Adaptación flexible de aplicaciones de base de datos mediante wrappers

Autor/Alumno: Aranda Granda, Francisco Javier.

Director: Lloret Gazo, Jorge.

Máster en Ingeniería de Sistemas e Informática

Programa Oficial de Posgrado en Ingeniería Informática

Septiembre de 2010

Adaptación flexible de aplicaciones de base de datos mediante wrappers

Resumen

Objetivos

Cuando es modificado el esquema de BD, los programas que acceden a éste frecuentemente deben ser modificados. En ocasiones la modificación de los programas es complicada y costosa.

Los objetivos generales de esta línea de investigación son 1) llegar a un mejor conocimiento del problema y de las soluciones existentes, y 2) desarrollar nuevas soluciones al problema que puedan ser ventajosas.

Conclusiones

Tras examinar diversas propuestas, se ha desarrollado una solución abierta con la esperanza de que otras partes puedan aplicarla sobre escenarios alternativos, y tengan un punto de apoyo sobre el que desarrollar soluciones ampliadas o alternativas.

A diferencia de otras propuestas vistas hemos preferido: 1) Dejar de lado el análisis estático del código fuente, y realizar una recopilación y análisis de la información obtenida de la ejecución del programa. 2) Dejar de lado (de momento) la idea de realizar una traducción automática basada en una especificación del cambio del esquema, y depender en los técnicos para definir y corregir la adaptación concreta según su criterio y experiencia.

El mecanismo de adaptación implementado es un wrapper JDBC que permite 1) obtener una información muy detallada del acceso a BD y 2) traducir dichas operaciones mediante matchers que identificarían cada caso de sentencia SQL solicitada, y handlers que se encargarían de realizar el tratamiento correcto.

La implementación incluye un proceso rudimentario para analizar las operaciones de acceso registradas. Este análisis produce un resumen de las sentencias SQL ejecutadas, así como de las dependencias entre las mismas (que es necesario para adaptaciones complejas). Con esta información se elabora una configuración base para el mecanismo de adaptación.

En el trabajo presentado se realiza una visión global del problema y se muestra de forma práctica cómo se soluciona un caso de estudio no trivial con el wrapper implementado.

Índice de Contenidos

1 Introducción

Frecuentemente los programas almacenan o recuperan información de las bases de datos. Denominamos *esquema* a la estructura de estos datos almacenados. Dicho esquema puede cambiar a lo largo del tiempo. Con motivo de estos cambios los programas pueden necesitar ser modificados para continuar funcionando correctamente. A este proceso de cambio lo llamamos *Evolución de programa de Bases de Datos*.

Así en nuestro caso de estudio, tenemos una BD relacional que contiene una Wiki simplificada. Su esquema es modificado de tal manera que las referencias que se hacían a tablas y columnas desde el programa ya no son válidas, y éste debe ser modificado para poder funcionar correctamente.

El problema de la evolución de programas de BD no es nuevo. Ya en el año 82 Schneiderman y Thomas [5] proponían definir el cambio de la BD como una secuencia de pasos elementales, en función de los cuales se podrían determinar los cambios sobre las operaciones de acceso a BD del programa. Posteriormente han aparecido distintas propuestas que iremos desgranando.

Sin embargo en opinión de algunos [2] estos esfuerzos resultan insuficientes (*"The lack of support, in terms of methods and tools, in database maintenance and evolution is testified by the low number of scientific references that can lead to practical application"*). Sería cuestionable si el número de referencias científicas es o no escaso. En cualquier caso a día de hoy no encontramos herramientas de código abierto o comerciales que permitan realizar una adaptación como la del caso de estudio. Si bien algunos autores afirman haber implementado determinadas herramientas, estas no están disponibles para el público.

Desconocemos los motivos por los que estas herramientas para la evolución de programas no están disponibles. Consideramos que sería útil la disponibilidad de una solución de código abierto. Esta solución permitiría que otros investigadores o desarrolladores, caso de interesarse en este tema, no tengan que partir de cero.

Concretamente el framework desarrollado facilita dos cosas: 1) La obtención de información real sobre los accesos a BD del programa, obtenidos a través de la ejecución del mismo; y 2) la adaptación/traducción de los accesos del programa (que no habría sido modificado) a la BD (donde el esquema ha sido modificado).

Esta memoria se estructura de la siguiente manera: Tras la actual introducción, en la que hemos descrito el problema de la evolución de programas de BD, en la Sección 2 describimos brevemente las alternativas planteadas por distintos autores. Tras ello planteamos nuestra solución, indicando en primer lugar, en la Sección 3 el caso de estudio sobre el que hemos trabajado, y describiendo en la Sección 4 qué hace el *wrapper* implementado para abordar dicho caso de estudio, comprendiendo aspectos como la adaptación en sí (4.1), y la obtención de la información necesaria para realizarla (4.2 y 4.3). Igualmente se comentan las limitaciones y los elementos que sería interesante desarrollar (4.4).

2 Alternativas de adaptación y trabajos relacionados

En esta sección se comentan distintas opciones que existen para realizar las tareas de la evolución de los programas de BD. Junto con cada opción aparece la reseña de algunos trabajos relevantes.

Un aspecto relevante para la realización de la evolución del programa es conocer el estado inicial del sistema, tanto del esquema de BD como de los accesos que realiza el programa.

Hainaut et al ([6][2]) comentan sobre la posibilidad de realizar una ingeniería inversa del esquema utilizando *DB-MAIN*. El esquema conceptual obtenido se tomaría de base para los siguientes pasos de la evolución.

La herramienta *PRISM* de Curino et al [11] utiliza un meta-esquema donde se almacenaría información de gestión variada.

