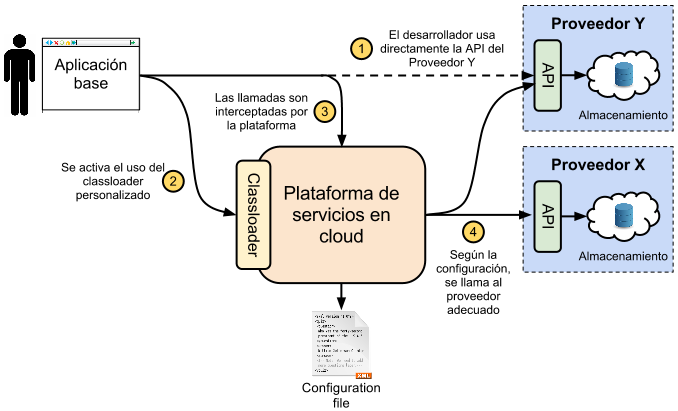
## 2. Enfoque

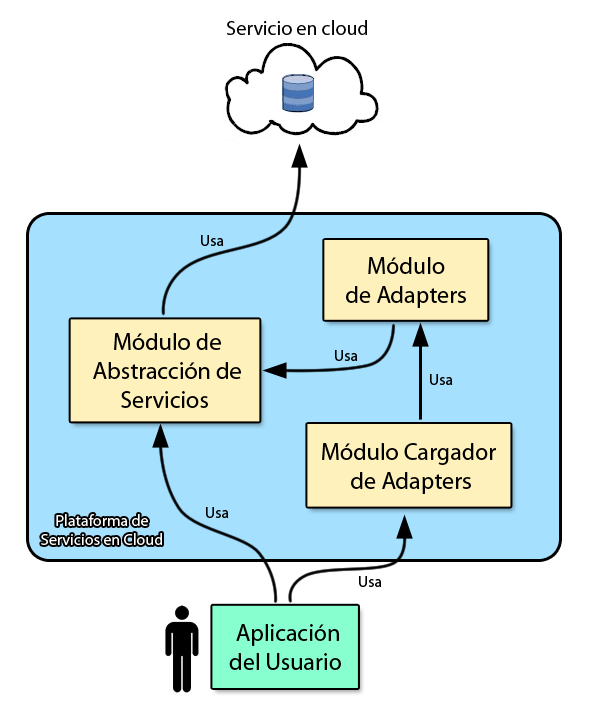
Este capítulo presenta el enfoque propuesto para darle una solución a los problemas encontrados en las herramientas actuales. Se pondrá énfasis en facilitar la migración entre protocolos y entre frameworks, intentando en todo momento mantener los cambios de código al mínimo.

La idea de la plataforma consiste en hacer uso de un class loader personalizado que permita inyectarle código a las clases que utiliza el desarrollador en su aplicación. La **Figura 2.1** muestra la operatoria de este enfoque. El primer paso consistiría en configurar la aplicación del usuario para que utilice el class loader provisto por la plataforma. Cuando la aplicación solicita la instancia de un servicio, este class loader modifica los métodos necesarios y devuelve la clase modificada al usuario. Cada una de estas modificaciones reemplaza el código original del método para que pase a utilizar al modulo de abstracción de servicios para realizar su tarea. De esta manera, cada vez que se haga uso de uno de estos métodos se estará invocando al código modificado de manera transparente al desarrollador, sin que este haya tenido que realizar cambios en su código. Dada la configuración de la plataforma de servicios en cloud cada método será redirigido al proveedor necesario.



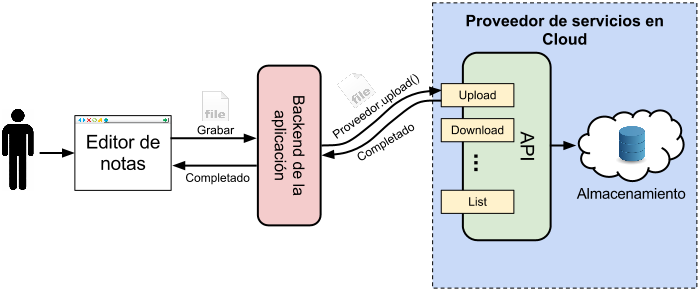
**Figura 2.1**

Dentro de la plataforma propuesta se pueden apreciar tres partes bien definidas como se muestra en la **Figura 2.2**. El módulo de abstracción de servicios será el encargado de proveer interfaces comunes a los servicios de cada proveedor (por ejemplo almacenamiento). Un desarrollador podrá 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 una herramienta particular (API de un proveedor, otro framework, etc.) y el módulo de abstracción de servicios de nuestra plataforma. Por último, el modulo cargador de adapters será el encargado de inyectar las invocaciones necesarias en la aplicación del usuario de manera transparente haciendo uso de los adapters.



