Refactoring de Bases de Datos

Desarrollo Evolutivo de Bases de Datos integrado con MDD

Lic. Gustavo Bartolomeo
Facultad de Informática (UNLP)
gbartolomeo@lifia.info.unlp.edu.ar

Lic. Damián Goti
Facultad de Informática (UNLP)
damian.goti@gmail.com

Orientadora: Dra. Claudia Pons (UNLP - UAI) cpons@lifia.info.unlp.edu.ar

Resumen. Cada vez es más frecuente que los requerimientos cambien a medida que un proyecto de software progresa. Al mismo tiempo el cliente demanda resultados rápidos, que puedan implementarse y medirse en períodos cortos de tiempo. Es por ello que en los últimos años han ganando terreno los procesos (o metodologías) evolutivos y ágiles, cuyas principales premisas son el trabajo colaborativo, iterativo e incremental. Pero no ha ocurrido lo mismo en la comunidad de profesionales de bases de datos. Se necesita profundizar las técnicas y herramientas que también soporten el desarrollo evolutivo para las bases de datos. Como respuesta a esto se hace un importante énfasis en el refactoring, ya que permite evolucionar el esquema lentamente con el tiempo, tomando así un enfoque evolutivo. Es por ello que se propone un marco teórico de cómo puede llevarse a cabo un esquema de trabajo evolutivo sobre las bases de datos y una herramienta que automatice las tareas de refactoring. La herramienta se integra totalmente a un paradigma MDD, asumiendo los modelos un rol protagónico en el proceso de desarrollo.

Palabras Claves. Refactoring, Bases de Datos, Procesos Evolutivos, Metodologías Ágiles, Model Driven Development (MDD), QVTo, JET.

1. Introducción

En respuesta a los requerimientos cambiantes y demandas de resultados rápidos, surgen los procesos o técnicas de naturaleza evolutivos y ágiles. Ejemplos de estas técnicas son: TDD, XP, AMDD, SCRUM. Las mismas se caracterizan por ser iterativas e incrementales. Trabajando en iteraciones, se hace una pequeña parte de una actividad como puede ser modelado, testing, codificación, deployment (o despliegue), etc., cada vez, luego en otra iteración otra pequeña parte y así sucesivamente. Este proceso difiere de las metodologías en cascada, en que se identifica el requerimiento que se va a implementar, se crea un diseño detallado, se

implementa, se hace testing y finalmente se hace un deployment de un sistema funcionando. Con un enfoque incremental, se organiza el sistema en una serie de pequeños raleases, más que en un gran ralease.

1.1 Refactoring según Fowler

Un punto clave en las metodologías ágiles es el refactoring. Refactoring [FO 99] es una forma disciplinada de hacer pequeños cambios en el código fuente para mejorar su diseño, haciendo más fácil la forma de trabajar con el mismo. Refactoring permite evolucionar el código lentamente con el tiempo, para tomar un enfoque evolutivo (iterativo e incremental) de programación. Un aspecto crítico sobre refactoring es que mantiene el comportamiento semántico del código. No se debe agregar ni sacar nada en un refactoring, sólo se mejora la calidad.

Más allá de los elementos de cada metodología ágil, en general coinciden que cuando se tiene un nuevo requerimiento que agregar, la primera pregunta que hay que hacer es: *¿Es este el mejor diseño que permite agregar este requerimiento?* Si la respuesta es sí, se agrega el requerimiento. Si la respuesta es no, primero se hace el refactoring necesario para que el código tenga el mejor diseño posible y luego se agrega el requerimiento. En principio esto suena como una carga importante de trabajo, en la práctica, sin embargo, si se comienza con un código de alta calidad, y si se aplica refactoring para mantenerla, este enfoque agiliza el desarrollo porque siempre estamos trabajando con el mejor diseño posible.

1.2 Refactoring de Bases de Datos

Fowler, en [FO 99] afirma que de la misma manera que es posible aplicar un refactoring en el código fuente de la aplicación, es también posible aplicar un refactoring en el esquema de la base de datos. Sin embargo, aplicar un refactoring en la base de datos es algo más complejo por los significativos niveles de acoplamiento asociados a los datos. Cuando se hace un refactoring sobre el esquema de bases de datos, se debe tener en cuenta que no sólo el refactoring aplica al esquema, sino también a sistemas externos que consumen la información. Por este motivo los refactorings de bases de datos son más complejos de implementar que los refatorings de código. La premisa principal consiste en realizar los cambios sobre el modelo garantizando durante un período determinado (lo cual llamamos período de transición) la coexistencia entre ambas versiones de la base de datos.

¿Que necesitamos para automatizar refactoring?

- Poder representar nuestra base de datos en un modelo.
- Poder expresar transformaciones sobre el modelo.
- Poder expresar transformaciones de modelo en código SQL.

Los conceptos de Modelos y Transformaciones que necesitamos para la herramienta son justamente el corazón del paradigma de Desarrollo de Software Dirigido por Modelos (MDD) [KWB 03] [PGP 10] (Ver Apéndice 3).

2. Desarrollo Evolutivo de Bases de Datos

Muchas de las técnicas orientadas a datos son en cascada por naturaleza, requiriendo la creación de modelos bastante detallados antes de que se “permita” comenzar con la implementación. Peor aún, estos modelos son puestos como una línea base y bajo un control de cambios para minimizar los mismos. Considerando los resultados finales, esto debería llamarse proceso de prevención de cambios. En esto radica el problema: las técnicas comunes de desarrollo de bases de datos no reflejan las realidades de los procesos modernos de desarrollo de software.

En cuanto al desarrollo evolutivo de bases de datos, como primer paso, podemos decir que en lugar de intentar diseñar el esquema tempranamente, se va construyendo a través de la vida del proyecto, para reflejar los cambios de requerimientos. Los profesionales que aplican las técnicas de desarrollo modernas, en lugar de gestionar el cambio y seguir técnicas que le permitan evolucionar, trabajan en pequeños pasos con requerimientos que evolucionan. En la medida que se fueron aplicando estas técnicas en el desarrollo de aplicaciones, se ve la necesidad de que las mismas técnicas y herramientas se pueden aplicar al proceso evolutivo de desarrollo de bases de datos.

