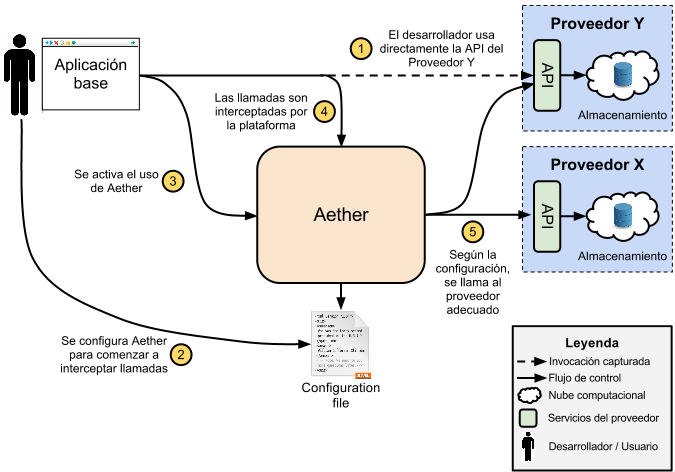
## 3. Enfoque

Durante el análisis de las herramientas actuales se logró detectar una serie de problemas o puntos no tenidos en cuenta que son de sumo interés para usuarios de este tipo de plataformas. Los problemas principales que se encontraron se relacionan fundamentalmente con las capacidades de migración de las herramientas. Por un lado podemos distinguir la migración de aplicaciones entre servicios, por ejemplo de Amazon S3 a Google Storage. Este tipo de migración suele ser tenida en cuenta por los diseñadores de las herramientas para cloud computing, pero no se encuentra disponible un mecanismo que realmente facilite la tarea al usuario que en muchos casos debe volver a codificar para adaptarse a los objetos del nuevo servicio. Por el otro lado se nos presenta la migración de aplicaciones entre herramientas o frameworks para cloud computing. Un ejemplo de esto sería tener una aplicación codificada con el SDK de Amazon S3 y querer comenzar a utilizar Google Storage provisto por jClouds. Ninguna de las herramientas disponibles en la actualidad considera este tipo de migración y esto deriva en que el desarrollador deba recodificar la aplicación para utilizar la nueva plataforma.

A lo largo de este capítulo se presenta el enfoque propuesto para darle una solución a esta problemática poniendo el foco en proveer una plataforma sencilla y de bajo costo en cuanto a líneas de código.

### 3.1. Vista general de Aether

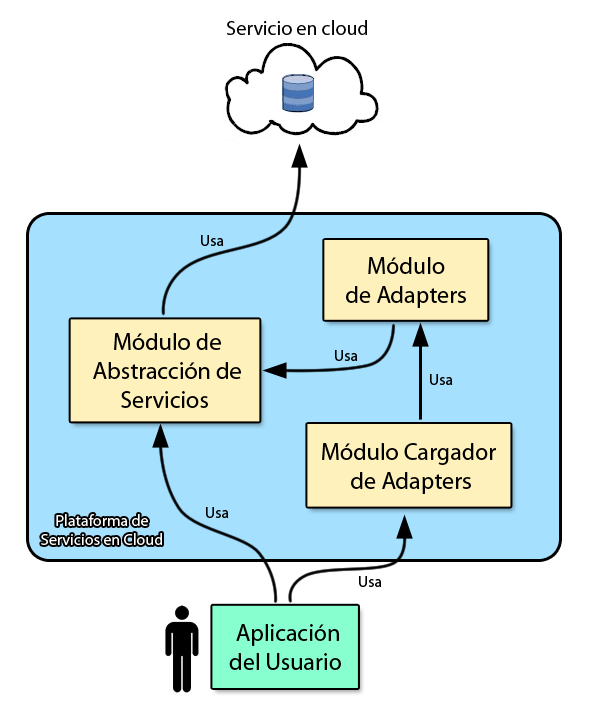
Para darle solución a la problemática planteada se ideo una plataforma que cuenta con varios puntos de interés. Esta plataforma es capaz de comunicarse con distintos servicios de cloud computing, proveyendo interfaces simples y homogéneas entre todos ellos. Esto brinda al desarrollador de una aplicación la posibilidad de utilizar Aether contra más de un servicio, intercambiándolos de ser necesario. El otro gran punto de interés de la plataforma es la capacidad de poder migrar aplicaciones ya codificadas con otro framework o que sencillamente utilizan una API ofrecida por un proveedor de servicios particular. Esto último es posible gracias a un mecanismo de reflexión, donde las invocaciones a un framework o API objetivo son interceptadas, traducidas y trabajadas dentro de Aether sin que el usuario o desarrollador tenga que involucrarse más allá de la configuración. Un primer vistazo de este enfoque puede observarse en la **Figura 3.1.**



