# Anexos

A Guía básica de usuario de DDBC

B Ejemplo de código difícilmente analizable

# Guía básica de usuario de DDBC

## Resumen

El framework *DDBC* (*Decorator DataBase Connection*) permite la adaptación de un programa ante un cambio en la base de datos subyacente. Dicha adaptación en ocasiones será preferible frente a una modificación "*manual*" del programa afectado. Pero para realizar dicha adaptación son precisas algunas explicaciones previas.

Este trabajo busca ser una guía básica para mostrar los conceptos básicos relacionados, e indicar paso a paso el proceso a seguir para realizar la adaptación

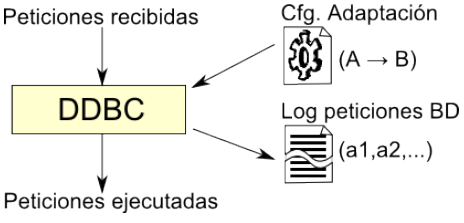
## 1 Conceptos básicos

El framework *DDBC* es una implementación del API *JDBC*(*Java DataBase Connection*) con el propósito de añadir funcionalidades de adaptación y de log sobre otra implementación subyacente.

JDBC es el API estándar de Java para acceder a BD relacionales. A nivel de las librerías del *Java Runtime* consiste esencialmente en una colección de interfaces que definen las propiedades y funcionalidades asociadas a los objetos *Connection*, *Statement*, etc.

Para acceder a una BD concreta, en un programa debe tener acceso a una implementación de JDBC que se encargue de traducir las peticiones del programa sobre el API, en peticiones de más bajo nivel (normalmente específicas de la BD). Ello implica que cada productor de cada BD distribuya su propia implementación. Estrictamente se debería llamar **driver** únicamente a estas implementaciones que abstraen de la BD a bajo nivel, pero por extensión llamamos driver a cualquier implementación de JDBC.

Antes del proceso de adaptación, el programa accedía a la BD modificada utilizando un driver específico. Nos referimos a este driver como ***driver subyacente***, o driver heredado. Lo llamamos así porque después de la adaptación, nuestro driver (***driver wrapper***) se configurará sobre el subyacente, recibiendo las peticiones en su lugar, y pasándoselas a éste una vez transformadas.




Nuestro *driver wrapper* puede asumir dos roles: La adaptación de sentencias, y la escritura de logs. Ambos roles son opcionales.

La **funcionalidad de adaptación** de sentencias consiste en convertir las peticiones de JDBC recibidas (que se refieren al esquema de BD en el estado inicial), a peticiones JDBC a ejecutar por el driver subyacente (ya traducidas para corresponder con el esquema de BD posterior).

La funcionalidad de adaptación es el propósito principal del driver, pero puede deshabilitarse (por ejemplo si en un momento dado solo queremos recuperar los logs), de manera que las peticiones se delegarán sin alteraciones sobre el driver subyacente, como si las hubiera solicitado directamente el programa. Esta deshabilitación la

utilizamos por ejemplo en las primeras fases de nuestro procedimiento de adaptación, cuando el programa todavía se encuentra trabajando sobre el esquema inicial.

Para realizar la adaptación es preciso una **especificación de adaptación**, que indicará qué sentencias SQL se esperan, y el tratamiento que se les debe dar. Las instancias de sentencias esperadas se agrupan en clases de sentencia. Una **clase de sentencia** es una abstracción de las sentencias SQL que son estructuralmente iguales, de forma que dos sentencias pertenecen a la misma clase si son idénticas o si lo único en que se diferencian es en los valores de sus variables. De esta manera las sentencias `select * from cur where title='Mercurio'` y `select * from cur where title='Venus'` pertenecerían a la misma clase, pero `select cid,title,text from cur where title='Mercurio'` o `select * from old where title='Mercurio'` pertenecerían a clases distintas.