El segundo paso es adoptar nuevas técnicas que permitan trabajar en una manera evolutiva. Las técnicas evolutivas son el refactoring de bases de datos, modelado de datos evolutivo, tests de regresión, gestión de la configuración de los artefactos de la base de datos y sandboxes para desarrolladores

3.1 Refactoring de bases de datos

Similarmente a un refactoring de código, un refactoring de bases de datos es un simple cambio en el esquema de la base de datos que mejora su diseño mientras mantiene su semántica de comportamiento y de información [AS 06]. Se puede hacer un refactoring sobre aspectos estructurales del esquema de base de datos como definiciones de tablas o vistas o sobre aspectos funcionales como triggers o stored procedures. Cuando se hace un refactoring sobre el esquema de bases de datos, se debe tener en cuenta que no sólo el refactoring aplica a la base de datos, sino también a sistemas externos que consumen la información en la base de datos. Por este motivo los refactorings de bases de datos son más complejos de implementar que los refatorings de código.

Se hace un refactoring a la base de datos para que sea más fácil agregar algo nuevo a la misma. De esta forma, de a pequeños pero continuos pasos se mejora el diseño del esquema, haciéndolo más fácil de entender y evolucionar.

3.2 Modelado de Datos Evolutivo

Más allá de lo que informalmente se diga, las técnicas evolutivas y ágiles no son simplemente codificar y arreglar. Aún se necesita explorar requerimientos y pensar a través de la arquitectura y diseño antes de implementar, es decir se requiere modelar antes de codificar. El ciclo de vida de Agile Model Driven Development (AMDD) [AM 04] [AM 03] se muestra en la Figura 1. Con AMDD se crean modelos iniciales de alto nivel al inicio del proyecto, que muestran el alcance del problema de dominio que se está direccionando así como la potencial arquitectura a construir. Uno de los modelos que típicamente se crean es un modelo conceptual o de dominio que represente las principales entidades del negocio y las relaciones entre ellas.

La cantidad de detalle mostrado en este modelo es todo el necesario para comenzar el proyecto. El objetivo es pensar tempranamente en el proyecto a través de los requerimientos de más alto nivel, sin tener en cuenta los detalles. Con estos detalles se debe trabajar en el momento que se requiera su implementación.

El modelo conceptual naturalmente evolucionará a medida que crezca el conocimiento del dominio, pero el nivel de detalle se mantendrá igual. Los detalles son capturados dentro del modelo de objetos (que podría ser el código fuente) y el modelo físico de datos. Estos modelos son guiados por el modelo conceptual del dominio y desarrollado en paralelo con otros artefactos para asegurar integridad.

El modelo físico de datos (PDM) representa los requerimientos de datos y cualquier restricción del negocio hasta ese momento en el proyecto. Los requerimientos de datos de futuros desarrollos serán modelados durante esos desarrollos.

El modelado evolutivo no es sencillo. Se deben tener en cuenta las restricciones del negocio en su debido momento, y es sabido que muchas veces estas restricciones mutilan los proyectos cuando se detectan tardíamente. Es importante que los profesionales de bases de datos entiendan los matices de las organizaciones para poder aplicarlos en momento necesario.

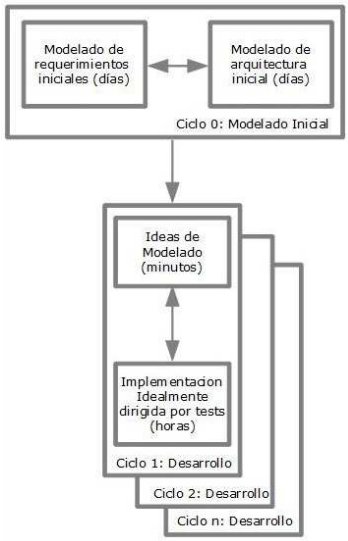

Figura 1: El ciclo de vida de Agile Model Driven Development (AMDD)

3.3 Tests de regresión de bases de datos

Para cambiar en forma segura software existente, se tiene que poder verificar que no se haya modificado un comportamiento. En otras palabras, se necesita poder correr un conjunto completo de tests de regresión en el sistema. Si se descubre que algo no funciona como antes, se deben arreglar o deshacer los cambios introducidos. En la comunidad de desarrolladores, se ha vuelto común para los programadores desarrollar un test de unidad para cada funcionalidad durante el desarrollo. Incluso, técnicas como Test First Development (TFD) [BK 03] fomentan escribir primero los tests que el código. De esto surge la idea de por qué no hacer tests sobre la base de datos. En muchos casos, importante lógica de negocios está implementada en la base de datos en la forma de procedimientos almacenados, reglas de violación de datos, reglas de integridad referencial, lógica de negocio que claramente debe ser testeada.

Test First Development (TFD) es un enfoque evolutivo para el desarrollo. Se debe escribir primero un test que falle antes de escribir el código que implementa la funcionalidad. El siguiente diagrama de actividad UML, describe los pasos de TFD:

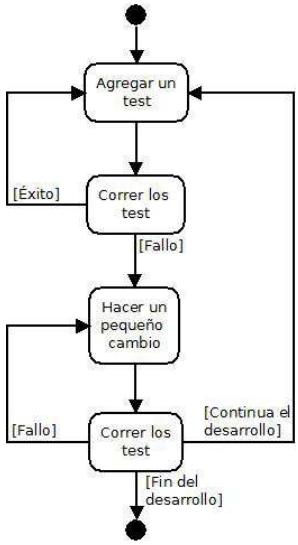

Figura 2: Un enfoque Test First Development

Las ventajas de TFD son que fuerza a pensar a través de la nueva funcionalidad antes de implementarla (efectivamente se está haciendo un diseño detallado), se asegura que hay tests que validan el código y esto da la seguridad que podemos hacer evolucionar el sistema porque sabemos que vamos a poder detectar si algún cambio modifica (o "rompe") algún comportamiento existente. Al igual que para el refactoring de código, tener una regresión completa de test para la base de datos permite el refactoring de base de datos.

Cuando combinamos TFD y refactoring tenemos la metodología Test Driven Development (TDD) [BK 03]. Primero se escribe el código tomando el enfoque TFD, luego que está funcionando, se asegura que el diseño es el mejor posible aplicando los refactorings necesarios.

3.4 Gestión de la configuración de los artefactos de la base de datos

En ocasiones se demuestra que un cambio en el sistema es una mala idea y se necesita volver atrás el mismo. Para esto se necesita poner los artefactos bajo un control de manejo de la configuración, como los scripts DDL, scripts de carga y migración de datos, archivos de modelos de datos, mapeos objeto-relacional, definiciones de vistas, stored prodedures, los scripts para tests y los datos para los mismos.