**Figura 3.1**

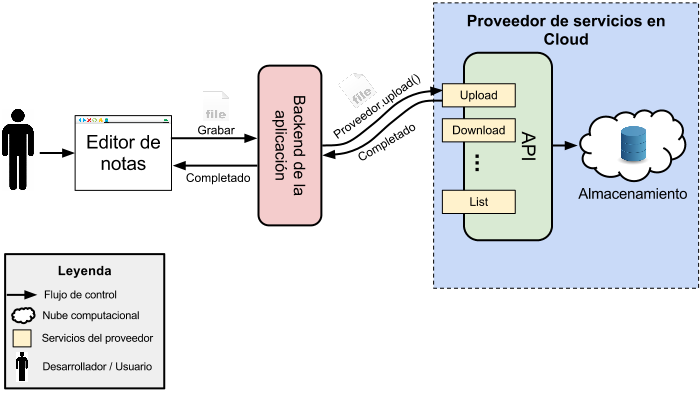
Como se puede observar, el usuario de Aether activa la plataforma por medio de archivos de configuración. En estos archivos se detallan los servicios que se desean utilizar y que tipo de llamadas se deben interceptar. Una vez configurado, Aether comenzara a capturar las invocaciones deseadas, generando de manera dinámica traducciones a métodos de Aether compatibles. Durante este proceso es necesario convertir el modelo utilizado por el código del usuario al modelo interno de Aether. Luego de generarse los mapeos, estos se ejecutan contra el servicio objetivo que el usuario configuró previamente. Al finalizar la ejecución de estas operaciones se retorna el control al código original del usuario, que en ningún momento debió modificarse para utilizar Aether.

Para cumplir sus funciones Aether cuenta con tres partes bien definidas como se muestra en la **Figura 3.2**. El módulo de abstracción de servicios es el encargado de proveer interfaces comunes a los servicios de cada proveedor (por ejemplo almacenamiento). Cada una de estas interfaces presenta al desarrollador una mirada única a un tipo de servicio sin importar la implementación que tenga detrás. Esto abstrae al usuario de cuestiones particulares de un proveedor como por ejemplo el modelado de carpetas en servicios de tipo mapa (por ejemplo S3). De este modo, un desarrollador puede usar estas interfaces para codificar una aplicación sin atarse a la API de un proveedor particular. Los módulos de adapters y cargador de adapters tienen la misión de solucionar el problema de migrar una aplicación ya codificada. El módulo de adapters se compone de diversos adaptadores, cada uno de los cuales hace de puente entre los métodos de una herramienta particular (API de un proveedor, otro framework, etc.) y el módulo de abstracción de servicios de nuestra plataforma. El nexo que une a la aplicación del usuario y al adapter requerido es el modulo cargador de adapters. Haciendo uso de las traducciones definidas en los adapters, este módulo es capaz de inyectar estas traducciones en la aplicación del usuario de manera transparente.



**Figura 3.2**

Llevaremos esto a un caso concreto suponiendo la existencia de una aplicación “Cloud Recorder” que permite elaborar y almacenar notas periodísticas. Esta aplicación presenta una interfaz gráfica sencilla en la que un usuario puede cargar archivos de audio o texto, añadir comentarios y persistirlos de manera segura. Al momento de persistir las notas, la interfaz gráfica hace uso de un método “Grabar” provisto por el backend de la aplicación. Para mantener un backup off-site de las notas los diseñadores de la aplicación decidieron utilizar un servicio de almacenamiento en cloud. Luego de realizar un relevamiento de la oferta de distintos proveedores, se selecciona a uno de ellos y se procede a integrar su API resultando en un flujo similar al presentado en la **Figura 3.3**.



