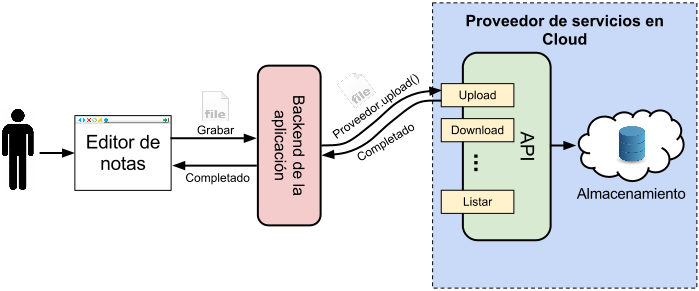
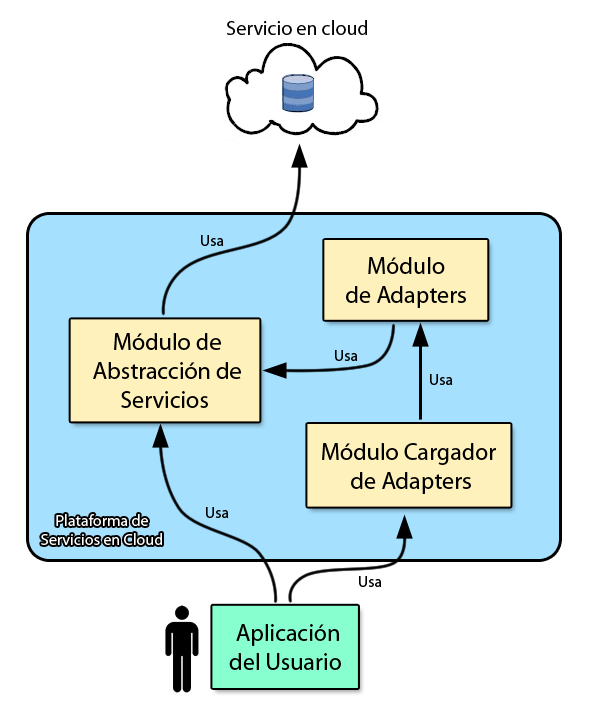
## 2. Enfoque

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 explicación será guiada por el desarrollo de un caso de ejemplo. Supongamos tener 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 2.1**.

​**Figura 2.1**

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 recodificar 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 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 cambios de 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.

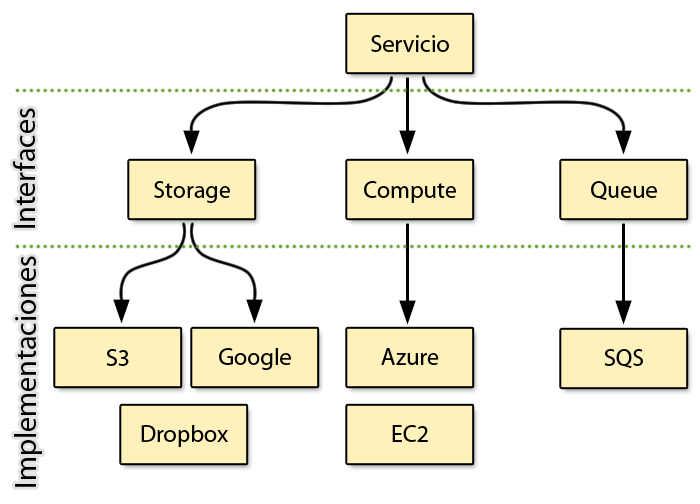
En base a la problemática anterior se presentan en la **Figura 2.2** los módulos base de nuestra plataforma.