3.5 Sandboxes para desarrolladores

Un sandbox es un ambiente totalmente funcional en el cual un sistema puede ser ejecutado y testeado. Tener varios sandboxes separados trae varias ventajas. Los desarrolladores pueden trabajar dentro de su sandbox sin preocuparse por dañar o interferir con otros desarrollos, el grupo de testing o calidad puede correr sus pruebas de integración en forma segura y los usuarios finales pueden correr sus sistemas sin preocuparse sobre desarrolladores corrompiendo sus datos o su sistema funcional. La Figura 3 muestra la organización lógica de sandboxes. Decimos que es lógica porque un ambiente grande y complejo puede tener 7, 8 o más sandboxes, mientras pequeños o simples ambientes pueden tener 2 o 3 sandboxes físicos.

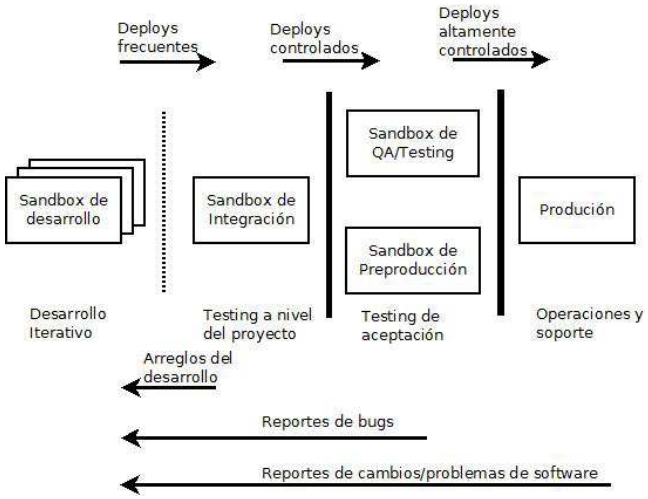

Figura 3: Organización lógica de varios *sandboxes*

3.6 Los dos escenarios de refactoring de bases de datos

Un aspecto crítico en los refactorings de bases de datos es que pueden volverse más complicados por el acoplamiento que puede existir en la arquitectura de la misma, como muestra la Figura 4. El acoplamiento es una media de dependencia entre dos componentes, cuanto más acoplamiento exista entre dos componentes, mayor será la probabilidad que un cambio en uno genere un cambio en el otro.

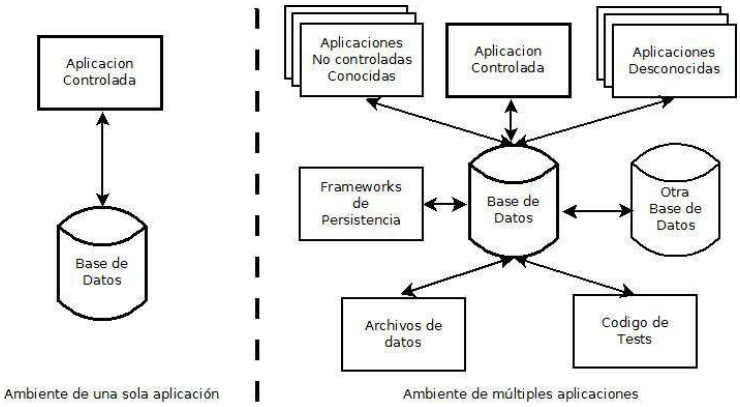

Figura 4: Las dos categorías de arquitectura de bases de datos

La arquitectura de una sola aplicación accediendo a la base de datos es la más sencilla, permitiendo hacer refactoring en paralelo en la aplicación y en la base de datos, y haciendo el deployment los mismos simultáneamente. Se tiene completo control sobre el esquema y sobre el código de la aplicación para acceder al mismo. Por este motivo se puede hacer refactoring en la aplicación y en el esquema al mismo tiempo. No se necesitará brindar soporte al esquema original. Esta situación no es la más común en la realidad.

La arquitectura de múltiples aplicaciones es más complicada porque se tienen varios programas externos interactuando con la base de datos, donde el control de muchos de ellos está fuera de nuestro alcance. En esta situación, no se puede asumir que se hará el deployment de todos los programas externos de una vez, y se debe por lo tanto proveer un periodo de transición (o periodo de deprecación) durante el cual la versión original del esquema y la nueva serán soportadas en paralelo.

Por ejemplo, para implementar el refactoring *Rename Column*, se crea una columna en la tabla con el nuevo nombre, se migran los datos, pero no se borra la columna original de la tabla. Ambas columnas quedan en paralelo durante un periodo de transición para dar tiempo a los equipos de desarrollo a actualizar todas las aplicaciones. También se deben agregar dos triggers, los cuales son corridos en producción durante el periodo de transición para mantener ambas columnas sincronizadas. Luego del periodo de transición, se remueve la columna original y los triggers, resultando en el esquema final.

3.7 El Proceso de Refactoring de BD

La Figura 5 muestra un diagrama de actividad UML con el proceso completo para aplicar un refactoring a la base de datos.

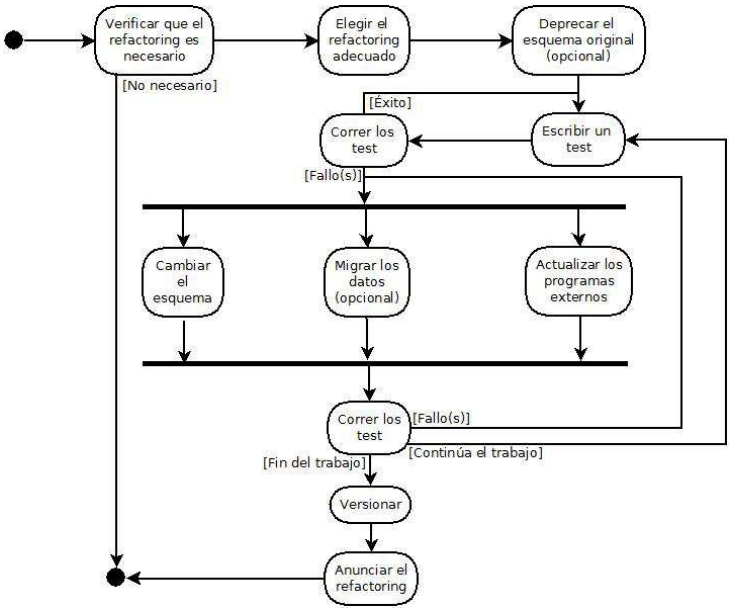

Figura 5: El proceso de refactoring de bases de datos

El proceso de refactoring de bases de datos consiste en trabajar iterativamente en las siguientes actividades:

Verificar que el refactoring es necesario. Se debe evaluar el impacto total del refactoring. Algunas cuestiones a considerar son: ¿Tiene sentido el refactoring? La falta de entendimiento puede llevar a creer que el diseño necesita un cambio. ¿Es el cambio realmente necesario ahora? ¿Se tiene claro el requerimiento? ¿Mejora el diseño? ¿Vale la pena el esfuerzo?

