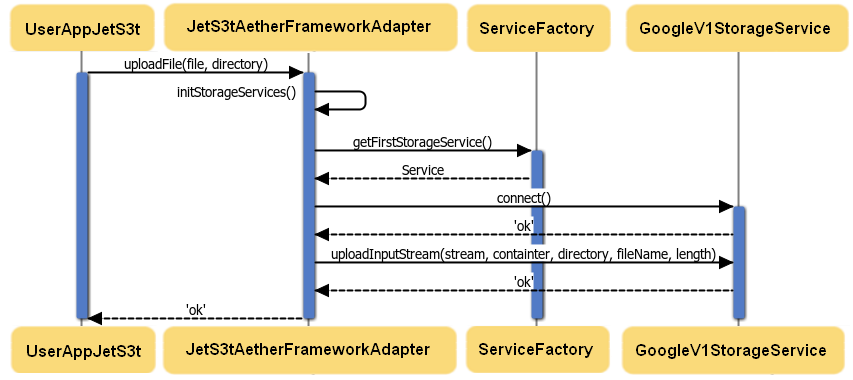


Figura 4.8 – Secuencia de inicialización de un adapter y solicitud de almacenamiento.

En la figura anterior puede apreciarse la solicitud de almacenamiento de un archivo por parte de la aplicación del usuario (denominada “UserAppJetS3t”) al adapter de JetS3t (JetS3tAetherFrameworkAdapter) por medio del método uploadFile. El adapter realiza el proceso de inicialización haciendo uso de su método initStorageServices desde el cual se solicita a ServiceFactory el servicio que tenga configurado por medio del método getFirstStorageService. Una vez que el adapter obtuvo el servicio, procede a realizar la conexión invocando al método connect de dicho servicio, en este caso correspondiente a GoogleV1StorageService. Cuando finaliza el proceso de conexión, el adapter realiza los llamados correspondientes a la solicitud de la aplicación del cliente. En este caso traduce la llamada a uploadFile de JetS3t por su equivalente en Aether uploadInputStream pasándole sus respectivos parámetros. GoogleV1StorageService es en este punto el responsable de almacenar los datos según corresponda.

## 4.3 Módulo para el reemplazo dinámico de llamadas (Aether-loader)

Como se pudo ver en la figura 4.8 correspondiente a la sección 4.2, la aplicación del usuario es la que realiza la llamada al método del adapter. Para que estas llamadas se hagan de manera transparente para la aplicación del usuario se creó un módulo para lograr detectarlas y redireccionarlas al adapter correspondiente.

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 en tiempo de ejecución la clase del framework que utiliza la aplicación del cliente 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 en tiempo de ejecución el código fuente de 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.

Debido a que la funcionalidad de este módulo es sencilla y concreta no fue necesario utilizar ningún patrón de diseño específico. A continuación en la figura 4.9 se presenta el diagrama de clases correspondiente que hace posible estas detecciones y modificaciones de código.

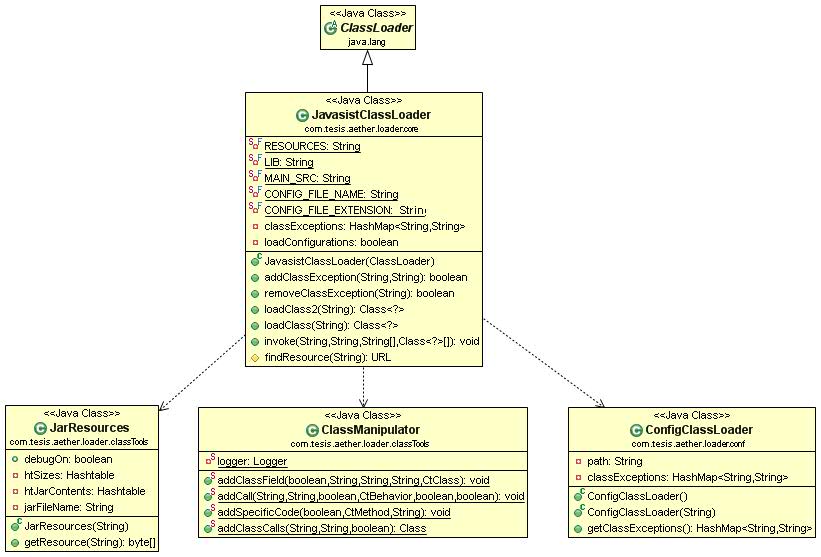


Figura 4.9 – Diagrama de clases de Aether-loader.

Como puede verse en el diagrama anterior, Aether-loader consta de cuatro clases. La principal es “JavasistClassLoader” la cual provee las funcionalidades de carga de clases comunes de un classloader pero posee además una funcionalidad para detectar clases particulares y modificarlas antes de retornarlas. ConfigClassLoader se encarga de configurar el cargador de arranque con los datos especificados en un archivo de configuración XML. En este archivo se encuentran detallados los mapeos que se deben realizar entre las clases del framework de tercero y las clases del adapter. JarResources permite buscar y cargar las clases que se soliciten y se encuentren dentro de archivos jar. Por último, ClassManipulator es la encargada de modificar la clase que se indique agregando las llamadas al adapter correspondiente.

El archivo de configuración correspondiente a este módulo es muy simple y se debe encontrar presente en cada adapter desarrollado debido a que las clases que se deben detectar y modificar difieren de un framework a otro. A continuación en la figura 4.10 se presenta la estructura del archivo XML de configuración.

<ClassLoaderConfig>

<classException>

<srcClass>com.cloudloop.storage.adapter.CloudStoreAdapterBase</srcClass>

<dstClass>com.tesis.aether.adapters.cloudloop.CloudloopAetherAdapter</dstClass>

</classException>

<classException>

<srcClass>com.cloudloop.storage.adapter.amazonS3.AmazonS3AdapterBase</srcClass>

<dstClass>com.tesis.aether.adapters.cloudloop.CloudloopAetherAdapter</dstClass>

</classException>

</ClassLoaderConfig>

Figura 4.10 – Estructura del archivo XML de configuración de un adapter

En la figura anterior pueden apreciarse cuatro elementos diferentes. ClassLoaderConfig determina la sección de configuración. ClassException contiene la información correspondiente a un mapeo de clases en el cual srcClass indica la clase original que deberá ser modificada al momento de cargarla mientras que dstClass hace referencia a la clase correspondiente al adapter que posee las implementaciones de los métodos de la original. Vale destacar que el archivo de configuración de un adapter puede contener más de un mapeo de clases como se puede apreciar en el ejemplo.

Cómo se indicó anteriormente, 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:

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 modificarla 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. Las aplicaciones ya desarrolladas que suelen necesitar la intervención del usuario para adaptar su código a Aether son las que hagan uso de classloaders personalizados. En estos casos se debe modificar el código del classloader de la aplicación para delegar la carga de clases al cargador de Aether.

Es muy importante destacar que para activar el uso del classloader de Aether se debe utilizar un switch adicional para la máquina virtual de Java. Esto permite definirlo como classloader del sistema 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*

Como es de suponer, el cargador de adapters se encuentra entre la aplicación del usuario y el framework que ésta utiliza. Una vez iniciada la aplicación utilizando este módulo, Aether comienza a capturar las invocaciones de la aplicación del usuario a la herramienta objetivo (jClouds por ejemplo). Cuando se detecta un método de interés se reemplaza la llamada original por una llamada a su método homónimo en el adapter indicado en el archivo de configuración. Cabe destacar también que el flujo original de la aplicación 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.