# Refactoring de Bases de Datos

## Desarrollo Evolutivo de Bases de Bases de Datos Integrado con MDD

Tesina de Licenciatura

**Alumnos**
Bartolomeo, Mario Gustavo
Goti, Damián Alejandro

**Directora**
Dra. Claudia Pons

2013


Facultad de Informática

Universidad Nacional de La Plata

# Dedicatoria

*A mi familia*

*Gustavo Bartolomeo*

*A mis padres*

*Damián Goti*

# Agradecimientos

A mis padres por darme la libertad y confianza para desarrollar mis estudios, a la Facultad de Informática, sus docentes y compañeros que en todo momento colaboraron con mi formación académica. Al Lifia, compañeros de trabajo y amigos que día a día me permiten crecer como persona y como profesional.

Gustavo Bartolomeo

Agradezco a la Facultad de Informática y sus docentes, que cada uno desde su área y experiencia me formaron como profesional, al Lifia por capacitarme y darme mi primer trabajo en la industria y a Claudia Pons, por el apoyo fundamental para la realización de este trabajo.

Damián Goti

# ÍNDICE

# 1. INTRODUCCIÓN

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 tiempos, han ganando mucho terreno los procesos de software, también llamados metodologías, de naturaleza evolutivos y ágiles, cuyas principales premisas son el trabajo altamente colaborativo, iterativo e incremental. Para esto los programadores adoptaron técnicas como TDD (Test Driven Development, Desarrollo Dirigido por Tests) y AMDD (Agile Model Driven Development, Desarrollo Ágil Dirigido por Modelos), y se construyeron herramientas que facilitaron la aplicación de las mismas. Pero no ocurrió lo mismo en la comunidad de profesionales de bases de datos. Se podrían encontrar principalmente dos razones que explican esto:

1. Impedimentos culturales: Muchos de los actuales profesionales del área de datos se formaron durante los años 70 y 80, perdiendo la revolución de los objetos en los años 90 y la experiencia ganada en el desarrollo evolutivo.
2. Curva de aprendizaje: Lleva tiempo aprender nuevas técnicas, más aún si es necesario un cambio de mentalidad que permita pasar de procesos seriales o en cascada a evolutivos.

En 1999 Martin Fowler propuso la técnica de programación de refactoring en su libro [FO 99]. Un refactoring es un pequeño cambio en el código fuente que mejora el diseño sin cambiar su semántica. En otras palabras, se mejora la calidad del código sin cambiar ni añadir ningún comportamiento. Refactoring permite evolucionar el código lentamente con el tiempo, para tomar un enfoque evolutivo (iterativo e incremental) de programación. En su libro, Fowler discute la idea de 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.

Actualmente se necesita profundizar las técnicas y herramientas que también soporten el desarrollo evolutivo para las bases de datos; y en nuestra opinión la más importante de estas técnicas es el refactoring de base de datos.

En [AS 06] se presenta un marco teórico de cómo puede llevarse a cabo un esquema de trabajo evolutivo sobre las bases de datos. 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. De esta forma, las aplicaciones que usen la base de datos seguirán funcionando sin la necesidad de adaptarse inmediatamente a la nueva versión de la base datos.

De este modo, principal motivación consiste entonces en poder brindar una herramienta que automatice las tareas de refactoring de base de datos según los lineamientos establecidos en [AS 06].

Por otra parte, el Desarrollo de Software Dirigido por Modelos (MDD, por sus siglas en inglés: Model Driven software Development) promete mejorar el proceso de

construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas. 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 de pasos de transformación y/o refinamientos, hasta llegar al código aplicando una última transformación. La transformación entre modelos constituye el motor del MDD.

Obsérvese que conceptos como Modelos y Transformaciones se ajustan adecuadamente para las necesidades de nuestra herramienta. Los modelos nos representarán el lenguaje relacional y las bases de datos concretas. Las transformaciones entre modelos y de modelo a texto son exactamente las modificaciones a un modelo y la generación de los scripts correspondientes que implementan un refactoring a la base de datos.