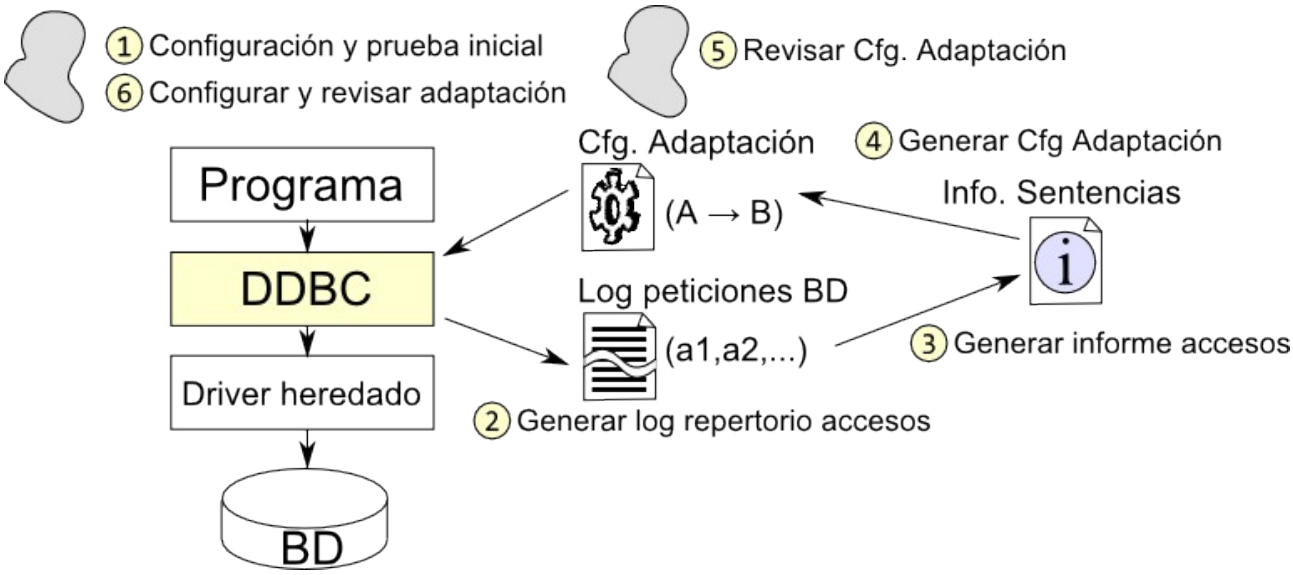
La **funcionalidad de obtención de logs** se encarga de registrar las peticiones (con sus parámetros y respuestas) realizadas sobre el API JDBC. Para nuestro proceso de adaptación necesitamos conocer no solo qué sentencias SQL se ejecutan, si no que necesitamos también información sobre la gestión de transacciones ACID (qué sentencias se ejecutan juntas), sobre la ubicación de las llamadas (a partir de los *stack frames*), y los valores de los parámetros JDBC y de los retornos de las llamadas. Posteriormente, una vez que la aplicación adaptada ya se encuentra funcionando y esa información es excesiva, se puede rebajar la verbosidad para registrar únicamente los fallos encontrados.

Para realizar la adaptación es necesario un fichero de especificación de adaptación, que se podría describir como un mapeo o asociación entre las sentencias SQL previstas (referidas al esquema inicial) y las sentencias SQL a ejecutar (referidas al esquema nuevo). Así por ejemplo una asociación dentro de nuestro caso de estudio sería:

```
  <mapping id="select cur where title">
    <matcher type="equals">
      SELECT c.cid, c.title, c.user, c.minor_edit, c.text,
c.timestamp, c.is_new, c.is_redirect FROM cur c  WHERE c.title=?
    </matcher>
    <handler type="literal">
      select r.rid as cid, p.title, r.user, r.minor_edit, t.text,
r.timestamp, p.is_new, p.is_redirect from REVISION r, PAGE p, TEXT t
WHERE r.PAGE=p.PID and r.RID=t.TID and p.LATEST=r.RID and p.title=?
    </handler>
  </mapping>
```

Gran parte de los esfuerzos del proceso de adaptación están enfocados a conseguir la especificación de adaptación.

El proceso de adaptación se podría describir según el siguiente gráfico:




Se pueden observar un par de siluetas junto a las tareas que pueden requerir del talento de un técnico experimentado. Existe un producto intermedio (el informe de sentencias) que se obtiene como resumen y análisis de la información contenida en el log.

En primer lugar se adapta el programa (sección 2) para que utilice DDBC sobre el driver subyacente, y se prepara el entorno de pruebas. A continuación se ejecuta la aplicación (sección 3) para obtener un log con un repertorio exhaustivo de accesos. Después de esto se genera el informe de sentencias (sección 4), que resume qué sentencias se ejecutan, y analiza qué dependencias se encuentran entre ellas. Con esta información se genera una propuesta de especificación de adaptación (sección 5). Esta propuesta debe ser revisada (sección 6) por un técnico para traducir las sentencias SQL que fuera necesario. Finalmente se habilita la adaptación del programa (sección 7), y se verifica el correcto funcionamiento del mismo. La especificación de adaptación puede precisar de algunos mecanismos adicionales que describimos en la sección 8.