Elegir el refactoring adecuado: Se debe entender el problema que se está enfrentando y de ese modo se elige la acción adecuada.

Deprecar el esquema original: Se debe definir el periodo de transición. Durante el mismo, ninguna aplicación usará ambas versiones del esquema, en un principio trabajarán con la versión original y luego con la nueva versión. No todo refactoring necesitará periodo de transición, por ejemplo, un refactoring que mejoran la calidad de datos no lo necesitará.

Testing antes, durante y después: Se debe testear la lógica de negocio en la base de datos.

Modificar la base de datos: Es indispensable trabajar con scripts versionados, pequeños scripts para refactorings individuales. Cada uno debe tener asignado un

código de versión que identifique cada archivo con el script que implementa el refactoring.

Migración de datos: Este script debe tener el mismo identificador. Es una buena práctica no concentrarse en la calidad de datos cuando se trabaja con un refactoring estructural.

Modificar programas externos. ¿Qué hacer cuando alguna de las aplicaciones no se actualizará? Un camino a seguir es no hacer el refactoring. Otra opción es hacerlo igual, pero con un periodo de transición muy largo. De esta forma se tiene un diseño de mejor calidad para las aplicaciones que migren al nuevo diseño

Correr los test: Las tareas de testing deberán estar automatizadas tanto como sea posible.

Controles de versión sobre el trabajo.

Anunciar el refactoring. Implica también actualizar la documentación y el diseño lógico o físico.

3. La Herramienta propuesta

En [AS 06], los autores definen un catálogo de Refactorings de Base de Datos. Los mismos son presentados en forma de patrones; cada uno de ellos indica una descripción, una motivación, potenciales inconvenientes, mecanismos para modificar el esquema, mecanismos para migración de datos, y para actualizar los programas que acceden a la base. Entonces,

> ***"El objetivo de la herramienta que hemos desarrollado es poder automatizar las tareas de modificación del esquema y migración de datos planteadas en [AS 06] para llevar a cabo un refactoring de base de datos"***

La interacción de la herramienta con el usuario consiste en un *wizard* que permite implementar un refactoring de una manera simple e intuitiva. En la primera página se deberá indicar el modelo al que se le aplicará el refactoring y el refactoring a aplicar. En la siguiente página se deberán ingresar los datos dependiendo del refactoring elegido. Como resultado, la herramienta aplica los cambios sobre el modelo y genera los scripts pertinentes que modifican el esquema y migran datos de ser necesario.

Los refactorings implementados por la herramienta se detallan en el apéndice 1.

4.1 Ejemplo: Rename Table

Supongamos que el usuario cuenta con un modelo de base de datos, con una tabla `MOVIE` que desea renombrar por `FILM`. Para ello, en la primera página del *wizard* de la herramienta se deberá seleccionar el modelo al cual se le aplicará el refactoring y el refactoring, *Rename Table* en este caso. Luego click en *Next* para continuar con la siguiente página, donde se deberá suministrar el nombre de la tabla que se quiere renombrar y el nuevo nombre. Para terminar click en *Finish* y la herramienta mostrará un mensaje indicando que el refactoring fue aplicado correctamente. Luego el usuario podrá inspeccionar el modelo con el refactoring aplicado.

Figura 6: Selección del modelo de la base de datos, el refactoring, la tabla a renombrar y el nuevo nombre.

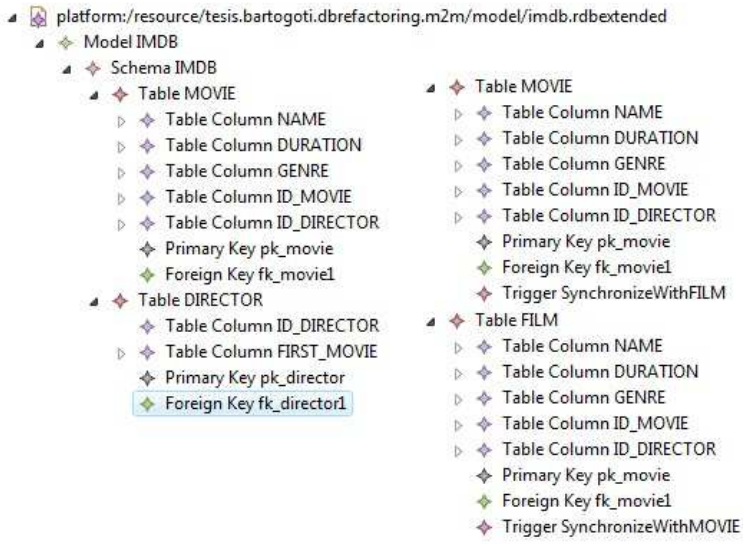

Figura 7: El modelo de base de datos antes y después de aplicar el refactoring

Se verifica que los cambios en el modelo son tal cual se indican en [AS 06] para este refactoring:

- Se creó una nueva tabla con el nombre `FILM`, con la misma estructura, campos, constraints e índices que la tabla original `MOVIE`.
- Se crearon dos triggers, `SynchronizeWithFILM` en la tabla `MOVIE` y `SynchronizeWithMOVIE` en la tabla `FILM`. Estos triggers mantienen sincronizados los datos en ambas tablas durante el período de transición.
- Se modificaron todas las Foreign Keys que antes apuntaban a alguna Unique Constraint de la tabla `MOVIE`, ahora apuntan a la correspondiente Unique Constraint de la tabla `FILM`. Tal es el caso de `fk_director1` que apuntaba a `pk_movie` de la tabla `MOVIE`, ahora apunta a `pk_movie` en la tabla `FILM`.