En la bibliografía encontrada se ha encontrado predominantemente un enfoque de analizadores estáticos [2][4]. Este análisis examina las dependencias de datos dentro del código fuente, tratando de averiguar qué sentencias SQL se ejecutan en cada llamada al API de BD. Sería cuestionable si el análisis estático es preferible. En nuestro trabajo hemos preferido un análisis dinámico, donde la información se obtiene ejecutando el programa.

Para la obtención de información dinámica existen algunas implementaciones de Wrapper JDBC similares a la utilizada, como *P6Spy*[10], *ASPY* [7]o *Elvyx* [8]. Sin embargo estas herramientas parecen estar más orientadas a la detección de problemas como queries que consumen demasiados recursos o provocan interbloqueos. Nuestra implementación permite capturar información adicional (como la pila de llamadas con la ubicación de las llamadas, o los valores de los parámetros JDBC), que pueden ser relevantes para la traducción de las sentencias.

Para realizar una evolución del esquema y del programa, es conveniente conocer también acerca de la naturaleza de la modificación a realizar. El cambio se suele definir como una secuencia de modificaciones elementales en el esquema (como copiar una tabla, renombrar una columna, etc). Tanto Hainaut et al [2] como Curino et al [1] definen sus propios repertorios de operaciones de modificación de esquema (SMO). Asimismo postulan sobre aspectos como la reversibilidad de estas operaciones. Según este modelo, la modificación sobre las operación de acceso a BD del programa sería igual a la acumulación de las modificaciones correspondientes a cada SMO de la secuencia aplicada sobre el esquema.

No obstante en ocasiones hay modificaciones peculiares en el esquema que no parecen tener un SMO asociado, o resulta complicado determinar la modificación sobre el programa que corresponde para un SMO concreto. Considerando que este enfoque no puede llevar en un futuro inmediato a una solución completa (o que por lo menos solucionara satisfactoriamente el caso de estudio), hemos preferido delegar en los técnicos la responsabilidad de poner a punto la adaptación concreta.

Curino et al [1], adicionalmente obtienen y utilizan un mapeo denominado DED (*Disjuntive Embedded Dependencies*), que indica la correspondencia entre los datos del esquema anterior y el esquema posterior al cambio. El mapeo directo indicaría cómo se rellenan los datos del esquema nuevo a partir de los datos del esquema antiguo, mientras que un mapeo inverso ayudaría para la traducción de las sentencias de consulta (que deberían obtener los datos antiguos a partir de los datos nuevos).

Hick y Hainaut [2]plantean la posibilidad de un "*enfoque híbrido*", según el cual una herramienta examinaría las versiones del esquemas anterior y posterior al cambio y deduciría los cambios efectuados. Sin embargo existe la posibilidad de cambios ambiguos, con lo que esta solución tampoco sería completa.

Respecto a la modificación general, la aproximación más inmediata es modificar directamente el código del programa para realizar las operaciones de acceso según corresponda con el nuevo esquema. Otras soluciones alternativas conocidas son 1) definir un esquema de BD que simule el anterior mediante *vistas SQL*, o 2) introducir en el programa un *wrapper* del API de acceso a BD que se encargue de traducir las operaciones solicitadas.

La opción de definir *vistas SQL* es planteada por Hick y Hainaut [2] y por Curino et al [1]. Estas *vistas SQL* que simularían las tablas y las vistas del esquema anterior a partir de las tablas/vistas del esquema posterior. Concretamente en la demo de *PRISM* [11] se generar un script SQL para definir estas vistas para las modificaciones definidas por el usuario desde un esquema inicial. El uso de vistas puede no ser una solución adecuada si necesitamos hacer operaciones de actualización.

El uso de tecnologías *wrappers*, planteado entre otros por Thiran et al [6] supope adaptar el programa parcheándolo con un componente que se interpondría bajo la forma de la propia API de acceso a datos. Mediante wrappers podemos no solo hacer actualizaciones en los datos, si no que se podría incluso cambiar el tipo de persistencia o BD utilizado. Sin embargo esto resulta ligeramente intrusivo en el programa, y no tiene por qué ser sencillo de implementar y configurar.

El uso de wrappers o de vistas es una solución superficial. Estos tipos de solución pueden ser adecuados si cambia la forma en la que los datos son almacenados, pero la naturaleza de estos permanece inalterada (por ejemplo si hacemos una normalización, o renombramos una columna). Pero en caso de que el cambio del esquema signifique un cambio conceptual en los datos (como la introducción de nuevos datos, o la alteración de las asociaciones existentes) será necesario cambiar el programa en profundidad para que todo el código relacionado se adapte a la nueva naturaleza de los datos (a no ser que los nuevos datos puedan ser ignorados por el programa).

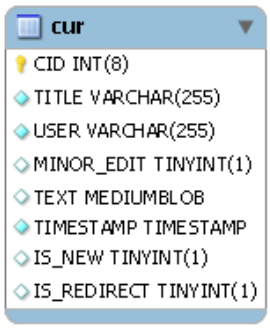
Por ejemplo si en una tabla de personas se introduce una nueva columna (por ejemplo teléfono), si dicho dato debe reflejarse en las pantallas de introducción y de consulta de datos, entonces una modificación superficial en el acceso a datos no es suficiente. Si por lo contrario dicho dato puede ser ignorado por el programa (no resulta relevante por el momento), posiblemente no haya que modificar ni el acceso a datos (no hace falta consultar la columna, y quizás ni establecer ningún valor al insertar o actualizar).

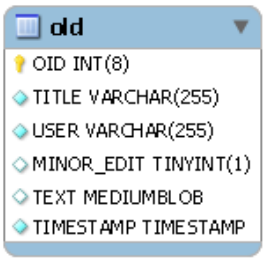
Karahasanovic [3] propone e implementa una visualización del impacto del cambio sobre un programa a través de un grafo de dependencias de las clases, métodos y propiedades de un programa Orientado a Objetos. Esto ofrecería mejor información a los técnicos que un simple listado de los ficheros y las líneas donde ser referencian tablas y columnas afectadas por el cambio.