# 2 Configuración y prueba inicial

En esta sección se trata sobre cómo hay que modificar el programa para que utilice el *driver wrapper* en lugar del *driver subyacente* (sección 2.1), y sobre cómo prepararíamos un repertorio de pruebas que fuera adecuado para obtener los logs de acceso a BD (sección 2.2). En estas tareas nos podemos encontrar con una variedad de escenarios posibles. Por ello, a parte de explicar los cambios correspondientes al caso de estudio resuelto, daremos alguna explicación sobre otros casos alternativos y cómo abordarlos.

## *2.1 Interponer el driver DDBC*

Para utilizar el driver DDBC en lugar del driver subyacente debemos hacer dos cosas: 1) Hacer que el programa utilice el driver DDBC, y 2) configurar DDBC para que conozca cómo se conecta con el driver subyacente.

En nuestro caso el programa a modificar estaba preparado para poder cambiarle el driver utilizado, ya que los parámetros necesarios (el *className* del driver, la URL de conexión, el usuario y el driver) venían indicados en el fichero de configuración del programa. Siendo la configuración inicial:

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/uwiki
jdbc.user=uwiki
jdbc.pwd=uwiki
```

La configuración pasaría a ser la siguientes:

```
jdbc.driver=ddbc.DDriver
jdbc.url=ddbc:jdbc:mysql://localhost:3306/uwiki
jdbc.user=uwiki
jdbc.pwd=uwiki
```

Con estos cambios el programa registrará el *driver wrapper*, y reconocerá la URL con el prefijo especial (*ddbc:*) como suya propia.

Los cambios pueden ser algo diferentes en función de cómo se obtienen en el programa las conexiones a BD. Los parámetros pueden estar *hardcodeados* dentro del programa, o puede que las conexiones se obtengan a través de componentes intermedios como los *DataSource*. Unos casos pueden necesitar modificar el código fuente para establecer los parámetros, o podría ser incluso necesario modificar las instrucciones de obtención de la conexión.

Es posible intercalar el wrapper sin necesidad de modificar el código de obtención de la conexión, gracias a una técnica copiada de P6Spy, al registrar el driver DDBC, éste hace un truco para anteponerse a los otros drivers en la lista del DriverManager, y entonces poder hacerse cargo, en lugar de los *drivers* subyacentes, de los esquemas que se desee adaptar. Sin embargo, dado que otros *drivers* pueden hacer la misma maniobra, es preferible utilizar el prefijo "*ddbc:*" para asegurarnos de que es nuestro driver el que construye la conexión.

El *classpath* del programa en ejecución deberá modificarse para incluir el fichero *ddbc.jar* con nuestro driver. Para configurar sus detalles dependemos de un fichero de configuración propio. La ubicación del fichero se puede indicar a través de la variable de entorno *DDBC_CONFIG*. La ubicación por defecto es *ddbc.xml* en el directorio de trabajo del proceso. Éste sería el fichero de configuración preparado para obtener los logs de acceso a BD:

```
<?xml version="1.0" encoding="UTF-8"?>
<config override-drivers="0">
  <urlMapping>
    <wrappedUrl url="jdbc:mysql://localhost:3306/uwiki_test"
      delegateDriver="com.mysql.jdbc.Driver" />
    <handler id="uwiki_test" type="cpN2">
      <logging profile="reportGen" >
        <logListener type="writeXML" file="test-log.xml" />
      </logging>
    </handler>
  </urlMapping>
</config>
```

Observar cómo aparece la URL y el *className* del driver subyacente. Con esto puede distinguir las URLs de conexión que debe gestionar, y con qué implementación creará las conexiones subyacentes.

En el elemento *handler* se le indica la gestión a realizar con las peticiones. Le estamos indicando la escritura de logs (con perfil "*report Generation*", para la elaboración del informe de sentencias). Le indicamos el nombre de fichero en el que debe escribir.

Este fichero de configuración, inicialmente configurado para escribir logs, pero no adaptar, lo modificaremos más adelante (en la sección 7) para habilitar la adaptación del esquema. Hasta ese momento, el programa y los módulos de pruebas ejecutados con

DDBC deberán acceder sobre una instancia de la versión inicial del esquema.

## *2.2 Preparación del repertorio de pruebas*

Para elaborar una buena adaptación, en la que no surjan sorpresas *a posteriori* (sentencias SQL no previstas que fallen), necesitamos un repertorio de pruebas suficientemente exhaustivo. Lo denomino repertorio de pruebas por semejanza con las pruebas del software, donde tratamos de verificar distintas variantes (caminos de ejecución, combinaciones de valores, etc). En aquel caso se utilizan las pruebas para ver que el comportamiento es el esperado en cualquier situación, y en nuestro caso se utiliza para ejecutar un caso de uso exhaustivo que deje claro lo que se espera de la adaptación de las sentencias.

Obviamente debe forzarse la ejecución de todas las clases de sentencias. Además, debemos identificar los valores fijos y los valores variables de cada clase de sentencia, y las dependencias que existen entre ellas cuando se ejecutan dentro de la misma transacción de BD. Así por ejemplo cada clase de sentencia debería ejecutarse al menos un par de veces, modificando cada valor que pueda ser modificado para permitir distinguir cuáles son variables.Así por ejemplo en la sentencia *insert* del ejemplo, para la sentencia