**Figura 2.2**

Llevaremos esto a un caso concreto suponiendo la existencia de una aplicación “Cloud Recorder” que permita elaborar y almacenar notas periodísticas por medio de una interfaz gráfica. Para facilitar el acceso a 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 2.3**.

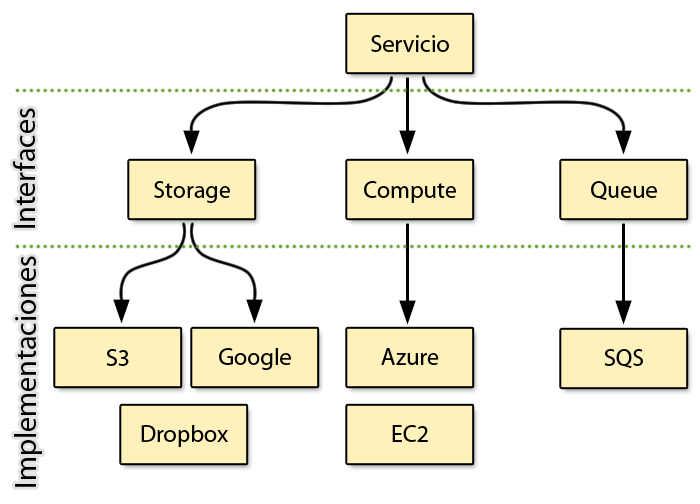


​**Figura 2.3**

Este enfoque presenta un problema claro, la aplicación del usuario queda atada a 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 la aplicación. Para solucionar este problema sería necesario contar con un framework o aplicación que permita abstraer los servicios de distintos proveedores en una interface única. La idea sería permitir que en cualquier momento el desarrollador pueda cambiar de proveedor sin necesidad de modificar el código de la aplicación. Para este fin sería útil contar con archivos de configuración que permitan modificar el proveedor utilizado sin la necesidad de realizar cambios en el código. También sería deseable contar con un mecanismo que facilite la migración en caso de tener una aplicación ya codificada como sucede en nuestro caso de ejemplo. Para solucionar estos problemas se desarrollara e implementara el uso de los módulos propuestos para la nueva plataforma. El primer paso será darle forma a la utilización del modulo de abstracción de servicios. Se tomaran en cuenta aspectos tales como configuración y migración entre protocolos. Para finalizar, se adoptara el uso del módulo cargador de adapters con el fin de proveer una alternativa sin cambios de código para la migración de Cloud Recorder.

### 2.1. Utilización del módulo de abstracción de servicios

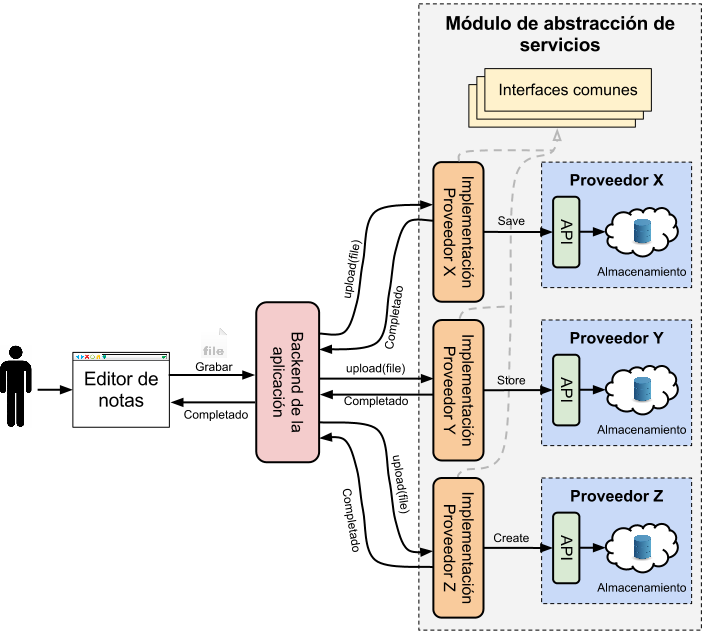
Comenzaremos el análisis con el módulo de abstracción de servicios. En este punto se implementa el soporte para cada servicio en cloud que se desee proveer (por ejemplo almacenamiento) y sus implementaciones particulares (por ejemplo almacenamiento en Amazon S3).