La modificación de programas de BD es un tema de interés dentro de las metodologías ágiles. Fowler y Saladage [9] describen una experiencia práctica. Llegan a comentar la implementación de wrappers para adaptar programas puntuales a esquemas modificados, pero sin concretar más. Parece que la preocupación principal dentro de las metodologías ágiles es gestionar los esquemas como un elemento más dentro del control de versiones y la integración continua, controlando la correcta propagación de los cambios de BD entre distintos entornos de desarrollo/producción. Esto se complica ante la posibilidad de que existan distintas ramas de desarrollo, o de que se quiera comparar y revertir cambios.

3 Descripción del caso de estudio

En este trabajo nos basamos en el ejemplo utilizado por Curino et al [1] para demostrar el funcionamiento de PRISM. Este ejemplo está basado en la evolución del esquema de MediaWiki, entre las versiones que identifica como 41 y 42 (que corresponden con las revisiones 6696 y 6710 del repositorio). Los estados inicial y final del cambio se representan en las figuras 1 y 2.





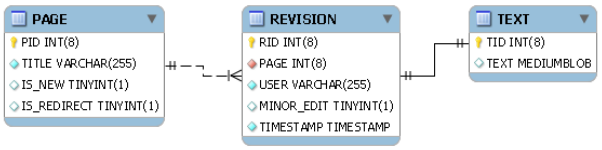
CUR

CID	TITLE	IS_NEW	TEXT	MINOR_EDIT
1	Mercurio	1	Primer planeta	0
3	Venus	0	Segundo planeta	1

OLD

OID	TITLE	TEXT	MINOR_EDIT
2	Venus	Segundo	0

Figura 1: Esquema y datos de ejemplo v.41



PAGE

PID	LATEST	TITLE	IS_NEW
1	1	Mercurio	1
3	3	Venus	0

REVISION

RID	PAGE	MINOR_EDIT
1	1	0
2	3	0
3	3	1

TEXT

TID	TEXT
1	Primer planeta
2	Segundo
3	Segundo planeta

Figura 2: Esquema y datos de ejemplo v.42

En este ejemplo nos centramos en la información elemental de una página (el título de la página, el texto contenido en la misma, quién la editó y cuándo se hizo), y en la historia de revisiones de ésta. Para ilustrar el ejemplo utilizamos un juego de datos mínimo: Mostramos dos páginas "*Mercurio*" y "*Venus*". *Mercurio* solamente tiene la versión actual. De *Venus* existe además una versión anterior en la tabla *OLD*.

El esquema de BD inicial partimos de la tabla *CUR* (versiones actuales de páginas) y de la tabla *OLD* (versiones antiguas de páginas). En este esquema posterior encontramos la tabla *PAGE*(información de una página, común a todas las versiones de la misma), la tabla *REVISION*(información específica de cada versión de cada página), y la tabla *TEXT* (textos de las versiones de las páginas). La versión actual de una página se

obtiene mediante la referencia de `PAGE.LATEST→REVISION.RID`(1 página, 1 revisión actual). La página asociada a una revisión se obtiene mediante `REVISION.PAGE→PAGE.PID`(N revisiones para cada página). El texto de una versión de la página se obtiene mediante `REVISION.RID→TEXT.TID`(1 revisión,1 texto asociado).

Para ensayar la adaptación de programas se ha implementado en Java una aplicación Wiki mínima (µWiki), que debe ser modificada conjuntamente con el esquema de BD. El código de acceso a datos que contiene es un ejemplo de cómo se programaría en Java utilizando la API JDBC básica. Asimismo se ha implementado sobre el código del programa una prueba unitaria con *DBUnit* que lo mismo que nos sirve para comprobar el correcto funcionamiento del acceso a datos, nos sirve para que el *wrapper* obtenga la información necesaria para realizar la adaptación.

Dentro de la aplicación se ejecutan algunas sentencias cuya traducción no supone una especial dificultad, pero hay otras que merecen examinarse con mayor detenimiento.

En un caso sencillo tendríamos la sentencia de consulta de una página actual:
```
select CID, TITLE, USER, TEXT, TIMESTAMP from CUR where TITLE=?
```
Que en la versión 42 se ejecutaría como
```
select PAGE AS CID, TITLE, USER, TEXT, TIMESTAMP from REVTEXT r,PAGE p,TEXT t
where r.RID=p.LATEST and r.RID=t.TID and p.TITLE=?
```

Cabe observar que 1) la consulta se convierte en una *join* de varias tablas, según las asociaciones descritas (`r.RID=p.LATEST`,`r.RID=t.TID`); 2) el símbolo "?" en JDBC identifica un parámetro de *PreparedStatement*; y 3) renombramos la columna una resultado (`PAGE AS CID`) para que el programa pueda seguir identificando igual dicha columna.

Sin embargo la operación de inserción/actualización entraña mayor dificultad. Examinemos las sentencias de la versión 41:

```
> insert into OLD (OID, TITLE, USER, TEXT, TIMESTAMP) select CID, TITLE, USER, TEXT,
 TIMESTAMP from CUR where TITLE=$title;
> delete from CUR where TITLE=$title;
> insert into CUR (TITLE, USER, TEXT, TIMESTAMP) values($title,$u,$tx,$tm);
```

Observar que 1) La modificación se realiza moviendo la fila de la versión anteriormente actual de la tabla *CUR* a la tabla *OLD*, y a continuación introduciendo la nueva versión en *CUR*. 2)Hemos representado a los parámetros como *$parametro* para identificarlos mejor (En el código Java correspondiente aparecerán como "?").