Por otro lado, la herramienta nos genera en el archivo `rename_table.sql` el conjunto de scripts para ejecutar sobre el esquema físico:

```
/* Se crea la tabla con el nuevo nombre, con las mismas
columnas, constraints e índices de la tabla vieja */
CREATE TABLE FILM (
	GENRE varchar(0) NOT NULL,
	ID_MOVIE long(20) NOT NULL,
	ID_DIRECTOR long(20) NOT NULL,
	DURATION int(0) NOT NULL,
	NAME varchar(0) NOT NULL,
	CONSTRAINT pk_movie PRIMARY KEY (ID_MOVIE),
	CONSTRAINT fk_movie1 FOREIGN KEY (ID_DIRECTOR) REFERENCES
DIRECTOR (ID_DIRECTOR)
);

/* Se crea el trigger de sincronización sobre la tabla nueva */
CREATE OR REPLACE TRIGGER SynchronizeWithMOVIE
BEFORE insert ON FILM

	REFERENCING OLD AS OLD NEW AS NEW
	FOR EACH ROW DECLARE
	BEGIN
		IF UPDATING THEN
			findAndUpdateIfNotFoundCreateMOVIE;
		END IF;

		IF INSERTING THEN
			createNewIntoMOVIE;
		END IF;

		IF INSERTING THEN
			deleteFromMOVIE;
		END IF;
```

```
	END;
	/

/* Se crea el trigger de sincronización sobre la tabla vieja */
CREATE OR REPLACE TRIGGER SynchronizeWithFILM
BEFORE insert ON MOVIE

	REFERENCING OLD AS OLD NEW AS NEW
	FOR EACH ROW DECLARE
	BEGIN
		IF UPDATING THEN
			findAndUpdateIfNotFoundCreateFILM;
		END IF;

		IF INSERTING THEN
			createNewIntoFILM;
		END IF;

		IF INSERTING THEN
			deleteFromFILM;
		END IF;
	END;
	/

/* Se copian los datos de la tabla vieja a la nueva */
INSERT INTO FILM
SELECT * FROM MOVIE;

/* Se borran las FKs que referencian a columnas de la tabla
vieja */
ALTER TABLE DIRECTOR DROP CONSTRAINT fk_director1;

/* Se vuelven a crear las FKs que referencian a columnas de la
tabla vieja, ahora apuntando a las mismas columnas pero de la
tabla nueva */
ALTER TABLE DIRECTOR ADD CONSTRAINT fk_director1 FOREIGN KEY
FIRST_MOVIE REFERENCES FILM (ID_MOVIE);
```

4.2 Diseño

La herramienta trabaja con un modelo de la base de datos independiente de la plataforma, que describe a la misma de manera abstracta y luego, a partir de este modelo y haciendo uso de mecanismos de transformación, obtenemos el código DDL, DML y SQL específico de una plataforma particular. Por otra parte, el refactoring también es aplicado sobre el modelo, por lo que ayuda a mantener un modelo lógico de la base de datos actualizado de la base de datos en todo momento.

Para la implementación de la misma elegimos la propuesta de Eclipse, en particular EMF, que tiene las siguientes ventajas: son un conjunto de herramientas de código abierto; la generación de código es posible a partir de la especificación de un modelo;

los modelos se especifican usando Ecore, versión simplificada del lenguaje de metamodelado MOF, lo cual establece un soporte para interoperabilidad con otras herramientas; JET permite fácilmente la transformación de un modelo Ecore en texto.

Para representar el modelo de una base de datos propusimos el metamodelo `rdbExtended` (Ver Apéndice 2). El metamodelo describe la sintaxis abstracta de un esquema de bases de datos y constituye la base para el procesamiento automatizado de los modelos.

La herramienta se compone de 4 plugins de Eclipse:

- **tesis.bartogoti.dbrefactoring.ui:** Define los aspectos de interfaz de usuario, es decir, la entrada de menú y el wizard.
- **tesis.bartogoti.dbrefactoring.m2m:** Contiene los archivos .qvto que definen las transformaciones de modelo a modelo. Habrá un archivo qvto por cada refactoring, por ejemplo: renameTable.qvto y renameColumn.qvto.
- **tesis.bartogoti.dbrefactoring.m2t:** Contiene los archivos .jet que definen las transformaciones de modelo a texto. Habrá un template JET por cada refactoring, por ejemplo: renameTable.jet y renameColumn.jet. Estos templates son los encargados de generar los scripts SQL.
- **tesis.bartogoti.dbrefactoring.jet.sqltaglib:** Este plugin define una librería de tags jet para generar código SQL. Desde los templates de JET se usan tags de esta librería como <createTable ...>, <createTrigger...>, etc.

En la actualidad la herramienta implementa 6 refactorings y genera código SQL sólo para ORACLE, pero cabe destacar que su diseño es escalable de manera que con poco esfuerzo es posible agregar nuevas implementaciones de refactorings así como soporte para otros motores de bases de datos.

4.2 A Futuro

Como trabajo futuro planeamos implementar las siguientes extensiones y mejoras:

- Extender la herramienta con más implementaciones de refactorings.
- Generar código para distintos proveedores de bases de datos.
- Mejorar la usabilidad.
- Generar un editor gráfico para el metamodelo. Se podría usar GMF en lugar de EMF.
- Extender la herramienta de manera que sirva para completar el refactoring una vez cumplido el período de transición.

4. Conclusiones

Los enfoques evolutivos de desarrollo, iterativos e incrementales por naturaleza, se convirtieron en el estándar de facto para el desarrollo de software moderno. Cuando un equipo decide tomar este enfoque, cada miembro debe trabajar en una forma evolutiva, incluyendo los profesionales encargados de las bases datos.

Las técnicas evolutivas incluyen refactoring, modelado de datos evolutivo, tests de regresión, gestión de la configuración de los artefactos y sandboxes para los desarrolladores.

De todas las técnicas evolutivas, destacamos el refactoring como la más importante. Para facilitar su aplicación sobre la base de datos se propone una herramienta extensible que genera el código DDL, DML y SQL necesario.

La herramienta se integra totalmente a un paradigma MDD, en el cual los modelos asumen un rol protagónico en el proceso de desarrollo del software y pasan de ser entidades contemplativas para convertirse en entidades productivas a partir de las cuales se deriva la implementación en forma automática.