**Figura 2.4**

Por cada tipo de servicio el usuario debe poder acceder a una interface única y simple. La **Figura 2.4** 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 deben contener 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 debería contener métodos para subir, bajar, eliminar y copiar archivos. Contar con estas interfaces facilita la inclusión de nuevos servicios concretos (Por ejemplo, S3 o EC2) a la plataforma.

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 2.5** presenta como sería la nueva estructura de la aplicación si se implementaran las mejoras del módulo de abstracción de servicios y sus interfaces comunes.



**Figura 2.5**

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. Por ejemplo, supongamos que se desarrolló una aplicación con la API de Amazon S3. Esta aplicación sencillamente se conecta al servicio, lista el contenido de un directorio y descarga todos los archivos. El pseudo código, 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**);**

**}**

Si adaptamos este mismo esquema a la estructura propuesta utilizando las interfaces en lugar de llamadas a la API puntual llegariamos 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. Ya no se muestran dependencias con la API de Amazon S3 debido que su funcionalidad quedó totalmente encapsulada dentro de StorageInterface. Una hipotética migración 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 facilito 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. 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. Atacaremos este problema por medio de un mecanismo de construcción y configuración utilizando factories y builders. Esto provee al usuario un punto de acceso único a los servicios. Idealmente tendría que 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. El factory debe entonces construir el servicio concreto con el builder indicado y devolverlo al llamador. Es útil destacar que el servicio retornado debe ser del tipo de una de las interfaces de servicios planteadas anteriormente. La configuración de este factory debería poder realizarse por medio de archivos de configuración que detalle uno a uno los servicios disponibles del usuario. El esquema propuesto 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

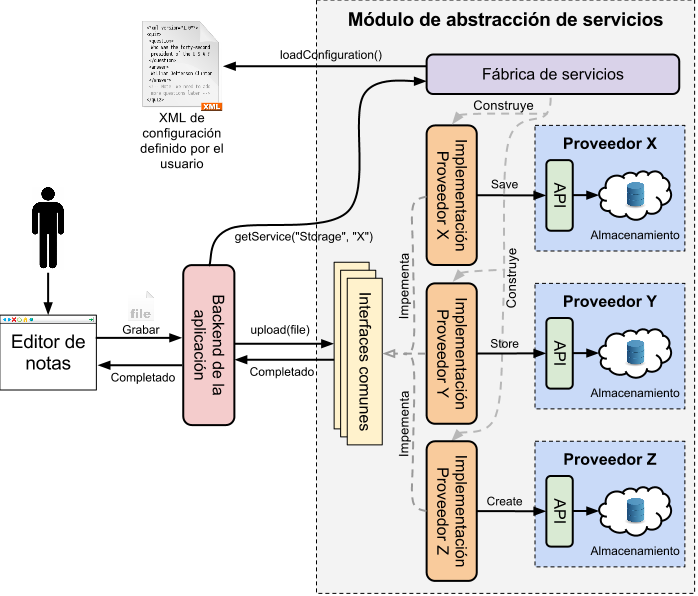
List archivos **=**service**.**upload**(**file**)**