en versión 42 del esquema la modificación se realiza como:

```
> insert into REVISION (USER, TIMESTAMP, PAGE) values($u,$tm, 0);
 (returning rid into $rid)
> insert into PAGE (LATEST, TITLE) values($rid, $title) on duplicate key update
 LATEST=$rid;
 (returning pid into $pid)
> insert into TEXT (TID, TEXT) values ($rid, $tx);
> update REVISION set PAGE=$pid where RID = $rid;
```

Observar que la modificación es entonces algo diferente: 1) El movimiento de la versión de la página de *CUR* a *OLD* deja de tener sentido, y las sentencias asociadas (*insert* y *delete* iniciales) no tienen correspondencia en la versión 42. 2) La inserción de una nueva versión de página en *CUR* se traduce en varias sentencias para insertar/actualizar en las tablas donde han quedado distribuidos los datos. 3) Debemos preocuparnos de mantener la coherencia entre las claves, concretamente 4) obtenemos las claves de las filas insertadas (`returning rid into $rid`), que necesitamos en las operaciones siguientes. También sucede que 5) hay operaciones de inserción/actualización (El `insert into PAGE`, donde no conocemos *a priori* si la fila existe).

Este ejemplo de evolución de programa de BD es adecuado porque 1) Ilustra un cambio

de esquema que resulta difícil de especificar mediante primitivas de modificación de esquema, en el cual sería preferible realizar un tratamiento personalizado a un tratamiento generado por reglas automáticas, y 2) porque aunque implica cambios radicales en el esquema, no implica cambios en los datos del programa, de modo que se puede solucionar 'limpiamente' con una modificación mínima..

4 Descripción del wrapper desarrollado

La solución desarrollada consiste en un *wrapper* del API JDBC, que permite realizar dos tareas diferentes de la evolución de los programas de BD: 1) La obtención de información sobre las operaciones de acceso realizadas y 2) la traducción de las operaciones de acceso de un programa no actualizado para que funcione correctamente sobre un esquema actualizado. Aunque ambas tareas se pueden realizar de forma completamente independiente, se busca cierta integración entre ellas: La información obtenida de los accesos es de sumo interés para realizar la posterior traducción.

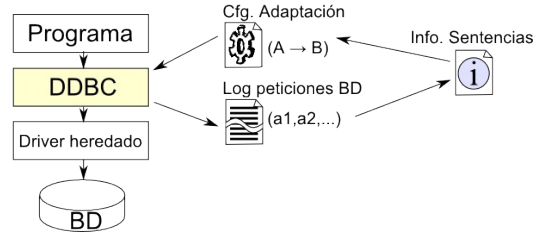


Figura 3: El wrapped DDBC permite obtener información y adaptar el acceso a BD

En la Figura 3 podemos observar cómo el wrapper DDBC (*Decorator DB Connection*) intermedia en las peticiones a BD del programa (que inicialmente iban dirigidas al driver heredado).

El log de peticiones registra los accesos a datos A realizados (a_1,a_2, …) antes de la modificación del esquema. Por otro lado, para realizar la traducción de los accesos (A→B) se necesita una especificación de los accesos a datos B que se deben realizar en lugar de los accesos A.

Además del wrapper se ha desarrollado una utilidad para resumir el log de acceso (que puede resultar verboso y redundante), y analizar las secuencias de peticiones y las dependencias datos entre ellas(necesarias para algunas traducciones no triviales). El informe de sentencias obtenido es de utilidad para orientar a los técnicos qué puntos del programa deben revisar, y para elaborar un borrador de la configuración de adaptación.

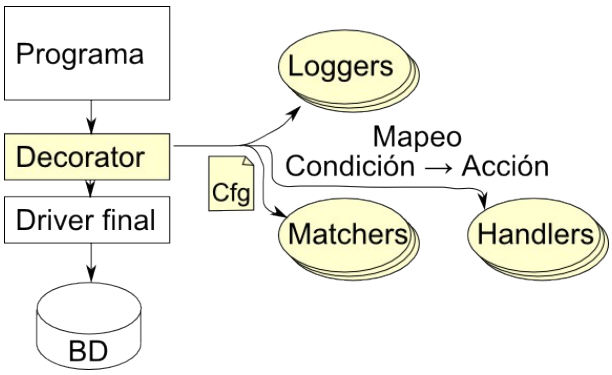


Figura 4: Arquitectura del wrapper

En la Figura 4 podemos observar cómo el *decorator* (un wrapper con capacidad para encadenarse con sucesivos decoradores, que aportan comportamiento adicional sobre la interfaz delegada) tiene un funcionamiento determinado por su configuración, y por sus componentes subordinados (*loggers*, *matchers* y *handlers*). El usuario no solo puede seleccionar y configurar sus componentes, si no que está previsto que pueda utilizar sus propios componentes personalizados si así le conviene. Con esta modularidad se busca que el wrapper tenga toda la flexibilidad y potencialidad para cualquier adaptación técnicamente posible.

Los componentes *logger* se encargan de registrar los accesos a datos realizados. Puede

desactivarse (y es esperable que se haga una vez adaptado el programa) para reducir carga. Dentro del intérprete/traductor, para cada sentencia SQL que se pide ejecutar, se busca a qué regla de traducción corresponde mediante los *matchers*, que comprueban si la sentencia verifica la condición de activación de la regla, en cuyo caso se ejecuta el *handler* asociado.

Tanto el programa del caso de estudio como el wrapper están escritos en Java. Los detalles sobre la configuración y los módulos, y el proceso de adaptación pueden consultarse en el manual de usuario de DDBC (anexos).

En toda esta documentación nos centramos en el "qué" hace el wrapper y otros desarrollos, y no en el "cómo" lo hace. Aunque hay aspectos técnicamente interesantes como la existencia de un generador de código que preserva modificaciones introducidas, o cómo se configuran elementos definidos dinámicamente, quizás ello se sale del tema tratado (que es por qué y para qué necesitamos un wrapper, y no cómo lo construimos).