Apéndice 1. Catálogo de refactorings implementados

Rename Column

Cambia el nombre a una columna existente en una tabla.

Tipo de refactoring: Estructural

Motivación: Las razones principales por las que se aplicará este refactoring son para mejorar la legibilidad del esquema de la base de datos, para adoptar convenciones de nombres en la organización o para permitir portabilidad de la base de datos, en caso que se esté usando una palabra reservada de otra plataforma a la que se quiera exportar la base de datos

Potenciales desventajas: Se debe analizar el costo de aplicar el refactoring a las aplicaciones externas que acceden a la columna contra la mejorada legibilidad y/o consistencia provista por el nuevo nombre.

Mecanismos de actualización del esquema: En primer lugar, se debe crear la nueva columna mediante la sentencia `ADD COLLUMN`. Luego se debe crear trigger que mantenga la sincronización entre las dos columnas. El trigger deberá ser invocado en cada cambio en los datos de la fila, teniendo en cuenta que no se creen ciclos.

Por otra parte, se deben tener en cuenta los siguientes aspectos:

- Si la columna a renombrar forma parte de la Clave Primaria (PK) de la tabla, la nueva columna pasará a formar parte de la Clave Primaria (PK) de la tabla.
- Si la columna a renombrar es referenciada por una Clave Foránea (FK) de otra tabla, la nueva columna pasará a formar parte de la Clave Foránea (FK) de la otra tabla.
- Si la columna a renombrar es forma parte de una Clave Única (UK) en la tabla, la nueva columna pasará a formar parte de la una Clave Única (UK).
- Si la columna a renombrar forma parte de algún índice en tabla, la nueva columna pasará a formar parte del índice.

Cada uno de estos cambios puede es también un refactoring. Estos cambios aseguran que cuando se elimine la tabla original no habrá dependencias a la misma.

Migración de datos: Se deberán copiar los datos de la columna original a la nueva columna.

Rename Table

Cambia el nombre a una tabla existente en la base de datos.

Tipo de refactoring: Estructural

Motivación: Los principales motivos por los que se aplicará este refactoring son para aclarar el significado de la tabla y su propósito dentro del esquema de la base de datos, o para adoptar convenciones de nombres en la organización. Idealmente, estos motivos deberán ser uno mismo.

Potenciales desventajas: Se debe analizar el costo de aplicar el refactoring a las aplicaciones externas que acceden a la tabla contra la mejorada legibilidad y/o consistencia provista por el nuevo nombre.

Durante el periodo de transición se deberá tener una tabla con el nombre original y otra con el nuevo nombre. Se deberá considerar el volumen de datos de tener la información replicada.

Mecanismos de actualización del esquema: Se deberá crear una nueva tabla mediante la sentencia `CREATE TABLE`. La nueva tabla incluirá las columnas con los nombres originales, al igual que las Claves Primarias (PK), Claves Foráneas (FK), Claves Únicas (UK) e índices de la tabla original.

Se deberá crear un trigger en la nueva tabla y en la tabla original que mantenga a las dos tablas sincronizadas. Cada cambio en una tabla debe estar reflejado en la otra durante el periodo de transición.

Por otra parte, si alguna columna de la tabla original es usada como una Clave Foránea (FK) de alguna otra tabla en el esquema, se deberán remplazar cada una de estas Claves Foráneas (FK) por nuevas que usen las columnas de la nueva tabla.

Migración de datos: Se deberán copiar los datos de la tabla original a la nueva tabla.

Rename View

Cambia el nombre a una vista existente en la base de datos.

Tipo de refactoring: Estructural

Motivación: Los principales motivos por los que se aplicará este refactoring son para mejorar la legibilidad del esquema de la base de datos, o para adoptar convenciones de nombres en la organización. Idealmente, estos motivos deberán ser uno mismo.

Potenciales desventajas: Se debe analizar el costo de aplicar el refactoring a las aplicaciones externas que acceden a la vista contra la mejorada legibilidad y/o consistencia provista por el nuevo nombre.

Mecanismos de actualización del esquema: Se debe crear la nueva vista mediante la sentencia SQL `CREATE VIEW`. En segundo lugar, hay que deprecar la vista original, para evitar que se apliquen redefiniciones o correcciones de bugs sobre la misma.

La vista original se debe redefinir de modo que use la definición de la nueva vista, para evitar código duplicado. De esta forma, cualquier cambio en la nueva vista será propagado a la vista original sin trabajo adicional.

Migración de datos: No se requiere migración de datos para este refactoring.

Replace One-to-many with Associative table

Remplaza una asociación uno a muchos entre dos tabla con una tabla asociativa.

Tipo de refactoring: Estructural

Motivación: La razón principal para introducir una tabla asociativa entre dos tablas es para implementar una relación muchos a muchos más adelante. Es algo común que una relación uno a muchos evolucione a una relación muchos a muchos. Debido a que una relación muchos a muchos es un subconjunto de una relación uno a muchos, no se está perdiendo semántica en el refactoring.

Potenciales desventajas: Si la asociación entre tablas no es probable que evolucione a una relación muchos a muchos, no es recomendable aplicar este refactoring, debido a que se está sobrecargando la base de datos al hacer los `JOINs` entre tablas para consultar la información necesaria. Esto puede provocar que se degrade la performance.

Mecanismos de actualización del esquema: Se debe crear la nueva tabla asociativa. Las columnas de esta tabla serán la combinación de las claves primarias de las otras dos tablas. Esta tabla no es necesario que tenga una columna como clave primaria, debido a que la misma se forma con las dos claves de las tablas existentes.

Luego, hay que deprecar la columna original. Para esto se debe indicar que la columna original que mantenía la relación uno a muchos será eliminada cuando finalice el periodo de transición.

Hay también que agregar un nuevo índice en la tabla con las columnas que forman la relación muchos a muchos. Si estas columnas forman la clave primaria, no será necesario crear el índice ya que los motores de bases de datos crean un índice sobre la clave primaria.

Se debe también crear los triggers de sincronización. Debe haber un trigger en la tabla original que mantenía la relación uno a muchos para que propague cualquier actualización en esta columna a la tabla nueva. También otro trigger es necesario para propagar una actualización en la tabla nueva hacia la tabla que mantiene la relación uno a muchos. Este ultimo trigger es necesario solo si la relación aún no ha migrado funcionalmente a una relación muchos a muchos.