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

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

**}**

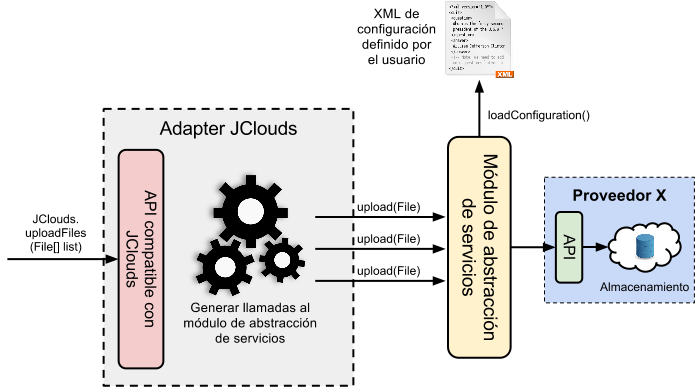
Ya estamos en condiciones de agregar este nuevo mecanismo a Cloud Recorder. Como se puede apreciar en la **Figura 2.6**, el desarrollador obtiene los servicios utilizando la “Fabrica de Servicios”. Esta fábrica de servicios obtiene los datos de configuración por medio del XML cargado por el usuario. Esto produce que ya no se tengan que instanciar y configurar los servicios de forma manual. Es decir, ya no se necesita comunicarse directamente con las implementaciones de cada proveedor sino únicamente con la instancia de la interface devuelta por la “Fabrica de Servicios”. Gracias a esto, en ningún punto del código el desarrollador deberá codificar para un proveedor particular.

​**Figura 2.6**

### 2.2. Utilización del módulo cargador de adapters

A partir de las últimas modificaciones, Cloud Recorder podrá ser migrada entre protocolos sin necesidad de cambios de código. Sin embargo, no hemos considerado un punto importante: para lograr esto se debió 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, podemos ir más allá e imaginar 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 realizaría 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. La plataforma debería proveer un adapter para el framework jClouds como el de la **Figura 2.7**.Este adapter debería trabajar de tal manera que un llamado “*llamado (parámetro…parámetro): valor de retorno*” pueda ser traducido a un set de llamados equivalentes de nuestra plataforma. Es útil destacar que cada adapter debe tener en cuenta la traducción de objetos desde y hacia las tecnologías de terceros.

**

**Figura 2.7**

Para gestionar la configuración interna de cada adapter se utilizará un archivo XML que defina que clases del framework objetivo a interceptar. El archivo también debe definir qué clase adapter se hará cargo de reemplazar las llamadas necesarias. El esquema propuesto es el siguiente:

<ClassLoaderConfig>

<classException>

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

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

</classException>

**...**

<classException>

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

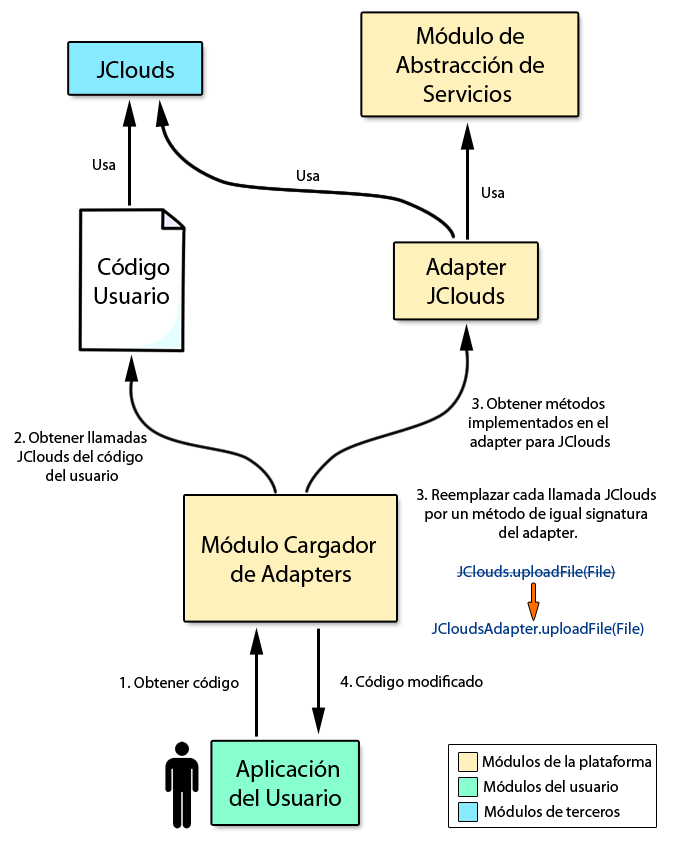
<dstClass>**JCloudsAetherFrameworkAdapter**</dstClass>