4.1 Traducción de las operaciones de acceso

Aunque en el proceso de adaptación, la elaboración y utilización de la configuración de adaptación van en último lugar, conviene explicarlas primero porque son la finalidad. Y la obtención de información y su proceso, aun anteriores en el proceso de adaptación, están orientadas a un propósito que es mejor conocer al principio.

La traducción de las sentencias SQL es realizada mediante un intérprete, el cual sigue una *especificación de adaptación*. La *especificación de adaptación* contiene un conjunto de reglas con la forma condición-acción. Las condiciones son comprobadas mediante los *matchers*. Actualmente las condiciones se comprueban secuencialmente. En caso de que la condición sea afirmativa, se ejecutará la operación de acceso mediante el *handler* asociado, y se dejarán de comprobar las condiciones posteriores. En caso de no se verifique ninguna condición, se ejecutará un tratamiento/*handler* por defecto.

En un cambio en el esquema de BD, es predecible que solo cambie una parte del mismo, de modo que la mayor parte de las sentencias permanecerán inalteradas. De las sentencias SQL alteradas cabe esperar que las consultas (sentencias *SELECT*) se solucionen ejecutando en su lugar otra consulta equivalente. En cambio las operaciones de actualización pueden suponer una mayor dificultad.

A continuación haremos un recorrido por las situaciones con las que esperamos encontrarnos en la traducción, y los medios previstos para solucionarlas: Ya sea con los componentes *matchers* y *handlers* incluidos actualmente en DDBC (los basados en literales y en expresiones regulares, y los especiales para permitir ejecutar sin alteración, o provocar un fallo). También veremos acerca de la utilización de *handlers* personalizados, y del uso de una variable de estado para casos en los que una misma sentencia tenga distintas traducciones según el contexto.

4.1.1 Traducción literal

En el caso de traducción más sencillo, encontramos que una de las sentencias que ejecuta recurrentemente el programa es siempre literalmente igual. Por ejemplo, para obtener un listado de las páginas disponibles podríamos ejecutar:

```
select TITLE from CUR;
```

que en la versión 42 del esquema se ejecutaría como

```
select TITLE from PAGE;
```

Para este tipo de traducción tenemos un *matcher* literal que se activan cuando la sentencia es literalmente la especificada, y un *handler* literal que ejecuta la sentencia indicada en la especificación en lugar de la solicitada por el programa. Así el ejemplo mostrado se podría traducir mediante la siguiente regla:

```
<mapping id="select title from cur">
  <matcher type="equals">select TITLE from CUR</matcher>
  <handler type="literal">select TITLE from PAGE</handler>
</mapping>
```

Cabe observar que muchas sentencias incorporan valores variables. Por ejemplo si tuviéramos una clausula "`where USER='user1'`" la identificación literal no sería adecuada si tuviéramos múltiples valores posibles para *USER*. Sin embargo en JDBC existe la posibilidad (incluso la recomendación) de ejecutar las sentencias como *PreparedStatement*, identificando los valores variables con el token *placeholder* (?), y estableciendo su valor con los métodos `setInt(idx, i)`, `setString(idx, s)`, etc. Esto permite que aunque consultemos y actualicemos utilizando valores distintos, las sentencias sean recurrentemente las mismas.

4.1.2 Traducción mediante expresiones regulares

En ocasiones podemos encontrarnos que una misma sentencia tiene pequeñas variaciones, como el valor de una variable, que aparece literalmente en la sentencia, o pequeñas variaciones en la lista de columnas del *resultset*, o en las condiciones del *WHERE*.

En estos casos resulta conveniente identificar las sentencias según el patrón que siguen, y poder hacer las modificaciones identificando los bloques que queremos que se ejecuten tal cual en la expresión de sustitución.

Así por ejemplo en las consultas recurrentes según el patrón
`select CID, * from CUR where *`
querríamos ejecutar en la versión 42 una consulta con la forma
`select PID as CID, * from PAGE P, REVISION R, TEXT T WHERE P.LATEST=R.RID AND R.RID=T.TID AND *`

La *join* realizada viene explicada en un ejemplo similar en la sección 3 sobre el caso de estudio. Así como el renombrado de CID (si se ejecuta un `getString("CID")` para recuperar la columna, no nos sirve que pase a llamarse PID).

Esta regla de traducción se podría representar con esta especificación:

```
<mapping id="select cur where ANY">
  <matcher type="regexMatcher">select CID, (.*) from CUR  where
 (.*)</matcher>
  <handler type="regexHandler" >
  <rx-rule>
    <rx-match>select CID, (.*) from CUR where (.*)</rx-match>
    <rx-subst>select PID AS CID, $1 from PAGE p, REVISION r, TEXT t
 where p.LATEST=r.RID and r.RID=t.TID and $2</rx-subst>
  </rx-rule>
</handler>
</mapping>
```

Observar como el wildcard "*" se representa con la expresión regular ".*" (cualquier carácter [.] de 0 a N veces), que los "capturing blocks" se identifican con paréntesis en la expresión de match, y se correlacionan con su número de orden ($1, $2) en la expresión de sustitución.

4.1.3 Uso de handlers especiales

Tenemos *handlers* que no hacen nada y *handlers* que hacen que suceda una excepción en el programa. Son chocantes en el sentido de que no ponemos un wrapper para no hacer nada, o hacer que fallen las operaciones.

Sin embargo, por un lado conviene especificar explícitamente las sentencias que no se han visto afectadas. Para estas se diría que se propague la petición al driver delegado sin alteración (*passThrough*). Es posible que aparezcan sentencias SQL que no habíamos previsto inicialmente. Un tratamiento plausible sería intentar ejecutar dichas sentencias inesperadas tal cual (con suerte no estarán afectadas), pero mostrando un warning en un log. Otra opción sería hacer que fallaran (en un mal caso una sentencia inesperada podría "funcionar" corrompiendo datos).