Es aconsejable también tomar una convención de nombres para la nueva tabla que mantiene la relación muchos a muchos, como por ejemplo que la tabla se puede concatenar el nombre de las dos tablas separadas por un guión bajo "_".

Migración de datos: La tabla asociativa debe cargarse con los datos de la clave primaria de la tabla que mantiene la relación y el valor de la columna que referencia a la otra tabla.

Delete Table

Elimina una tabla existente en la base de datos.

Tipo de refactoring: Estructural

Motivación: Se debe aplicar cuando una tabla ya no es requerida o usada, ya sea porque fue reemplazada por otra fuente de datos similar, como otra tabla o vista, o finalizó el periodo de transición de un refactoring Rename Table, o simplemente porque la tabla ya no es necesaria. Idealmente, estos motivos deberán ser uno mismo.

Potenciales desventajas: Eliminar una tabla borra información de la base de datos, posiblemente sea necesario preservar algunos datos o toda la información. Si este es el caso, será necesario que los datos necesarios sean almacenados en otra fuente de datos.

Mecanismos de actualización del esquema: Se debe eliminar la tabla mediante la sentencia `DROP TABLE`.

Migración de datos: No se requiere migración de datos, a excepción los datos que se requieran guardar a discreción del usuario.

Make Column Non Nulleable

Cambia una columna existente de modo que no acepte valores nulos.

Tipo de refactoring: Calidad de datos

Motivación: Hay dos razones para aplicar este refactoring. La primera, es porque se quiere reforzar reglas de negocio a nivel de la base de datos de modo que cada aplicación que actualiza la columna es forzada a proveer un valor para la misma. La segunda, es para eliminar lógica de validación repetitiva ende las aplicaciones que implementan validaciones para no permitir valores nulos en el dato a mapear en la columna.

Potenciales desventajas: Todas las aplicaciones que actualizan datos en la tabla en cuestión deberán proveer un valor para la columna. Algunos programas probablemente asuman que la columna admite valores nulos y por lo tanto no proveerán dicho valor. Cuando una actualización o inserción ocurra, se debe asegurar que un valor es provisto, implicando que los programas sean actualizados o que la misma base de datos provea un valor por defecto.

Mecanismos de actualización del esquema: Para realizar el refactoring simplemente se debe hacer mediante la sentencia `ALTER TABLE`, aplicando la constraint `NOT NULL`.

Adicionalmente, se puede proveer un valor por defecto para la columna durante un tiempo de transición.

Migración de datos: Antes de aplicar el refactoring, se deberá asegurar que no existan valores nulos en la columna. Si los hay, se deberán remplazar por un valor no nulo adecuado.

Apéndice 2. El metamodelo rdbExtended y su editor

Una de las principales decisiones consistió en la elección del Metamodelo Relacional a emplear. Se requería que el mismo sea instancia de ECORE (por trabajar bajo la plataforma Eclipse) y que sea lo suficientemente robusto para representar modelos relacionales reales, que defina no sólo tablas y columnas, sino también

elementos como vistas, triggers, restricciones de integridad de distintos tipos, índices y tipos de datos. La plataforma Eclipse cuenta con el metamodelo relacional: `rdb.ecore`, bajo la URI: `http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb.`

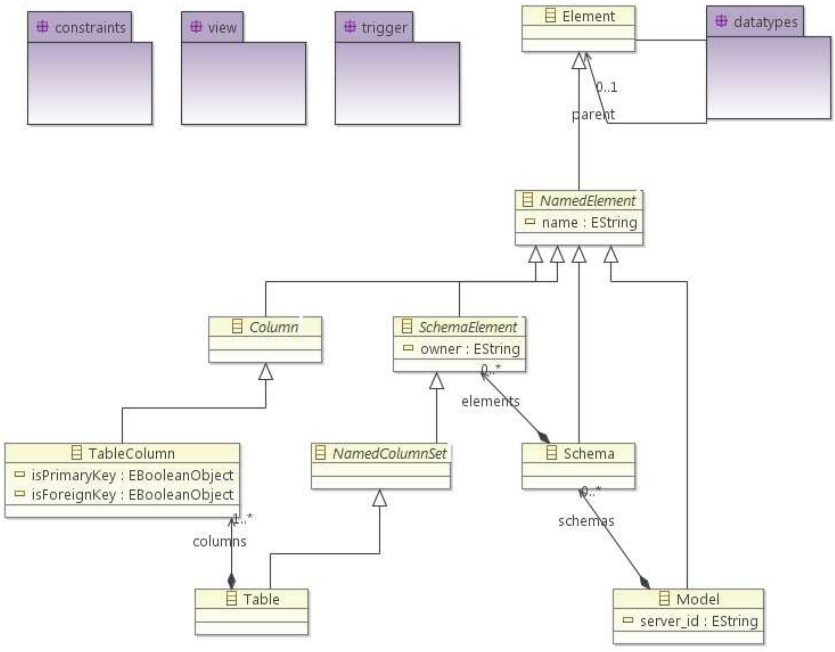

Figura 8: El metamodelo rdb.ecore: *Elements*

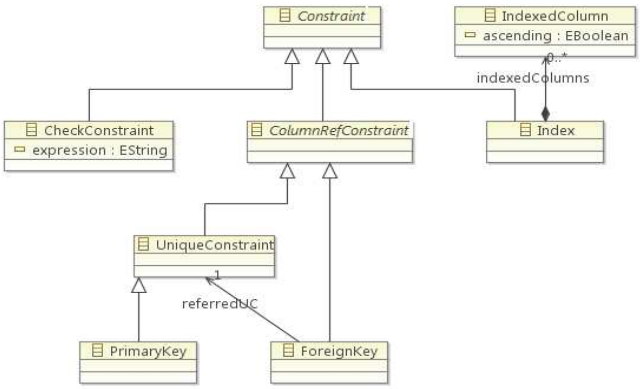

Figura 9: El metamodelo rdb.core: *Constraints*

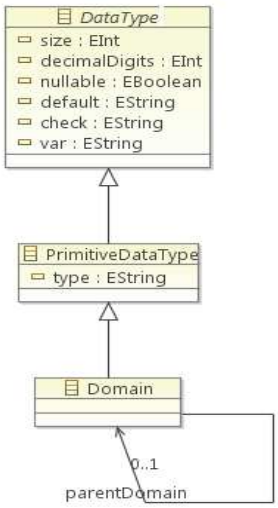

Figura 10: El metamodelo rdb.core: *DataTypes*

El mismo resulta ser acorde a los requerimientos de la herramienta. Por otra parte, se asemeja bastante en cuanto a nombres y estructura al paquete Relacional definido por el estándar CWM [CWM]. Sin embargo, carecía de un concepto fundamental para la implementación de los refactorings, el *Trigger*. Por tal motivo, se decidió extender el metamodelo dando origen a otro metamodelo que denominamos `rdbExtended.ecore`, con la URI `http://rdbExtended.ecore/rdbExtended`.