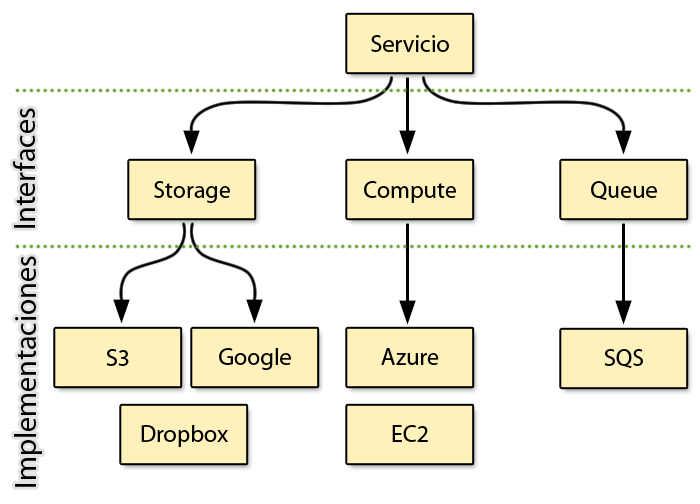
​**Figura 3.3**

Este enfoque presenta un problema claro, el backend de la aplicación queda atado a los métodos de la API del proveedor que haya elegido. Si por alguna razón se requiere cambiar el proveedor de servicios de almacenamiento (costos, performance, disponibilidad del servicio, etc.) los desarrolladores se verían obligados a re codificar el backend de la aplicación. Para solucionar este problema es necesario contar con un paso intermedio que permita abstraer al backend de los servicios de distintos proveedores. La idea consiste en permitir que en cualquier momento el desarrollador pueda cambiar de proveedor sin necesidad de modificar el código de la aplicación. Para este fin también es útil contar con archivos de configuración que permitan modificar el proveedor utilizado sin la necesidad de realizar cambios en el código. También es deseable contar con un mecanismo que facilite la migración en caso de tener una aplicación ya codificada como sucede en nuestro ejemplo.

En las secciones siguientes se analizan los módulos de Aether, concebidos para dar solución a la problemática del ejemplo. El primer punto a analizar será el módulo de abstracción de servicios, siguiendo con los módulos de adapters y cargador de adapters.

### 3.2. Módulo de abstracción de servicios

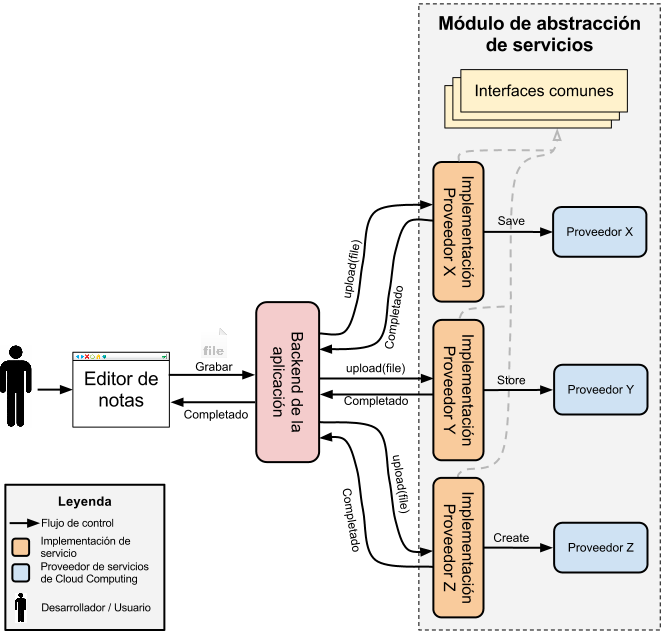
Comenzaremos con el análisis del módulo de abstracción de servicios. En este punto se implementa el soporte para cada servicio en cloud que se desee proveer (por ejemplo almacenamiento) y sus implementaciones particulares (por ejemplo almacenamiento en Amazon S3).



**Figura 3.4**

Por cada tipo de servicio el usuario puede acceder a una interface única y simple. La **Figura 3.4** presenta como ejemplo interfaces para servicios como “storage” (almacenamiento en cloud), “compute” (utilización de máquinas virtuales en cloud) y “queues” (colas distribuidas para pasaje de mensajes). Estas interfaces contienen todos los métodos requeridos para la utilización del tipo de servicio que se está tratando. Por ejemplo, para un servicio de almacenamiento la interface genérica debe proveer métodos para subir, bajar, eliminar y copiar archivos. Contar con estas interfaces tiene grandes beneficios. En primer lugar facilita la inclusión de nuevos servicios concretos (Por ejemplo, S3 o EC2) a la plataforma. Para esto tan solo se debe implementar una serie de métodos bien definidos para el nuevo servicio. En segundo lugar, abstraen a la aplicación que las usa de las particularidades del servicio que seleccione.