Respecto al *handler* de fallo (*fail*), puede ser útil tanto para sentencias inesperadas, como para aquellas sentencias afectadas para las que todavía no hemos trabajado una traducción, o que están definitivamente "rotas", y como tal se expresan e interpretan.

De esta manera podríamos representar las reglas *otherwise* (sentencias que no concuerdan con ninguna condición del conjunto):

```
<otherwise>
  <handler type="passThrough" unexpected="true" />
</otherwise>
```

o

```
<otherwise>
  <handler type="fail">Sentencia inesperada</handler>
</otherwise>
```

4.1.4 Uso de handlers personalizados

Es posible, particularmente con las operaciones de actualización, que los mecanismos de traducción sencillos, mostrados hasta ahora, sean insuficientes. En nuestro caso de estudio (ver secuencia de actualización descrita en la sección 3), encontramos sentencias ("`insert into old (...) select ... from cur`" y "`delete from cur`") que se anulan, mientras que "`insert into cur(...) values ...`" se desdobla en múltiples sentencias.

Crearse un *handler* personalizado significa crearse una implementación de la interfaz *StatementHandler* (Un buen punto de partida para tomar como modelo podría ser la clase *HandlerCmdEdit410in420* creada para el caso de estudio).

La clase del *handler* personalizado debe referenciarse en la configuración de adaptación, asignándole un nombre de tipo de *handler* (en este caso "*cmdEdit*").

```
<def type="cmdEdit" class="uwiki.evol.HandlerCmdEdit410in412" />
```

Posteriormente, se puede referenciar el *handler* personalizado en las reglas como uno más. No es necesario crear un *handler* personalizado para cada sentencia. En nuestro caso hemos optado por crear un *handler* para todas las sentencias de la secuencia de operaciones de actualización. También sería posible especificar parámetros para los *handlers* personalizados (aunque en nuestro caso no se necesita/está *hardcodeado*).

Este sería un ejemplo de regla de mapeo/traducción utilizando el *handler* personalizado:

```
<mapping id="delete cur where title">
  <matcher type="equals" fromState="cmdEdit-deleteCur">
     DELETE FROM cur WHERE title=?</matcher>
  <handler type="cmdEdit" toState="cmdEdit-insertCur" />
</mapping>
```

Aparecen los atributos *fromState* y *toState*, que veremos en el siguiente apartado.

4.1.5 Uso de variables de estado

Es posible que dependiendo del contexto una sentencia se traduzca de distintas maneras. En el caso de estudio la sentencia "`delete from CUR where TITLE=?`" se podría traducir como sentencias de eliminación de las tablas *TEXT* y *REVISION*. Pero si resulta que inmediatamente antes se ha hecho el "`insert into OLD (<columnas>) select <columnas> from CUR where TITLE=?`", que se traduciría como una inserción en *TEXT* y *REVISION* (justo lo contrario que el *DELETE*), entonces ambas sentencias se anulan.

Para poder traducir correctamente las sentencias en este tipo de situaciones, puede ser necesario poder detectar las dependencias de datos entre sentencias de la misma secuencia/transacción. Para ello en nuestro caso necesitaríamos una variable de estado (cuyo valor estableceríamos en función de la sentencias ejecutadas previamente), y posiblemente otras variables (como el título) para verificar que se cumple la dependencia (una *delete* con un título y una *insert* con otro título distinto serían independientes aunque se ejecutaran en la misma transacción).

El estado por defecto sería "*initial*". En la mayor parte de las operaciones se permanecería en este estado, pero en el caso de que hubiera una secuencia de operaciones con dependencia de datos, deberíamos definirnos una máquina de estados, identificando los estados intermedios que nos encontraremos, y las condiciones/transiciones entre estos estados. La información obtenida previamente en el proceso de adaptación debe de ayudar en esta tarea.

Así en este caso, a parte del estado inicial tendríamos el estado *cmdEdit-insertOld* (después de `insert into old ...`) y *cmdEdit-deleteCur* (después de `delete from cur ...`).

En algunos casos, gracias a la variable de estado, podremos determinar el tratamiento a realizar. Sin embargo es posible que el tratamiento no dependa (solo) de las operaciones previas, si no de las operaciones posteriores. Es posible que la acción sea no hacer nada, y esperar a realizar la operación correspondiente una vez que ante una petición posterior se resuelva la cuestión.

4.2 El informe de sentencias

Para elaborar el borrador de la especificación de adaptación, necesitamos conocer qué sentencias SQL se ejecutan en la aplicación, descartando las repeticiones, y abstrayéndonos de los datos diferentes que puedan aparecer en cada instancia de la sentencia.

Asimismo también es importante conocer qué sentencias están agrupadas dentro de una misma transacción SQL, y la dependencia de datos que existiría entre estas sentencias. Las dependencias de datos se deducirían a partir de la coincidencia del valor de los parámetros y los resultados de las sentencias ejecutadas. Conviene cotejar las coincidencias de sucesivas transacciones para descartar las coincidencias casuales y conservar las coincidencias sistemáticas.

Este análisis de las dependencias de datos implica que en la obtención de datos deberemos obtener y almacenar información no solo de las sentencias ejecutadas, si no de los parámetros y de los resultados.

La información de las sentencias ejecutadas también puede ser útil para informar a los programadores de los puntos del programa que deben revisar ante determinado cambio en el esquema. Con tal finalidad deberían registrarse las ubicaciones del código desde donde se piden las sentencias correspondientes. En Java, gracias a la información de depuración es sencillo conocer la clase, el método y la línea desde donde se realiza un acceso determinado.