```
insert into CUR (CID, TITLE, USER, TEXT, IS_NEW, IS_REDIRECT) values
(?,?,?,?,?,?)
```

deberíamos invocarla cambiando todos los parámetros (insertando en al menos dos páginas distintas, con usuarios distintos, *etc*).

Igualmente, para ilustrar correctamente cada operación compleja (una transacción de BD que involucra varias sentencias), necesitaremos varias ejecuciones para dar evidencia de que los datos que coinciden entre sentencias son dependencias de datos, y no meras coincidencias.

Se suele recomendar la utilización de pruebas automatizadas (si bien esto no siempre es esto fácil, y no siempre se consiguen cubrir todos los casos). En su defecto (o además de las automatizadas) un operador puede "jugar" con el programa, recorriendo todas las operaciones disponibles y ejecutándolas con diversos parámetros.

En nuestro caso hemos utilizado una prueba basada en *DbUnit*. Este entorno de prueba nos permite establecer la base de datos en un estado inicial antes de ejecutar nuestra prueba, y verificar una serie de asertos a lo largo de la ejecución, para verificar que todo está en orden.

Para realizar las pruebas con *DbUnit* hemos utilizado la clase *SampleTest* (en *uWiki_Evol/src/uwiki/evol*), y para obtener los datos iniciales de la BD la clase *GenDBUnitData*, en el mismo directorio. Los ficheros con los datos iniciales de la BD de pruebas se encuentran en los ficheros *uwiki_initial_410_flat.xml* y *uwiki_initial_42_flat.xml*.

# 3 Generar log del repertorio de accesos

Una vez configurado el programa, y preparado el repertorio de pruebas, simplemente ejecutamos el repertorio de pruebas, verificamos que el fichero de log generado es adecuado, y guardamos una copia. Esta es una muestra del contenido del fichero de log:

```
  <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>`. 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 adicional 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 grandes (del orden de KB's o más allá). Se pretende que este mecanismo se pueda utilizar también para enmascarar datos confidenciales (aunque en primer lugar estos datos no deberían estar presentes en un entorno de pruebas). Hacen faltas algunas mejoras para configurar qué datos se convierten a *hash* (y de qué manera).

# 4 Generar el informe de sentencias

En el log de acceso a datos obtenido hay una gran cantidad de información para poder analizar lo que sucede en la aplicación, pero no resulta totalmente adecuada para lo que pretendemos hacer (la especificación de adaptación). El principal problema es que para cada clase de sentencia existirán múltiples instancias, y lo que necesitamos es mostrar

cada tipo de sentencia SQL una sola vez (en lugar de varias), con el resumen de la información computada a partir de sus N apariciones. Así por ejemplo la *select* mostrada del fichero de log (`SELECT ... FROM cur c ORDER BY c.title ASC`), aparecerá varias veces en el fichero de log, mientras que solo queremos que aparezca una vez en el informe de sentencias.

Por otro lado también se quiere analizar qué sentencias se ejecutan dentro de la misma transacción, y qué dependencias de datos existen entre estas sentencias. Esta información no se utiliza por el momento. El objetivo era tenerla en consideración para una traducción automática de las sentencias.

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.

# 5 Generar propuesta de especificación de adaptación

Para generar la especificación de adaptación que utiliza el driver necesitamos un fichero que asocie las sentencias SQL esperadas con el tratamiento que se les debe dar. Por el momento utilizamos una plantilla XSL (sr2spec.xsl) para obtener una especificación inicial. En esta especificación inicial aparecen reflejadas todas las clases de sentencia detectadas, y como tratamiento (pendiente de modificación por un técnico) establecemos que se ejecute la misma sentencia literalmente. También se indica que para las sentencias inesperadas (elemento *otherwise*) éstas sean propagadas sin modificaciones (handler tipo *passThrough*) al driver subyacente.