El presente documento teórico está estructurado en tres partes fundamentales. En los capítulos 2, 3 y 4 se presenta una introducción a las teorías de MDD, desarrollo evolutivo y Refactoring de Base de Datos respectivamente. Estas teorías constituyen las áreas principales de investigación en las cuales se encuadra el presente trabajo teórico y la herramienta desarrollada. Los capítulos 5 y 6 están dedicados a la herramienta, las tecnologías empleadas, su descripción funcional, forma de uso y decisiones de diseño. Luego se muestran algunos trabajos relacionados al refactoring de bases de datos y se finaliza indicando las conclusiones sobre el trabajo realizado.

# 2. Desarrollo de Software Dirigido por Modelos (Conceptos Básicos)

## 2.1 Desarrollo Basado en Modelos (MBD)

Hacia finales de los años 70 Tom De Marco en su libro Structured Analysis and System Specification [DM 79] introdujo el concepto de desarrollo de software basado en modelos o MBD (por sus siglas en inglés Model Based Development). De Marco destacó que la construcción de un sistema de software debe ser precedida por la construcción de un modelo, tal como se realiza en otros sistemas ingenieriles. El modelo del sistema es una conceptualización del dominio del problema y de su solución. El modelo se focaliza sobre el mundo real: identificando, clasificando y abstrayendo los elementos que constituyen el problema y organizándolos en una estructura formal.

De esta forma, el modelo de un sistema provee un medio de comunicación y negociación entre usuarios, analistas y desarrolladores que oculta o minimiza los aspectos relacionados con la tecnología de implementación.

### *2.1.1 El ciclo de vida del desarrollo basado en modelos*

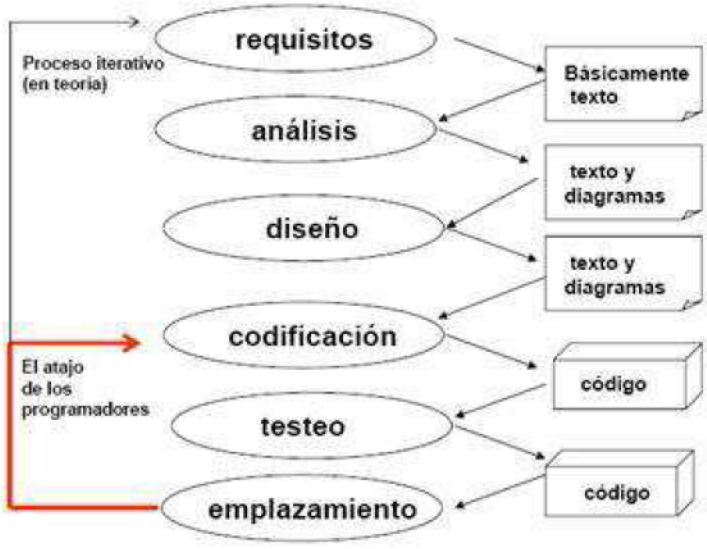


Figura 2.1: El clico de vida en el Desarrollo Basado en Modelos

En las primeras fases se construyen distintos modelos tales como los modelos de los requisitos (generalmente escritos en lenguaje natural), los modelos de análisis y los modelos de diseño (frecuentemente expresados mediante diagramas). Estas fases pueden realizarse de una manera iterativa-incremental o en forma de cascada.

### 2.1.2 ¿CUÁLES SON LOS PROBLEMAS DE MBD?

Si bien, los modelos constituyen una herramienta importante en las distintas fases del Desarrollo de Software, el paradigma MDB presenta algunos problemas que motivaron la aparición del Desarrollo Dirigido por Modelos (MDD). A continuación enumeramos dichos problemas:

***Mantenimiento de los modelos y del sistema:*** La conexión entre los diagramas y el código se va perdiendo gradualmente mientras se progresa en la fase de codificación. Como puede verse en la figura 2.1, los programadores suelen hacer los cambios sólo en el código, porque no hay tiempo disponible para actualizar los diagramas y otros documentos de alto nivel. Esto puede tener un impacto importante en el mantenimiento del sistema, especialmente luego de transcurrido un tiempo considerable desde la construcción del mismo.