Tomando estas ideas podemos pensar en mejorar Cloud Recorder. Recordemos que el principal problema a resolver era la dependencia con el API de un proveedor particular. La **Figura 3.5** presenta como resulta la nueva estructura de la aplicación incluyendo las mejoras del módulo de abstracción de servicios y sus interfaces comunes.



**Figura 3.5**

Como se puede apreciar, se introduce una serie de interfaces comunes cuyas implementaciones separan al backend de la aplicación del usuario de las API finales de cada proveedor. Haciendo uso de las implementaciones de estas interfaces se logra eliminar la dependencia directa que surgía al programar contra una API específica. Esto produce que el cambio de proveedor se reduzca a intercambiar la implementación de la interface de servicio utilizada, su instanciación y configuración. Por ejemplo, supongamos que se desarrolló una aplicación con la API de Amazon S3. Esta aplicación sencilla se conecta al servicio, lista el contenido de un directorio y descarga todos los archivos. El pseudo código de esta aplicación, obviando funcionalidad sin interés para el ejemplo, puede verse a continuación:

//Instanciación del servicio

AmazonS3 s3 **=** **new** AmazonS3Client**(new** BasicAWSCredentials**(**accessKey**,** secretKey**));**

//Utilización del servicio

ObjectListing objectListing **=** s3**.**listObjects**(new** ListObjectsRequest**().**withBucketName**(**bucketName**));**

**for** **(**S3ObjectSummary objectSummary **:** objectListing**.**getObjectSummaries**())** **{**

S3Object object **=** s3**.**getObject**(new** GetObjectRequest**(**bucketName**,** objectSummary**.**getKey**()));**

InputStream is **=** object**.**getObjectContent**();**

writeInputStream**(**is**,** archivo**);**

**}**

Como puede verse, se trabaja con objetos dependientes de S3 por lo que cambiar de servicio (Google Storage por ejemplo) significaria recodificar la aplicación. Si adaptamos este mismo ejemplo a la estructura propuesta utilizando las interfaces en lugar de llamadas a la API puntual llegamos al siguiente pseudo codigo:

//Instanciación del servicio

StorageInterface service = new AmazonS3Service(accessKey, secretKey, region)

//Configuración del servicio

service**.**setEndpoint**(**…**)**

…

//Utilización del servicio

List archivos **=** service**.**list**(**file**)**

**for(**archivo**:** archivos**){**

service**.**download**(**archivo**)**

**}**

Se pueden apreciar los bloques sencillos de instanciación, configuración y uso del servicio, en este caso Amazon S3. A diferencia de la versión anterior ya no se muestran dependencias con la API de Amazon S3 debido a que su funcionalidad quedó totalmente encapsulada dentro de StorageInterface. Una hipotética migración de la aplicación a Google Storage resultaría únicamente en un cambio para la instanciación y configuración del servicio como se muestra a continuación:

//Instanciación del servicio cambiada para utilizar Google Storage

**StorageInterface service = new GoogleStorageService(user, password)**

//Configuración del servicio

service**.**setEndpoint**(**…**)**

**service.setGoogleStorageApiVersion(…)**

…

//Utilización del servicio

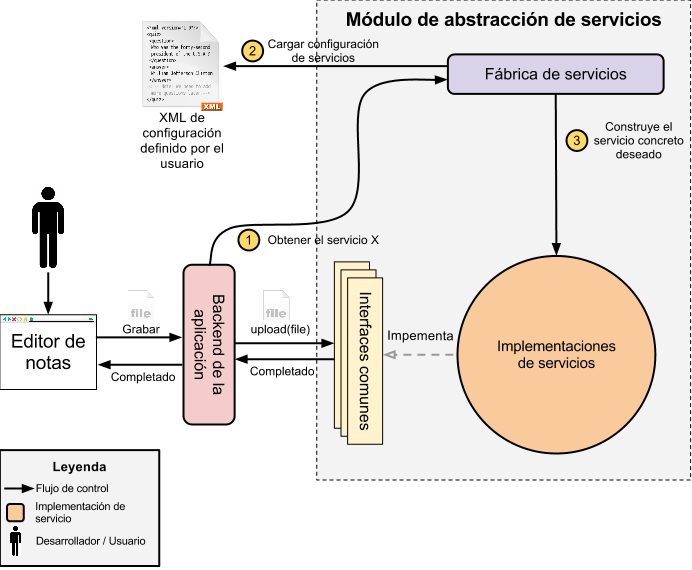
List archivos **=** service**.**list**(**file**)**