Así quedó definido el elemento Trigger en el nuevo metamodelo:

Figura 11: El elemento Trigger

Apéndice 3. Desarrollo de Software Dirigido por Modelos

El Desarrollo de Software Dirigido por Modelos (MDD) se ha convertido en un nuevo paradigma de desarrollo software. MDD promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes

herramientas. El adjetivo “dirigido” (driven) en MDD, a diferencia de “basado” (based), enfatiza que este paradigma asigna a los modelos un rol central y activo: son al menos tan importantes como el código fuente. Los modelos se van generando desde los más abstractos a los más concretos a través pasos de transformación y/o refinamientos, hasta llegar al código aplicando una última transformación. La transformación entre modelos constituye el motor de MDD.

Los modelos pasan de ser entidades contemplativas (es decir, artefactos que son interpretadas por los diseñadores y programadores) para convertirse en entidades productivas a partir de las cuales se deriva la implementación en forma automática.

A3.1 Modelos y Transformaciones

El modelo es una representación conceptual o física a escala de un proceso o sistema, con el fin de analizar su naturaleza, desarrollar o comprobar hipótesis o supuestos y permitir una mejor comprensión del fenómeno real al cual el modelo representa y así perfeccionar los diseños antes de iniciar la construcción de las obras u objetos reales.

MDD identifica 4 tipos de modelos:

- **El modelo independiente de la computación (CIM –** Computation Independent Model). Un CIM es una vista del sistema desde un punto de vista independiente de la computación. El CIM juega un papel muy importante en reducir la brecha entre los expertos en el dominio y sus requisitos por un lado, y los expertos en diseñar y construir artefactos de software por el otro.
- **El modelo independiente de la plataforma (PIM –** Platform Independent Model). Un PIM es un modelo con un alto nivel de abstracción que es independiente de cualquier tecnología o lenguaje de implementación. Dentro del PIM el sistema se modela desde el punto de vista de cómo se soporta mejor al negocio, sin tener en cuenta cómo va a ser implementado. Por lo tanto un PIM puede luego ser implementado sobre diferentes plataformas especificas.
- **El modelo específico de la plataforma (PSM –** Platform Specific Model). Como siguiente paso, un PIM se transforma en uno o más PSM. Un PSM representa la proyección de los PIMs en una plataforma específica. Un PIM puede generar múltiples PSMs, cada uno para una tecnología en particular.
- **El modelo de la implementación (Código).** El paso final en el desarrollo es la transformación de cada PSM a código fuente. Ya que el PSM está orientado al dominio tecnológico específico, esta transformación es bastante directa.

Anneke Kepple en [KWB 03] brinda las siguientes definiciones de transformación:

- Una transformación es la generación automática de un modelo destino desde un modelo fuente, de acuerdo a una definición de transformación.
- Una definición de transformación es un conjunto de reglas de transformación que juntas describen como un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino.
- Una regla de transformación es una descripción de como una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.

A3.2 La Arquitectura de 4 capas de modelado de OMG

El metamodelado es un mecanismo que permite definir formalmente lenguajes de modelado, como por ejemplo UML. La Arquitectura de cuatro capas de modelado es la propuesta del OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos[OMG].

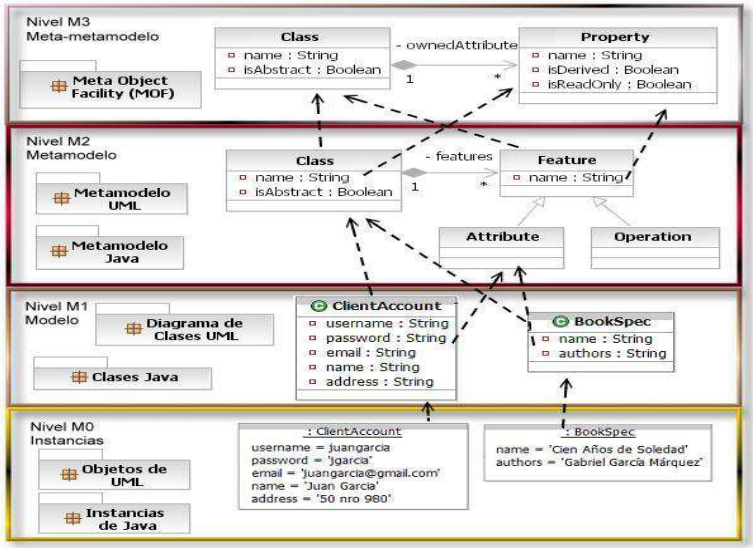

Figura 12: Arquitectura en cuatro capas de la OMG

Referencias

[AM 03]Ambler, Scott W. Agile Database Techniques: Effective Strategies for the Agile Software Developer. New York: John Wiley & Sons, 2003

[AM 04] Ambler, Scott W. The Object Premier, 3er Edition: Agile Model Driven Development with UML2. New York: Cambridge University Press, 2004

[AS 06] Scott W. Ambler, Pramod J. Sadalage. Refactoring Databases: Evolutionary Database Design. Addison Wesley Professional, 2006

[BK 03] Beck, Kent. Test Driven Development: By Example. Boston, Addison Wesley, 2003

[JET] Java Emitter Templates JET http://www.eclipse.org/projects/project.php?id=modeling.m2t.jet

[KWB 03] Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003

[FO 99] Fowler, Martin. Refactoring: Improving the Design of Existing Code. Addison Wesley Longman, 1999.

[OMG] OMG's MDA Web site. http://www.omg.org/mda/

[PGP 10] Claudia Pons, Roxana Giandini, Gabriela Pérez. Desarrollo de Software Dirigido por Modelos. Mac Graw Hill y Edulp. La Plata, 2010.

[QVT] Query/View/Transformation (QVT) Specification. Final Adopted Specification ptc/07-07-07. OMG. (2007)