***Mantenimiento de la documentación:*** Dada la complejidad de los sistemas de software actuales, resulta imprescindible contar con documentación en los distintos niveles de abstracción. Sin embargo, el problema de mantenimiento de los modelos, hace que también sea muy difícil mantener actualizada la documentación, sobre todo la que corresponde a las primeras fases de desarrollo.

***Falta de flexibilidad ante cambios tecnológicos:*** Sabemos que cada año (o en menos tiempo), aparecen nuevas tecnologías que rápidamente llegan a ser populares (a modo de ejemplo, se pueden listar JAVA, Linux, XML, HTML, UML, .NET, JSP, ASP, PHP, flash, servicios de Web, etc.). Estas nuevas tecnologías ofrecen beneficios concretos para las compañías y muchas de ellas no pueden quedarse atrás. Por lo tanto, tienen que pasar a utilizarlas cuanto antes. Como consecuencia del cambio, las inversiones en tecnologías anteriores pierden valor. Por consiguiente, el software existente debe migrar a una nueva tecnología, o a una versión nueva y diferente de la usada en su construcción. También puede darse otro caso en donde el software permanece utilizando la vieja tecnología, pero ahora necesita comunicarse con sistemas nuevos, los cuales sí han adoptado las nuevas tecnologías. Todas estas cuestiones nos plantean la existencia del problema de la flexibilidad tecnológica, al cual el desarrollo basado en modelos no aporta una solución integral.

## 2.2 DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS (MDD)

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 del 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.

Se pueden identificar los siguientes puntos claves de la iniciativa MDD:

***Abstracción:*** El enfoque de MDD para incrementar los niveles de abstracción es definir lenguajes de modelado específicos de dominio cuyos conceptos reflejen estrechamente los conceptos del dominio del problema, mientras se ocultan o minimizan los aspectos relacionados con las tecnologías de implementación. Estos lenguajes utilizan formas sintácticas que resultan amigables y que transmiten fácilmente la esencia de los conceptos del dominio. Por otra parte, el modelo permite reducir el impacto que la evolución tecnológica impone sobre el desarrollo de aplicaciones, permitiendo que el mismo modelo abstracto se materialice en múltiples plataformas de software.

***Automatización:*** La automatización es el método más eficaz para aumentar la productividad y la calidad. En MDD la idea es utilizar a las computadoras para automatizar tareas repetitivas que se puedan mecanizar, tareas que los seres humanos no realizan con particular eficacia. Esto incluye, entre otras, la capacidad de transformar modelos expresados mediante conceptos de alto nivel, específicos del dominio, en sus equivalentes programas informáticos ejecutables sobre una plataforma tecnológica específica. Además las herramientas de transformación pueden aplicar reiteradas veces patrones y técnicas con éxito ya comprobado, favoreciendo la confiabilidad del producto.

***Estándares:*** MDD debe ser implementado mediante una serie de estándares industriales abiertos. Estas normas proporcionan numerosos beneficios, como por ejemplo la capacidad para intercambiar especificaciones entre herramientas complementarias, o entre herramientas equivalentes de diferentes proveedores. Los estándares permiten a los fabricantes de herramientas centrar su atención en su principal área de experiencia, sin tener que recrear y competir con funcionalidades implementadas por otros proveedores. Por ejemplo, una herramienta que transforma modelos no necesita incluir una funcionalidad de edición de modelos. En lugar de ello, puede usar otra herramienta de edición de modelos de otro fabricante que se ajuste a un estándar común.