Este sería un extracto del fichero xml generado:

```
  <adapt-spec>
    <mapping id="1">
      <matcher type="equals">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</matcher>
      <handler type="literal">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</handler>
    </mapping>
  ..........................(more mappings).......................
    <otherwise>
      <handler type="passThrough" unexpected="true" />
    </otherwise>
  </adapt-spec>
```

Se puede pensar que la sugerencia no es nada sofisticada (y es cierto), pero ofrece al técnico una estructura en la que simplemente tiene que traducir las sentencias que deban ser adaptadas. Su preocupación no es "qué estructura tiene el fichero" o "qué hay que adaptar", si no simplemente realizar la adaptación, como veremos en la siguiente sección.

# 6 Revisar especificación de adaptación

La especificación de adaptación generada inicialmente debe ser revisada. Un técnico examinará cada sentencia mapeada, modificando el tratamiento propuesto en el *handler* para que sea lo más adecuado para el caso concreto. Así para el ejemplo mostrado (la consulta de las páginas actuales en la tabla CUR), el *handler* definido debería quedar del siguiente modo:

```
  <handler type="literal">
    select r.rid as cid, p.title, r.user, r.minor_edit, t.text,
r.timestamp, p.is_new, p.is_redirect from REVISION r, PAGE p, TEXT t
WHERE r.PAGE=p.PID and r.RID=t.TID and p.LATEST=r.RID order by p.TITLE
asc
  </handler>
```

Observar que las sentencias traducidas deben conservar la "interfaz" de la original: deben retornar las mismas columnas y tener los mismos parámetros. Así por ejemplo en el listado del *resultset* hemos utilizado `r.rid as cid` para que para el programa la primera columna siga siendo *cid*.

En aquellos casos de sentencias que no necesitan ningún cambio, es razonable modificar el *handler* a la definición `<handler type="passThrough" />` (que es más económico, y queda más claro que no hay cambios).

Respecto al *otherwise* para las sentencias no previstas, por defecto se sigue una política permisiva: Se intenta ejecutar la sentencia inesperada en el esquema nuevo y se escribe una advertencia en el log (para que dicha sentencia pueda ser incluida en la especificación). Si la sentencia sigue siendo correcta en el nuevo esquema, lo deseable es que siga funcionando. Si es incorrecta fallará (pero normalmente no perdemos nada por intentarlo).

Sin embargo también sería razonable hacer que la política del *otherwise* fuera "*draconiana*": ¿Quién nos garantiza que una sentencia inesperada no corrompe la BD o produce resultados erróneos que producen un perjuicio real? En tal caso el tratamiento sería fingir un fallo en la ejecución de la sentencia y escribir en el log. Esto lo conseguimos definiendo el handler como `<handler type="fail">Texto del mensaje de fallo</handler>`.

Con esto cubrimos las necesidades más sencillas. Pueden suceder casos en los que necesitemos utilizar expresiones regulares, o puede que necesitemos utilizar un "estado de transacción" para hacer el tratamiento correcto en función de las sentencias previas, o puede incluso que necesitemos definir nuestro propio handler para hacer tratamientos a medida (como sucede en nuestro ejemplo en la operación de modificación). De todo ello tratamos en la sección 8.

# 7 Configurar y revisar la adaptación

Hasta este punto hemos estado utilizando el driver adaptador con una configuración de logs verbosos, pero sin alterar los accesos a BD realizados. Si queremos que el programa funcione sobre la nueva versión del esquema deberemos activar el comportamiento de adaptación con la especificación que hemos elaborado.

Cabe observar la posibilidad de que existan distintos esquemas adaptados, de modo que el nombre de la especificación para un esquema dado debe ser suficientemente identificativo. Si por ejemplo queremos especificar que el fichero contiene la traducción de sentencias para el esquema *uwiki*, para simular la versión *41* sobre la versión *42*, un nombre adecuado podría ser *st_uwiki41_42.xml*.

Dentro del fichero de configuración del driver (*ddbc.xml*), habría que modificar la definición del *handler del esquema*, que podría quedar del siguiente modo:

```
<handler id="uwiki_test" type="cpN2">
  <logging profile="none" >
    <logListener type="writeXML" file="ddbc-log.xml" />
  </logging>
  <dai type="StN1" src="file:st_uwiki41_42.xml"/>
</handler>
```

Observar que establecemos el perfil de trazas a "*none*", de modo que solo se escribirán los avisos y fallos en el log, y que especificamos el *dai* (*Data Adaption Interface*) a nuestra implementación (*Statement Translator N1*).