**for(**archivo**:** archivos**){**

service**.**download**(**archivo**)**

**}**

Se puede observar que el uso de las interfaces facilita mucho la migración del código entre Amazon S3 y Google Storage. Tan solo se tuvo que cambiar la línea de construcción del servicio y proveer los datos de configuración necesarios a la implementación de Google Storage. Pese a esto sigue quedando un punto flojo en el esquema ya que la configuración de cada servicio concreto se debe realizar por código sin ningún mecanismo de construcción que simplifique la tarea. Este problema se soluciona utilizando factories y builders proveyendo al usuario de un punto de acceso único a los servicios para que este no tenga que preocuparse de la construcción y configuración de las instancias que desea. La **Figura 3.6** presenta el mecanismo de construcción propuesto.

​**Figura 3.6**

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)*”, donde “tipo” se refiere al tipo de servicio e “implementación” se refiere al servicio concreto que lo implementa. De este modo, cuando se recibe una llamada el factory selecciona el builder indicado y devuelve 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. El esquema propuesto para el archivo de configuración se muestra a continuación.

<Services>

<storageServiceclass=**"GoogleStorageService"**>

<parameterkey=**"googleStorageAccessKey"**value=**""**/>

<parameterkey=**"googleStorageSecretKey"**value=**""**/>

</storageService>

<storageServiceclass=**"S3StorageService"**>

<parameterkey=**"s3AccessKey"**value=**""**/>

<parameterkey=**"s3SecretKey"**value=**""**/>

</storageService>

</Services>

En este caso el usuario posee dos servicios de almacenamiento en cloud, uno provisto por Amazon S3 y el otro por Google Storage. Cada proveedor debe configurarse con una serie de parámetros específicos a la plataforma dentro de los elementos “parameter”. Estos parámetros pueden ir desde claves para validación de usuario hasta directorios locales para cache, pasando por endpoints, puertos y demás elementos de interés. Esta configuración será utilizada por el factory de servicios al momento de construir la instancia del servicio deseado. Debe notarse que gracias a este mecanismo ya no es necesario que el usuario modifique el código de su aplicación para realizar una migración entre protocolos. A partir de ahora el código producido tendrá la forma siguiente:

//Instanciación y configuración del servicio

StorageInterfaceservice**=**Factory.getService**(“Storage”, “S3”)**

//Utilización del servicio

List archivos **=**service**.**upload**(**file**)**

**for(**archivo**:** archivos**){**

service**.**download**(**archivo**)**

**}**

Si nos interesa migrar a Google Storage tan solo debemos cambiar el identificador de la implementación. En nuestro caso, cambiaríamos “S3” por “Google” resultando en el siguiente código.