### 2.2.1 El Ciclo de Vida en el Desarrollo Dirigido por Modelos

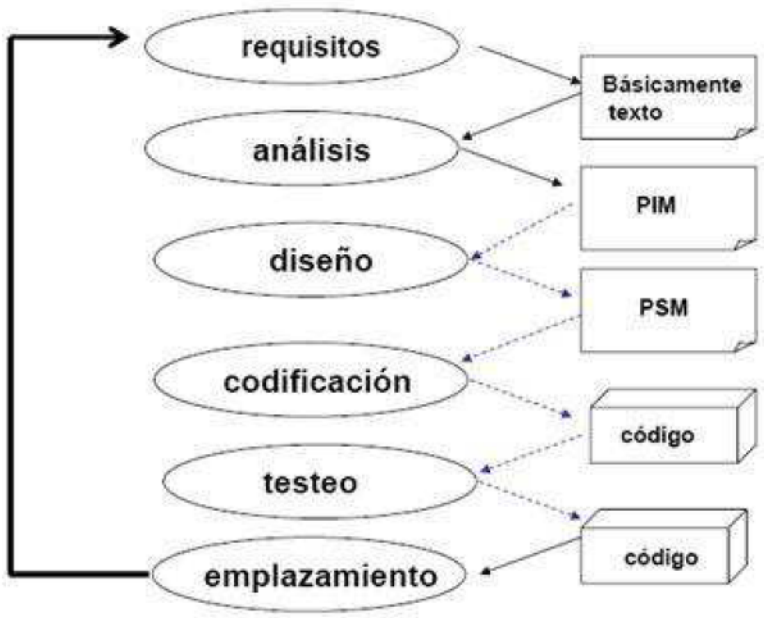


Figura 2.2: El ciclo de vida en el Desarrollo Dirigido por Modelos

Una de las mayores diferencias que presenta este ciclo de vida está en el tipo de los artefactos que se crean durante el proceso de desarrollo. Los artefactos son modelos formales, es decir, modelos que pueden ser comprendidos por una computadora. En la figura 2.2, las líneas punteadas señalan las actividades automatizadas en este proceso.

### 2.2.2 Tipos de modelos que identifica MDD

- Modelos con alto nivel de abstracción independientes de cualquier metodología computacional, llamados **CIMs** (Computational Independent Model),
- Modelos independientes de cualquier tecnología de implementación llamados **PIMs** (Platform Independent Model),
- Modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como **PSMs** (Platform Specific Model),
- Y finalmente modelos que representan el código fuente en sí mismo, identificados como **IMs** (Implementation Model).

Las transformaciones MDD son siempre ejecutadas por herramientas. Existen muchas herramientas que pueden transformar un PSM a código; no hay nada nuevo en eso. Dado que un PSM es un modelo muy cercano al código, esta transformación no es demasiado compleja. Lo novedoso que propone MDD es que las transformaciones entre modelos (por ejemplo de un PIM a PSMs) sean automatizadas.

Por otra parte, las técnicas de refinamiento en general se limitan a transformar un modelo formal en otro modelo formal escrito en el mismo lenguaje (es decir, se modifica el nivel de abstracción del modelo, pero no su lenguaje), mientras que MDD es

más amplio pues ofrece la posibilidad de transformar modelos escritos en distintos lenguajes (por ejemplo, podemos transformar un modelo escrito en UML en otro modelo escrito en notación Entidad-Relación).

### *2.2.3 Beneficios de MDD*

***Incremento en la productividad:*** MDD reduce los costos de desarrollo de software mediante la generación automática del código y otros artefactos a partir de los modelos, lo cual incrementa la productividad de los desarrolladores. Notemos que deberíamos sumar el costo de desarrollar (o comprar) mantener transformaciones, pero es esperable que este costo se amortice mediante el re-uso de dichas transformaciones.

***Adaptación a los cambios tecnológicos:*** el progreso de la tecnología hace que los componentes de software se vuelvan obsoletos rápidamente. MDD ayuda a solucionar este problema a través de una arquitectura fácil de mantener donde los cambios se implementan rápida y consistentemente, habilitando una migración eficiente de los componentes hacia las nuevas tecnologías. Los modelos de alto nivel están libres de detalles de la implementación, lo cual facilita la adaptación a los cambios que pueda sufrir la plataforma tecnológica subyacente o la arquitectura de implementación. Dichos cambios se realizan modificando la transformación del PIM al PSM. La nueva transformación es re aplicada sobre los modelos originales para producir artefactos de implementación actualizados. Esta flexibilidad permite probar diferentes ideas antes de tomar una decisión final. Y además permite que una mala decisión pueda fácilmente ser enmendada.

***Adaptación a los cambios en los requisitos:*** Poder adaptarse a los cambios es un requerimiento clave para los negocios, y los sistemas informáticos deben ser capaces de soportarlos. Cuando usamos un proceso MDD, agregar o modificar una funcionalidad de negocios es una tarea bastante sencilla, ya que el trabajo de automatización ya está hecho. Cuando agregamos una nueva función, sólo necesitamos desarrollar el modelo específico para esa nueva función; el resto de la información necesaria para generar los artefactos de implementación ya ha sido capturada en las transformaciones y puede ser re usada.