**Figura 2.2**

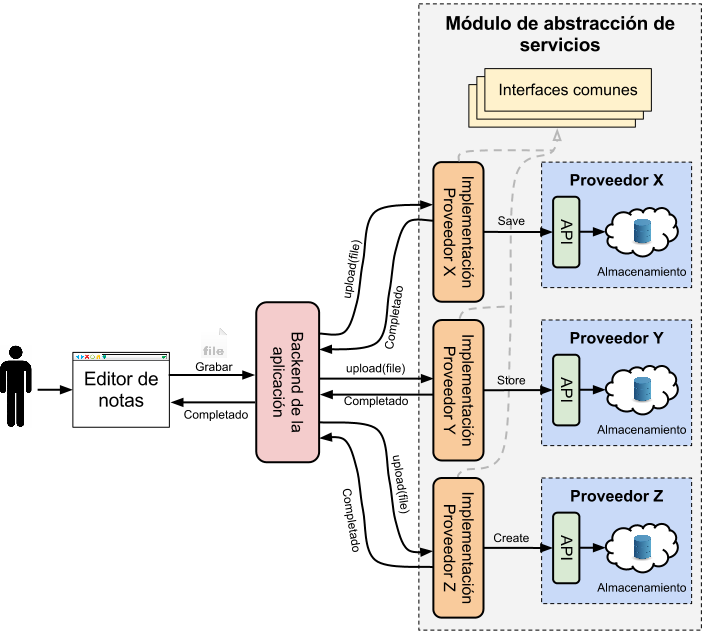
Dentro de la plataforma propuesta se pueden apreciar tres partes bien definidas. El módulo de abstracción de servicios será el encargado de proveer interfaces comunes a los servicios de cada proveedor (por ejemplo almacenamiento). Un desarrollador podrá usar estas interfaces para codificar una aplicación 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 de una aplicación existente. 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 los adaptadores necesarios en la aplicación del usuario de manera transparente. Cada módulo se analizara en detalle a continuación.

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 2.3**

Por cada tipo de servicio el usuario debe poder acceder a una interface única y simple. La **Figura 2.3** 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.



**Figura 2.4**

Continuando con el ejemplo de Cloud Recorder, la **Figura 2.4** presenta como seria la nueva estructura de la aplicación si se implementaran las mejoras del módulo de abstracción de servicios. Como se puede apreciar, se elimina la dependencia con cualquier API específica, que ahora son encapsuladas detrás de las interfaces de servicios del framework. A partir de este momento el desarrollador puede intercambiar la implementación específica del servicio que esté utilizando sin necesidad de mayores cambios en el código de la aplicación. Por ejemplo, si el desarrollador implementó la aplicación con Amazon S3 como sigue:

//Instanciación del servicio

StorageInterface service **=** **new** AmazonS3Service**(**…**)**

//Configuración del servicio

service**.**setEndpoint**(**…**)**

service**.**setAwsAccessKey**(**…**)**

service**.**setAwsSecretKey**(**…**)**

…

//Utilización del servicio

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

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

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

**}**

Una hipotética migración a Google Storage resultaría en el código siguiente:

//Instanciación del servicio

StorageInterface service **=** **new** GoogleStorageService**(**…**)**

//Configuración del servicio

service**.**setEndpoint**(**…**)**

service**.**setUsername**(**…**)**

service**.**setPassword**(**…**)**

…

//Utilización del servicio

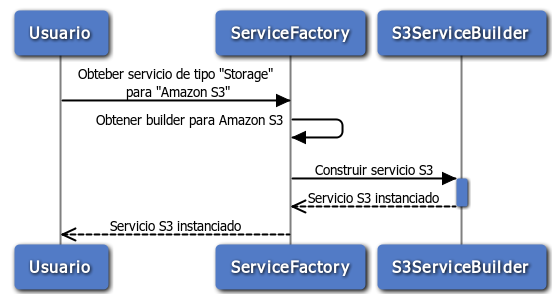
List archivos **=** service**.**upload**(**file**)**

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

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

**}**

Como ilustra el ejemplo, ya se ha podido eliminar el problema de del cambio entre proveedores, pero se puede ver 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. La idea de este punto de acceso se ve reflejada en la **Figura 2.5**.