//Instanciación y configuración del servicio

StorageInterfaceservice**=**Factory.getService**(“Storage”, “Google”)**

//Utilización del servicio

List archivos **=**service**.**upload**(**file**)**

**for(**archivo**:** archivos**){**

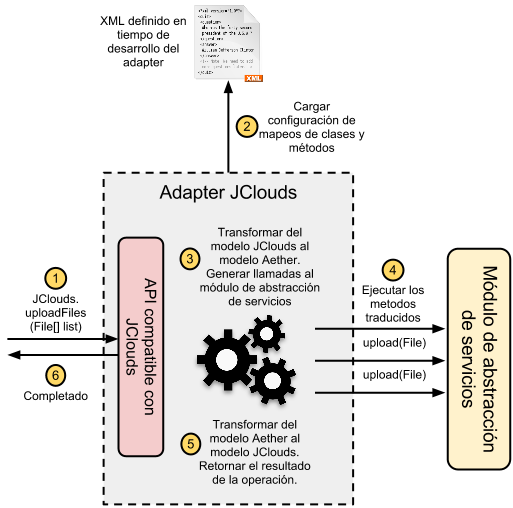
service**.**download**(**archivo**)**

**}**

### 3.3. Utilización del módulo cargador de adapters

Ya hemos analizado el soporte para migraciones de aplicaciones entre protocolos sin necesidad de cambios de código. Sin embargo, no hemos considerado un punto importante: para lograr esto se debe recodificar la aplicación original para que utilice de manera directa el módulo de abstracción de servicios. Pese a que este es el modo más directo de utilizar la nueva plataforma, al desarrollador seguramente no le resulte agradable tener que cambiar toda la aplicación ya que perdería una base de código estable y testeado. Teniendo esto en mente, iremos más allá e incluiremos un modo de migración que no requiera cambio de código alguno. Es por este motivo que surge la idea de generar una capa de adapters para tecnologías ajenas a la plataforma. Cada uno de estos adapters realiza traducciones entre una tecnología X y nuestra plataforma haciendo uso de las interfaces del módulo de abstracción de servicios.

Siguiendo el ejemplo de Cloud Recorder, supongamos que la aplicación original estaba implementada con jClouds para acceder a S3 y ahora se desea acceder a Google Storage por medio de la nueva plataforma. Para estos casos Aether provee un adapter para el framework jClouds como se muestra en la **Figura 3.7**. Este adapter presenta una interface compatible con la herramienta original (jClouds) donde cada método es capaz de utilizar el modelo original de jClouds y traducir la llamada entrante en una o más invocaciones equivalentes al módulo de abstracción de servicios de Aether. La respuesta, en el modelo de Aether, es reconvertida por el adapter al modelo de jClouds completando la traducción transparente al usuario.

**

**Figura 3.7**

Para gestionar la configuración interna de cada adapter se utiliza un archivo XML que define que clases del framework objetivo se van a interceptar y que clase adapter se hará cargo de las traducciones necesarias. El esquema propuesto es el siguiente:

<ClassLoaderConfig>

<classException>

<srcClass>**jclouds.class.FirstClass**</srcClass>

<dstClass>**JCloudsAetherFrameworkAdapter**</dstClass>

</classException>

**...**

<classException>

<srcClass>**jclouds.class.SecondClass**</srcClass>

<dstClass>**JCloudsAetherFrameworkAdapter**</dstClass>

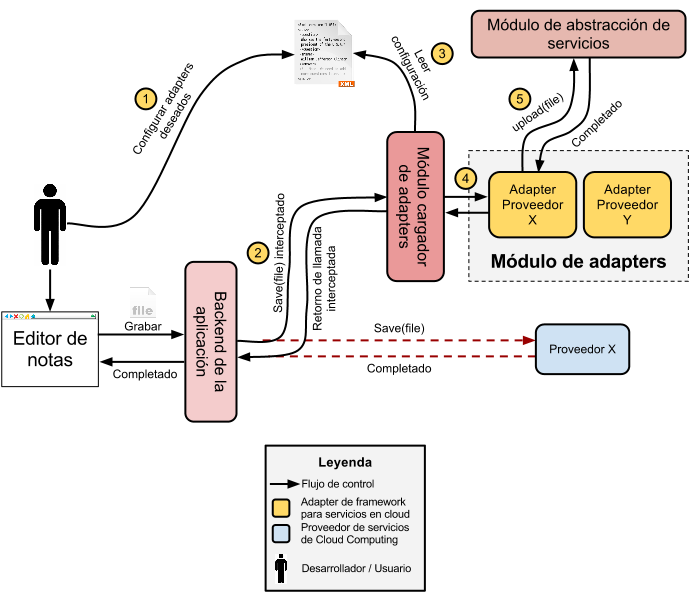
</classException>

</ClassLoaderConfig>

Siguiendo este mismo mecanismo podemos crear adapters para cualquier herramienta externa que deseemos teniendo en cuenta los detalles de cada una.

Resta analizar cuál es el mecanismo de carga para estos adapters, motivo por el cual se crea el módulo cargador de adapters. Este módulo es capaz de tomar cada adapter necesario e inyectar sus métodos dentro de la aplicación del usuario manteniendo al mínimo las modificaciones de código y configuración. Al lograr esto, el usuario no se ve obligado a cambiar su diseño / código ya probado y tan solo necesita familiarizarse con el uso de un módulo de nuestra plataforma.

La **Figura 3.8** presenta una continuación para el ejemplo de Cloud Recorder planteando el uso del reemplazo dinámico de llamadas.



**Figura 3.8**

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.