***Consistencia:*** la aplicación manual de las prácticas de codificación y diseño es una tarea propensa a errores. A través de la automatización MDD favorece la generación consistente de los artefactos.

***Reúso:*** en MDD se invierte en el desarrollo de modelos y transformaciones. Esta inversión se va amortizando a medida que los modelos y las transformaciones son reusados. Por otra parte el reúso de artefactos ya probados incrementa la confianza en el desarrollo de nuevas funcionalidades y reduce los riesgos ya que los temas técnicos han sido previamente resueltos.

***Mejoras en la comunicación con los usuarios:*** los modelos omiten detalles de implementación que no son relevantes para entender el comportamiento lógico del sistema. Por ello, los modelos están más cerca del dominio del problema, reduciendo la brecha semántica entre los conceptos que son entendidos por los usuarios y el lenguaje en el cual se expresa la solución. Esta mejora en la

comunicación influye favorablemente en la producción de software mejor alineado con los objetivos de sus usuarios.

***Mejoras en la comunicación entre los desarrolladores:*** los modelos facilitan el entendimiento del sistema por parte de los distintos desarrolladores. Esto da origen a discusiones más productivas y permite mejorar los diseños. Además, el hecho de que los modelos son parte del sistema y no sólo documentación, hace que los modelos siempre permanezcan actualizados y confiables.

***Captura de la experiencia:*** las organizaciones y los proyectos frecuentemente dependen de expertos quienes toman las decisiones respecto al sistema. Al capturar su experiencia en los modelos y en las transformaciones, otros miembros del equipo pueden aprovecharla sin requerir su presencia. Además este conocimiento se mantiene aun cuando los expertos se alejen de la organización.

***Los modelos son productos de larga duración:*** en MDD los modelos son productos importantes que capturan lo que el sistema informático de la organización hace. Los modelos de alto nivel son resistentes a los cambios a nivel plataforma y sólo sufren cambios cuando los requisitos del negocio lo hacen.

***Posibilidad de demorar las decisiones tecnológicas:*** cuando aplicamos MDD, las primeras etapas del desarrollo se focalizan en las actividades de modelado. Esto significa que es posible demorar la elección de una plataforma tecnológica específica o una versión de producto hasta más adelante cuando se disponga de información que permita realizar una elección más adecuada.

## 2.3 LOS PILARES DEL MDD: MODELOS, TRANSFORMACIONES Y HERRAMIENTAS

A grandes rasgos, podemos decir que el proceso MDD se apoya sobre los siguientes pilares:

- Modelos con diferentes niveles de abstracción, escritos en lenguajes bien definidos.
- Definiciones de como un modelo se transforma en otro modelo más específico.
- Herramientas de software que den soporte a la creación de modelos y su posterior transformación.

### *2.3.1 ¿QUÉ ES UN MODELO?*

Necesitamos una definición que sea lo suficientemente general para abarcar varios tipos diferentes de modelos, pero que al mismo tiempo sea lo suficientemente específica para permitirnos definir transformaciones automáticas de un modelo a otro modelo.

En el ámbito científico un modelo puede ser un objeto matemático (ej., un sistema de ecuaciones), un gráfico (ej., un plano) o un objeto físico (ej., una maqueta). 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 permitir así perfeccionar los diseños, antes de iniciar la construcción de las obras u objetos reales.

**Modelos a lo largo del proceso de desarrollo**

Durante el proceso de desarrollo de software diferentes modelos son creados (ver figura 2.3). Los modelos de análisis capturan sólo los requisitos esenciales del sistema de software, describiendo lo que el sistema hará independientemente de cómo se implemente. Por otro lado, los modelos de diseño reflejan decisiones sobre el paradigma de desarrollo (orientado a objetos, basado en componentes, orientado a aspectos, etc.), la arquitectura del sistema (distintos estilos arquitecturales) y los modelos de implementación describen como el sistema será construido en el contexto de un ambiente de implementación determinado (plataforma, sistema operativo, bases de datos, lenguajes de programación, etc.). Si bien algunos modelos pueden clasificarse claramente como un modelo de análisis, o de diseño o de implementación, por ejemplo, un Diagrama de Casos de Uso es un modelo de análisis, mientras que un Diagrama de Interacción entre objetos es un modelo de diseño y un Diagrama de Deployment es un modelo de implementación. En general, esta clasificación no depende del modelo en sí mismo sino de la interpretación que se dé en un cierto proyecto a las etapas de análisis, diseño e implementación.