Una vez configurado el programa (y/o módulo de pruebas) podemos ejecutar el mismo repertorio de pruebas que habíamos utilizado inicialmente para obtener el log (secciones 2.2 y 3) para verificar que el comportamiento del programa es el esperado. Si existe alguna anomalía esta debería ser detectada, y modificada revisados los logs y la

configuración asociada para tratar de solucionarlo.

# 8 Configuración para casos especiales

## *8.1 Utilización del handler fail*

El *handler fail* hace que la sentencia solicitada falle. Esto puede ser útil cuando las peticiones inesperadas en el *otherwise* no son bienvenidas, o cuando necesitamos indicar que una petición concreta no se encuentra disponible.

Podría suceder que por la premura, dejemos una adaptación parcialmente operativa en la que algunas peticiones (las más sencillas y prioritarias) se encuentren adaptadas, mientras que otras se dejan deshabilitadas. Así por ejemplo en la aplicación wiki de ejemplo podríamos tener adaptadas las operaciones de consulta, permitiendo visualizar los contenidos existentes, pero no permitir la modificación (de modo que cuando alguien intentase modificar una página, no se permitiera). Puede suceder que las peticiones más complejas de un programa sean también las más prescindibles, de modo que una adaptación parcial permitiría demorar *sine die* las partes más complejas de una adaptación (lo cual es una oportunidad y un riesgo).

## *8.2 Utilización de matchers y handlers con expresiones regulares*

La identificación y traducción literal utilizados en el ejemplo son suficientemente buenos si todos los elementos variables de una sentencia están representados como parámetros JDBC (símbolos “?”). En caso contrario, si los valores de entrada de una sentencia aparecen como literales (NULL, 9, 'Mercurio', etc), será necesario un mecanismo capaz de tratar correctamente con las partes fijas y las variables de las sentencias. Las expresiones regulares son un concepto útil para identificar y modificar estas sentencias variables.

En la siguiente definición encontramos un ejemplo de uso del matcher y del handler basados en expresiones regulares:

```
  <mapping id="stmtSelect Cur">
    <matcher type="regexMatcher">select .* from cur c .*</matcher>
    <handler type="regexHandler" >
      <rx-rule exclusion-group="where"><!-- if a WHERE was present -->
        <rx-match>select (.*) from cur c where (.*)</rx-match>
        <rx-subst>select $1 from REVISION r, PAGE p, TEXT t where
r.page=p.pid and p.latest=r.rid and r.rid=t.tid AND $2</rx-subst>
      </rx-rule>
      <rx-rule exclusion-group="where"><!-- ELSE if a WHERE was NOT
present -->
        <rx-match>select (.*) from cur c (.*)</rx-match>
        <rx-subst>select $1 from REVISION r, PAGE p, TEXT t where
r.page=p.pid and p.latest=r.rid and r.rid=t.tid $2</rx-subst>
      </rx-rule>
      <!-- Translate columns -->
      <rx-rule >
        <rx-match>c\.title</rx-match>
        <rx-subst>p\.title</rx-subst>
      </rx-rule>
  ..............(resto de reglas de traducción de columnas)...........
    </handler>
  </mapping>
```

Con este mapeo se trataría de traducir cualquier consulta sobre la tabla cur. El *matcher* ***regexMatcher*** verifica si la sentencia cumple la sentencia regular (consulta de la tabla *cur*, con cualquier lista de columnas y cualquier clausula adicional).El *handler* ***regexHandler*** permite alterar las sentencias SQL de un modo flexible. Permite definir una lista de reglas (***rx-rule***) que se ejecutarán en secuencia. Para cada regla, hay un patrón de búsqueda (***rx-match***) y un patrón de sustitución (***rx-subst***). Para cada ocurrencia de la expresión buscada será sustituida. La expresión puede referirse a la sentencia SQL en su conjunto, o a parte de ella (como la referencia a la columna *title*). Es posible establecer cierta ejecución condicional a través de grupos de exclusión (atributo ***exclusion-group***). Si en una sentencia del grupo se encuentra el patrón indicado, se excluye al resto de reglas del grupo de la ejecución.