**Figura 2.5 (Construcción de un servicio)**

Como se puede apreciar el código del usuario realizaría un pedido al factory especificando el tipo de servicio y el proveedor del mismo. 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>

<storageService class=**"GoogleStorageService"**>

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

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

</storageService>

<storageService class=**"S3StorageService"**>

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

<parameter key=**"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

StorageInterface service **=** 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

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

//Utilización del servicio

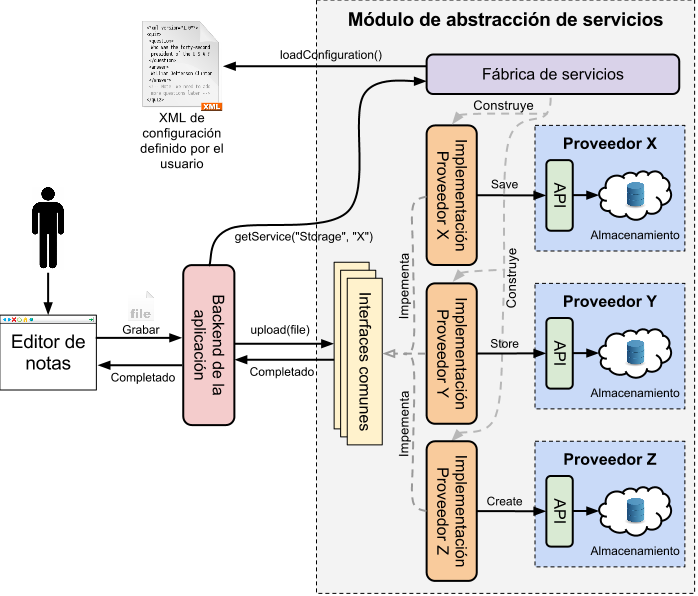
List archivos **=** service**.**upload**(**file**)**

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

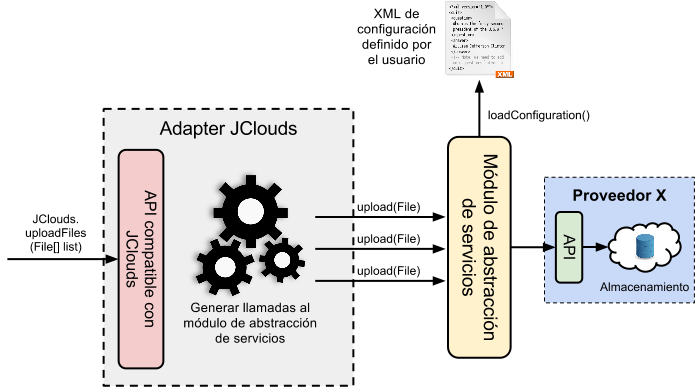
service**.**download**(**archivo**)**

**}**

Ya estamos en condiciones de agregar este nuevo mecanismo a Cloud Recorder. Como se puede apreciar en la **Figura 2.6**, el desarrollador ya no necesita comunicarse directamente con las implementaciones de cada proveedor, ya sea para configurarlas o utilizarlas. Todo lo referido a instanciación y configuración queda encapsulado en la “Fabrica de servicios” y el XML de configuración que el desarrollador define.

​**Figura 2.6**

A partir de estas 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 al 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 testeada. 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 utilizara un archivo XML que defina que clases del framework objetivo 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>