Para generar el informe de sentencias se utiliza la clase Java *MainGenSr*, indicándole el fichero de entrada (log) y el fichero de salida (informe de sentencias).

```
java ddbc.tool.sr.MainGenSr test-log.xml test-sr.xml
```

Este sería un extracto del informe de sentencias generado:

```
<statementReport>
  <statements>
    <statement stmtId="SR-stmtC0000">
      <normSql>select c.cid, c.title, c.user, c.minor_edit, c.text,
 c.timestamp, c.is_new, c.is_redirect from cur c order by c.title
 asc</normSql>
      <location>
        <f class="org.apache.commons.dbcp.DelegatingConnection"
 m="prepareStatement" f="DelegatingConnection.java" l="248"/>
        <f class="uwiki2.WData" m="selectCurWhere" f="WData.java"
 l="83"/>
        <f class="uwiki2.WData" m="listCurrent" f="WData.java"
 l="44"/>
                  (more stack frames).................
      </location>
                  (possibly more locations)..........
    </statement>
                  (more statements)..................
  </statements>
  <sequences>
    <sequence seqId="SR-seqC0003">
      <stmtRefs>
        <stmtRef idRef="SR-stmtC0003"/>
        <stmtRef idRef="SR-stmtC0004"/>
        <stmtRef idRef="SR-stmtC0005"/>
      </stmtRefs>
      <dataDependencies>
        <group>s1p1,s2p1,s3p1</group>
 (possibly more sequence data dependencies).........................
      </dataDependencies>
    </sequence>
                  (possibly more sequences)..........
  </sequences>
</statementReport>
```

En primer lugar se puede observar la enumeración de las clases de sentencias encontradas. Para cada clase de sentencia se indican las ubicaciones (*location*) desde las que se ejecuta, mediante los *stack frames*.

A continuación aparecen las secuencias de sentencias localizadas. Se indica las dependencias de datos encontradas (una sentencia SQL puede traducirse de distintas maneras en función de las sentencias relacionadas). Cada elemento coincidente se describe con el formato `s<n>{p<n>|r<n>|k<n>}` indicando el número de sentencia (*s*), y la categoría y número de orden del dato indicado, pudiendo ser un parámetro (*p*), una

columna de *resultset* (*r*), o una clave retornada (k). Así `s1p1,s2p1,s3p1` indica que el primer parámetro de la primera, segunda y tercera sentencias (correspondientes al título de la página modificada) coinciden.

4.3 Obtención del log de acceso a BD

En nuestro caso el objetivo del log obtenido es la adaptación del programa, para lo cual es importante obtener el valor de los datos de parámetros y resultados (para determinar las dependencias de datos) y las pilas de llamadas de las llamadas a BD (para facilitar la intervención de los programadores). Esta información puede implicar un volumen elevado de información.

Otros wrappers dedicados a la escritura de logs (como *Elvyx* [8]) no contienen esta información. Su cometido es diferente: Averiguar qué sentencias son problemáticas (invocadas a menudo, duración excesiva, tráfico excesivo). Parece probado y optimizado para estar habilitado permanentemente en aplicaciones web.

Otros wrappers encontrados tienen menor interés. *P6Spy* [10] es el antecesor de *Elvyx*, y se cita en algún artículo de investigación. Sin embargo carece de mantenimiento desde 2003, y muchas de las librerías de las que depende para compilar son obsoletas y difíciles de encontrar. De *ASPY* [7] no hay código fuente en abierto.

La obtención de los logs de DDBC para la adaptación del programa se espera que sea una tarea puntual. Es necesario que se registren todas las sentencias SQL y todas las transacciones distintas ejecutables por el programa, y que se repita alguna vez variando los datos para evitar que una coincidencia casual de datos se interprete como una dependencia real. De esta manera, para obtener la información necesaria es necesario ejecutar un repertorio de pruebas equivalente a lo que sería un acceso a datos.

Aunque dicho repertorio de pruebas se puede ejecutar "manualmente", lo recomendable sería disponer de una prueba unitaria. En nuestro caso disponemos de una prueba unitaria basada en *DBUnit* (una extensión de *JUnit* que permite la inicialización, verificación y limpieza del esquema probado).

El log obtenido se almacena en un fichero local con el siguiente aspecto:

```
<custom conn="CONN0000" src="CONN0000" tag="CONN_new"
          start="20100507.115324.453" d="0">
  <f class="ddbc.cpn2.impl.CpConnection" m="&lt;init&gt;"
          f="CpConnection.java" l="48"/>
                  (more stack frames).................
</custom>
<call conn="CONN0000" src="CONN0000" m="CONN_setAutoCommitB"
          start="20100507.115324.468" d="16">
  <p>true</p>
</call>
<call conn="CONN0000" src="CONN0000" m="CONN_prepareStatementS"
          start="20100507.115324.484" d="94">
  <f class="org.apache.commons.dbcp.DelegatingConnection"
 m="prepareStatement"
          f="DelegatingConnection.java" l="248"/>
                  (more stack frames).................
  <p>SELECT c.cid, c.title, c.user, c.minor_edit, c.text,
 c.timestamp, c.is_new, c.is_redirect FROM cur c  ORDER BY c.title
 ASC</p>
  <r>CpPreparedStatement:PRST0000</r>
</call>
<call conn="CONN0000" src="PRST0000" m="PRST_executeQuery"
 start="20100507.115324.578" d="0">
  <r>CpResultSet:RSST0000</r>
</call>
                  (remaining lines ...)..............
```

El log tiene un formato XML (excluyendo la cabecera y el elemento raíz, que están ausentes). Los elementos principales son el `<call>` (para representar llamadas a las interfaces de JDBC) y `<custom>`(elementos adicionales que no corresponden a un método JDBC). Estos internamente pueden contener los elementos `<f>` (*stack frame* de la invocación), `<p>` (parámetro de la invocación) y `<r>` (valor de retorno de la invocación).