Con los grupos de exclusión, no solo expresamos que solo se espera que se ejecute una regla del grupo. Puede suceder en el caso de tratamientos alternativos que el *rx-match* se pudiera verificar para varios de ellos, y que la suma de las modificaciones propuestas corrompieran la sentencia SQL, por ejemplo introduciendo un segundo *WHERE* (esto podría suceder en un caso parecido al del ejemplo, si no hubiera cambios en las tablas consultadas). Si se pueden verificar varias reglas, hay que tener cuidado con el orden, no vaya a ejecutarse primero la que no debería. Así si en el ejemplo se antepusiera la regla para sentencias sin WHERE, sucedería que las sentencias con WHERE acabarían corrompiéndose.

Respecto a la sintaxis de las expresiones regulares, se puede observar cómo el patrón "`.*`" designa un texto variable (cualquier carácter apareciendo de 0 a N veces). Los paréntesis alrededor "`(.*)`" se utilizan aquí para definir un grupo de captura, que son útiles para indicar la posición de los elementos variables ($1, $2) dentro de la nueva sentencia. Se puede encontrar más información sobre la sintaxis de las expresiones regulares en la documentación de la clase *Pattern* [w3JavaPattern].

## 8.3 Utilización de handlers a medida

Es posible que los tratamientos básicos no sean adecuados para lo que se necesita en un caso concreto. De hecho así sucede en nuestro caso de estudio de la aplicación *µWiki*.

Por ello se da libertad a los técnicos para que programen e integren en el framework un handler a medida.

Para quien quiera implementar su propio handler, vale cualquier clase que implemente la interfaz *StatementHandler*, pero conviene observar y utilizar como modelo la clase *HandlerCmdEdit41in42* utilizada en el ejemplo. Los *handlers* personalizados deben de ser declarados en la especificación de adaptación para poder ser utilizados.

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

En este caso asociamos el tipo *cmdEdit* a nuestro handler, e indicamos el nombre de la clase de la implementación.

El nuevo tipo de handler puede ser utilizado ahí donde se necesite.

```
<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>
```

## 8.4 Utilización de estado de transacción

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.

Esta situación se podría solucionar identificando el estado que debe darse antes de ejecutarse una sentencia, y el estado posterior al ejecutar una sentencia. El estado por defecto es "*initial*". Las operaciones no transaccionales (como las consultas del ejemplo) comienzan en *initial* y terminan en *initial*. En nuestra secuencia transaccional (`insert into OLD`, `delete from CUR`, `insert into CUR`), definimos dos estados intermedios (*cmdEdit-insertOld*, y *cmdEdit-deleteCur*). Observar cómo identificamos el estado utilizando el nombre de la operación/grupo de sentencias, y el nombre de la sentencia inmediatamente anterior que lleva al estado indicado.

Utilizo el atributo *fromState* en un *matcher* para indicar el estado que requiere para su ejecución. Por defecto es "*initial*". Si el estado no es igual, el *matcher* considerará que la condición no se cumple, y se continuará la búsqueda. El atributo *toState* en el *handler* indica el estado en el que se deja la variable tras la ejecución.

# Ejemplo de código difícilmente analizable

En análisis estático del código fuente busca determinar qué accesos a BD (qué sentencias SQL se ejecutan) localizando las llamadas al API, y rastreando según las dependencias de datos exactamente qué valor tienen las variables que determinan la operación (qué sentencia SQL contiene la cadena que se le pasa al executeQuery/executeUpdate).

Implementar este proceso no es trivial. Para colmo existen casos en los que resulta inasumible implementar un analizador adecuado.

- Caso de uso de estructuras de datos que dificultan conocer el valor de un dato concreto
- Invocación de API's cuyo retorno es difícil de determinar.

Un caso que no es en absoluto infrecuente es el uso de mapeos objeto-relacionales en el acceso a BD. Existen librerías tales como Hibernate que de hecho utilizan estos mecanismos, también suele suceder que en distintos desarrollos se utilice su propia implementación a medida.

Nos encontramos dentro de los método s de acceso a datos que se consulta el nombre de las clases y el nombre de los campos, para asociarlos con la tabla y las columnas. Sin embargo para conocer estos valores concretos hay que conocer el tipo de datos que se le especifica en concreto al método de acceso. Una vez conocidos todos los tipos de datos que se le pueden suministrar al método, además habría que hacer que el analizador supiera deducir los nombres de clase y de campos a través de las llamadas al API de reflexión (getFields(), getName(), etc).

He rechazado utilizar análisis estático, me corresponde entonces aportar al menos un ejemplo de código que considero que sea dificilmente analizable.