</classException>

</ClassLoaderConfig>

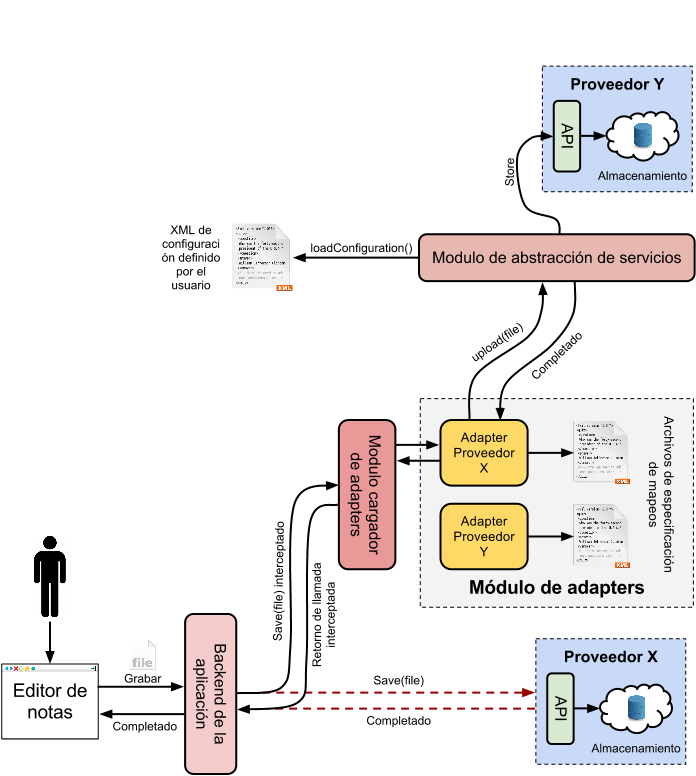
Teniendo el modulo que realizará la traducción de llamadas surge la pregunta: ¿Cómo trabajan estos adapters con la aplicación del usuario? Hasta ahora lo único que hicimos fue diseñar un intérprete para traducir los métodos de un framework a otro, pero no analizamos cómo se haría esto de manera cómoda para el usuario. Para dar una respuesta entra en escena el ultimo módulo de la plataforma, el modulo cargador de adapters. Teniendo en cuenta que el usuario ya cuenta con una aplicación diseñada y programada para trabajar con una tecnología X, este módulo debe ser capaz de tomar cada adapter necesario e inyectar sus métodos dentro de dicha aplicación manteniendo al mínimo las modificaciones de código y configuración. Si se logra esto, el usuario no se vería obligado a cambiar su diseño / código ya probado y tan solo necesitaría familiarizarse con el uso de un módulo de nuestra plataforma.

La **Figura 2.8** presenta una continuación para el ejemplo del adapter de jClouds planteando en este contexto el modelo adoptado para realizar el reemplazo dinámico de llamadas.



**Figura 2.8**

Como se puede apreciar, el cargador de adapters se encuentra entre la aplicación del usuario y el framework utilizado. De esta manera el usuario sólo debe tener conocimiento del módulo cargador y cómo configurarlo, lo que se traduce en simpleza a la hora de utilizar la plataforma. Puertas adentro el cargador debe encargarse de detectar cada llamada a jClouds para poder traducirlas por medio del adapter. Cada una de estas llamadas es reemplazada por una llamada del adapter que posea la misma signatura pero sólo si el método fue implementado. Por ejemplo, una llamada “JClouds.uploadFile (file)” se traduciría en “JCloudsAdapter.uploadFile (file)”. Internamente el adapter trabaja con los servicios del módulo de abstracción de servicios y con el mismo jClouds para poder mantener todos los objetos que espera el código del usuario.Una vez que se realizan todos los reemplazos estamos en condiciones de retornar el código modificado para utilizar los adapters. La **Figura2.9** completa el ejemplo de Cloud Recorder mostrando el uso de este nuevo mecanismo dentro del contexto de la aplicación.



**Figura 2.9**

Como se puede apreciar, el circuito básico de la aplicación original del usuario (líneas punteadas) se mantiene intacto con respecto a la nueva implementación. El nuevo framework, por medio de su configuración y el adapter seleccionado, inyecta sus métodos para reemplazar los utilizados por la aplicación del usuario. A partir de ese momento, el método “save” invocado por la aplicación pasa a ser el método “save” interceptado. Si el usuario desea cambiar el proveedor de almacenamiento tan solo deberá cambiar el XML de configuración del framework de la misma manera que lo hacía anteriormente.