En las líneas mostradas se puede observar cómo se crea una conexión (con el identificador *CONN0000*), se prepara una sentencia (creando una *CpPreparedStatement* con identificador *PRST0000*), y sobre esta sentencia se ejecuta una consulta.

Un aspecto interesante es la representación de algunos valores como un *hash*/*digest* del mismo. Por ejemplo:

```
<call conn="CONN0000" src="RSST0000" m="RSST_getBytesS" start="20100507.115324.593"
 d="0">
  <p>text</p>
  <r>96edae55495cc8e82d9b1f090360fcb9de01649eb113d8214780b0a6b002bdcd:    14:Primer
 planeta</r>
</call>
```

Observar cómo un dato binario lo hemos convertido en una cadena con su *hash* criptográfico (*SHA1*), junto con su longitud y una representación de sus primeros bytes. Esta representación es útil para los valores demasiado grandes como para escribirlos en el log(del orden de KB's o más allá). También se puede utilizar para enmascarar datos confidenciales (aunque en primer lugar estos datos no deberían estar presentes en un entorno de pruebas).

4.4 Limitaciones y trabajo futuro

Aunque el wrapper y sus utilidades adicionales permiten una adaptación sumamente flexible de los programas ante una evolución del esquema, lo que se ha hecho no es si no una base sobre la que desarrollar funcionalidad adicional que haga que la evolución del programa sea bastante más automática. Asimismo hay algunas aristas y detalles que

sería interesante pulir. Al explicarlos, evitamos al lector figurarse que el desarrollo es más de lo que en realidad es, y proponemos a qué aspectos, más interesantes, se deberían dirigir esfuerzos futuros.

- **Mejorar** el **soporte** para sentencias donde las **variables** aparezcan como **literales** en lugar de como parámetros JDBC. Sería posible que dos sentencias estructuralmente iguales se identificaran como distintas por llevar distintos literales. Asimismo dichos valores no se estarían teniendo en cuenta para la identificación de dependencia de datos. Hay que modificar el proceso de elaboración del informe de sentencias para solventar estos problemas.
- Almacenar y reproducir **metadatos** del **esquema anterior**. En ocasiones el comportamiento del acceso a datos está influido por los metadatos (definición de las tablas, columnas, etc) del esquema. Al cambiar el esquema los datos cambiarían y esto interferiría el funcionamiento del programa. Esto puede ser causa de problemas, de modo que habría que "traducir" los accesos a los metadatos para que devuelva aquellos que el programa "necesita".
- Implementar un lenguaje de especificación del cambio de esquema. Puede estar basado en los SMO de Curino et al [1]. Asimismo puede ser interesante el mapeo DED.
    - Una vez especificado el cambio en la BD, es posible definir cambios automáticos en las sentencias existentes (reduciendo así el trabajo de los técnicos). Este área puede ser la que suponga un trabajo más extenso, ya que la casuística de las operaciones de modificación, el esquema subyacente y las sentencias de acceso es muy amplia.
    - Asimismo permitiría indicar qué ubicaciones del programa (donde se accede a BD) se ven afectadas, y permitir hacer una estimación objetiva del precio del cambio. Esto sería útil tanto para programadores como para gestores.
- Buscar o desarrollar un repositorio de casos de estudio para la evolución de programas de BD.
    - Casos previsibles del mundo real/tutoriales (casos sencillos).
    - Casos sintéticos. Con una complejidad/tamaño mínimo para representar cada requisito individual de la herramienta.
- Enriquecer el repertorio de *matchers* y de *handlers*. Que adaptar sea lo más sencillo posible. Evitar en la medida de lo posible la necesidad de implementar *handlers* personalizados.
- Desarrollar una funcionalidad que permita (basándose en logs de actividad recuperados), hacer pruebas unitarias para verificar que ante cualquier modificación en el sistema las peticiones y las respuestas sigan los patrones esperados.
- Probar y poner a punto para funcionar a gran escala. Encadenamiento de wrappers (evoluciones acumuladas). Realizar un filtrado de reglas de traducción para evitar tener que ejecutar un número excesivo de *matchers* a cada petición.

Bibliografía

[1] C. Curino, H. Moon, C. Zaniolo, *Graceful Database Schema Evolution: the PRISM Workbench*, 2008,PVLDB 1,1 p761-772
[2] J.M. Hick, J.M. Hainaut, *Database Application Evolution. A transformational Approach*, 2006,DKE 59,3 p534-558
[3] Amila Karahasanovic, *Identifying impacts of database schema changes on applications*, 2001,in: Proc of the 8th Doctoral Consortium at theCAiSE*01. Freie Universität Berlin, Interlaken, Switzerland, 2001, pp. 93–104.
[4] A.Maule, W. Emmerich, D.S.Rosenblum, *Impact Analysis of Database Schema Changes*, 2008,ISCE 2008
[5] B.Schneiderman, G.Thomas, *An Architecture for Automatic Relational Database System Conversion*, 1982,ACM Transactions on Database Systems, Vol. 7, No. 2, June 1982, Pages 235-257
[6] Ph. Thiran, J.L. Hainaut, *Wrapper Development for Legacy Data Reuse*, 2001,WCRE 2001, pages 198-207
[7] A. Torrentí-Román, L. Pascual-Miret, L. Irún-Briz, S. Beyer, F. D. Muñoz-Escoí, *Aspy An Access-Logging Tool for JDBC Applications*, 2007,Univ.Politécnica Valencia TR-ITI-ITE-07/24
[8] (various project commiters), *Elvyx*, http://elvyx.sourceforge.net/
[9] Martin Fowler, Pramod Saladage, *Evolutionary Database Design*, http://www.martinfowler.com/articles/evodb.html
[10] Andy Martin, Jeff Goke, Alan Arvesen, Frank Quatro, *P6Spy*, http://www.p6spy.com/
[11] C.Curino, H.J.Moon, C.Zaniolo, *Prism*, http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Prism