A continuación se muestra un esbozo trivial de consulta mediante reflexión y mapeo objeto-relacional. Considerar que conlleva dificultad para su análisis, y todavía se puede hacer más complejo. En la línea 55 se puede observar el método de acceso con mapeo. En la línea 91 se puede observar una versión simplificada (haría lo mismo sin reflexión).

```
1. import java.lang.reflect.Field;
2. import java.sql.*;
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. public class TrivialORM {
7.    public static class SampleData {
8.          public String field1;
9.          public String field2;
10.         };
11.
12.         public static void main(String[] args) {
13.               try {
14.                     Class l_class = SampleData.class;
15.                     Connection l_conn = borrowConnection();
16.                     List<SampleData> l_dataList = new
  ArrayList<SampleData>();
17.                     try {
18.                           l_dataList = selectFromClassTable(l_conn,
  l_class, l_dataList);
```

```
19.                     } finally {
20.                           returnConnection(l_conn);
21.                     }
22.                     for (SampleData l_sample : l_dataList) {
23.                           System.out.println(l_sample);
24.                     }
25.               } catch (Throwable e) {
26.                     System.err.println("Catch in
  TrivialORM.main()");
27.                     e.printStackTrace(System.err);
28.               }
29.         }
30.
31.         private static void returnConnection(Connection conn) {
32.               // TODO Auto-generated method stub
33.
34.         }
35.
36.         private static Connection borrowConnection() {
37.               // TODO Auto-generated method stub
38.               return null;
39.         }
40.
41.         /**
42.          * Retrieves a complete list for the selected
  dataType/table.
43.          * The table and column names must match the class and
  fields name.
44.          * Note a field f is accessed directly, not through accesor
  methods such as setF.
45.          * @param <T> Component type of the target list. Must be
  the dataClass or superclass.
46.          * @param conn JDBC Connection. Transaction and closure (if
  any) must be handled by caller.
47.          * @param dataClass Class of the data stored in the table.
48.          *    The Class name and the instance fields names and
  types must match the DB table.
49.          * @param targetList List container where the retrieved
  data is appended
50.          * @return List of the retrieved data. Same reference as
  targetList parameter.
51.          * @throws SQLException
52.          * @throws InstantiationException The data class must have
  a default constructor.
53.          * @throws IllegalAccessException The data class and
  constructor must be accesible(public).
54.          */
55.         public static <T> List<T> selectFromClassTable(Connection
  conn,
56.                     Class<? extends T> dataClass,
57.                     List<T> targetList)
58.                     throws SQLException, InstantiationException,
  IllegalAccessException {
59.               // Values repeatedly used thereafter
60.               // Table names = class name ; columnNames = Field
  names.
61.               Field[] l_fields = dataClass.getFields();
62.               String l_className = dataClass.getSimpleName();
63.
64.               // Compose query
65.               StringBuilder l_sbQuery = new StringBuilder("SELECT
```

```
  ");
66.               String l_separator = "";
67.               for (Field curF : l_fields) {
68.                     l_sbQuery.append(l_separator);
69.                     l_sbQuery.append(curF.getName());
70.                     l_separator = ", ";
71.               }
72.               l_sbQuery.append(" FROM " + l_className);
73.
74.               // Execute query and retrieve results
75.               Statement l_stmt = conn.createStatement();
76.               ResultSet l_rs =
  l_stmt.executeQuery(l_sbQuery.toString());
77.               while(l_rs.next()) {
78.                     T l_instance = dataClass.newInstance();
79.                     for (Field curF : l_fields) {
80.                           Object l_value =
  l_rs.getObject(curF.getName());
81.                           curF.set(l_instance, l_value);
82.                     }
83.                     targetList.add(l_instance);
84.               }
85.               l_rs.close();
86.               l_stmt.close();
87.
88.               return targetList;
89.         }
90.
91.         public static List<? super SampleData>
  selectFromSampleData(Connection conn,
92.                     List<? super SampleData> targetList)
93.                     throws SQLException {
94.               // Compose query
95.               String l_query = "select FIELD1, FIELD2 from
  SAMPLEDATA";
96.
97.               // Execute query and retrieve results
98.               Statement l_stmt = conn.createStatement();
99.               ResultSet l_rs = l_stmt.executeQuery(l_query);
100.              while(l_rs.next()) {
101.                    SampleData l_instance = new SampleData();
102.                    l_instance.field1 = l_rs.getString("FIELD1");
103.                    l_instance.field2 = l_rs.getString("FIELD2");
104.
105.                    targetList.add(l_instance);
106.              }
107.              l_rs.close();
108.              l_stmt.close();
109.
110.              return targetList;
111.        }
112.
113.  }
```