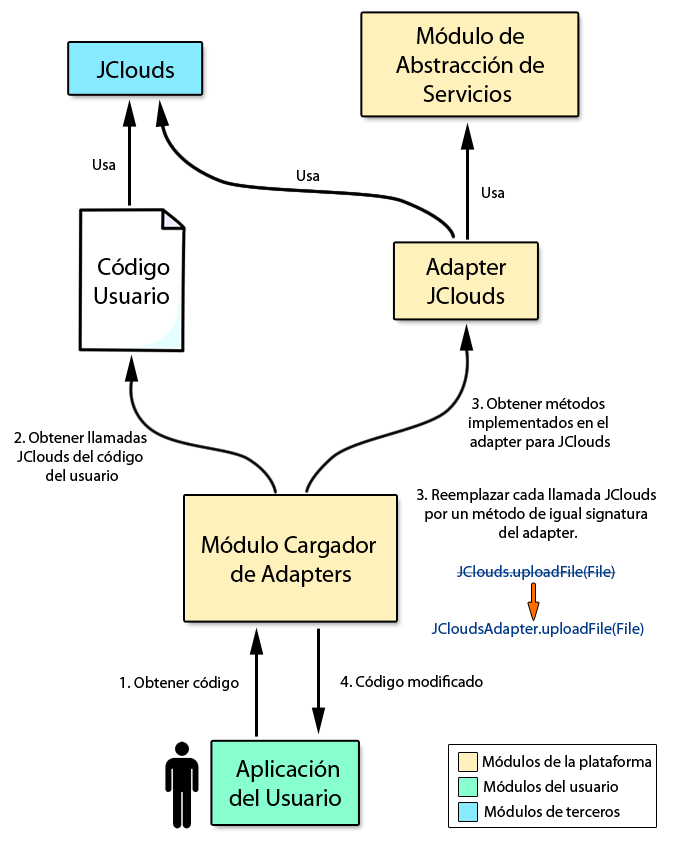
<dstClass>**JCloudsAetherFrameworkAdapter**</dstClass>

</classException>

</ClassLoaderConfig>

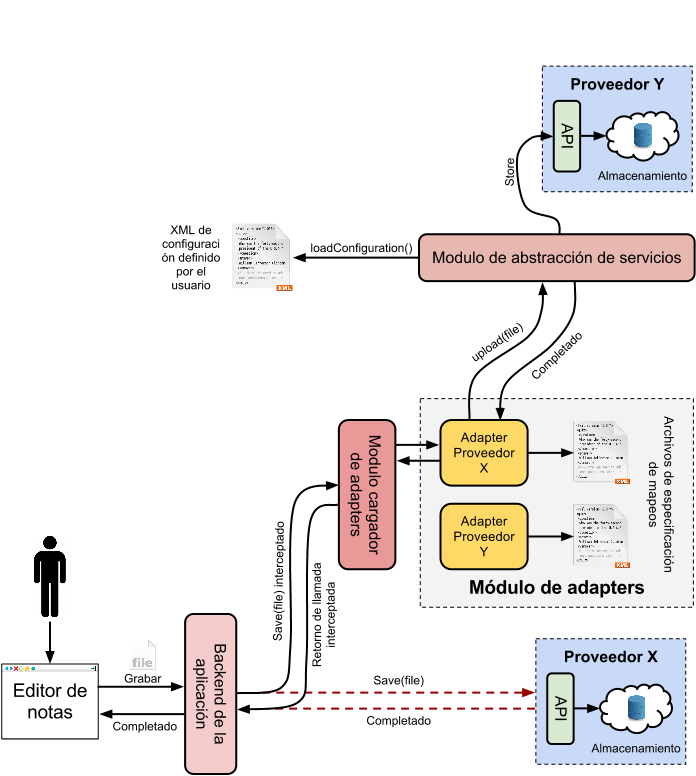
Teniendo el modulo que realizara la traducción de llamadas surge la pregunta: ¿Cómo trabajan estos adapters con la aplicación del usuario? Hasta ahora lo único que hicimos fue diseñar un intérprete para traducir los métodos de un framework a otro, pero no analizamos cómo se haría esto de manera cómoda 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 en un 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 su código. De esta manera el usuario solo debe tener conocimiento del módulo cargador y como 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 **Figura** **2.9** 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 aplicación original. 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.