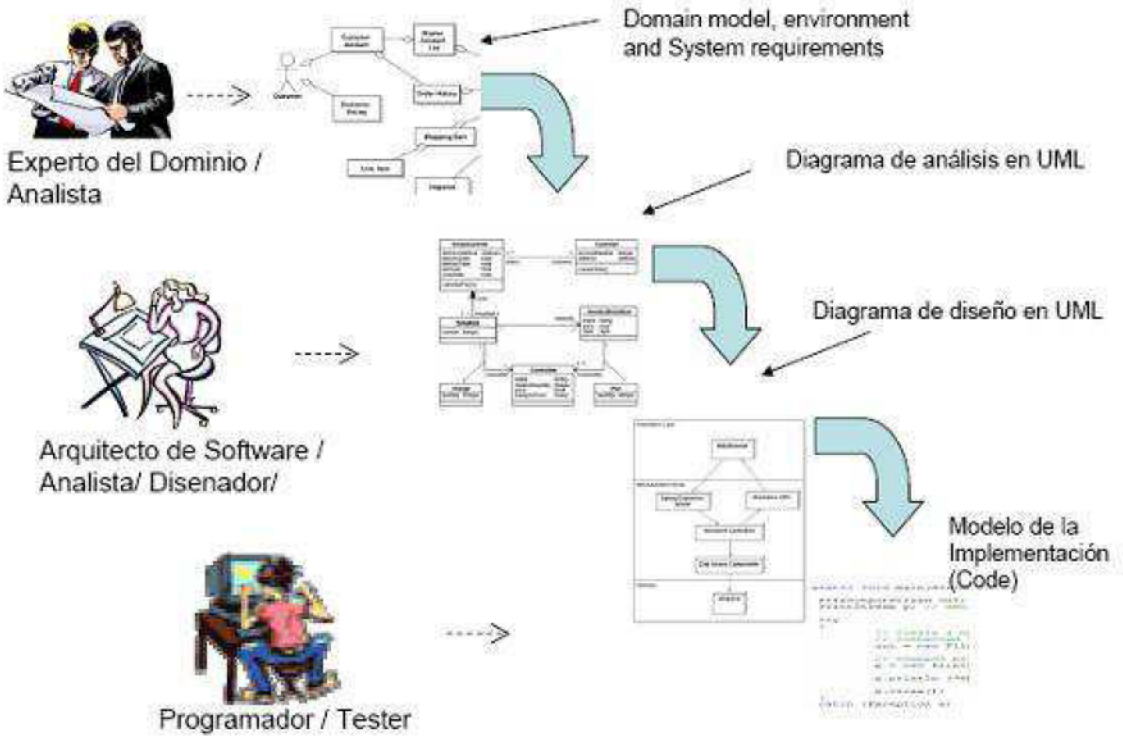


Figura 2.3. Diferentes modelos a lo largo del proceso de desarrollo de software

Según sus características y objetivos podemos realizar distintas clasificaciones de modelos:

- ***Modelos abstractos y detallados:*** Esta clasificación posee cierto grado de subjetividad. En general las herramientas de modelado nos permiten visualizar a un mismo modelo en distintos niveles de detalle agregando u

ocultando información como atributos, operaciones, cantidad de niveles de la jerarquía, etc.

- ***Modelos de Negocio y Modelos de Software:*** Los primeros describen un negocio o empresa (o parte de ellos). El lenguaje que se utiliza para construir el modelo del negocio contiene un vocabulario que permite al modelador especificar los procesos del negocio, los clientes, los departamentos, las dependencias entre procesos, etc. Este modelo de software, en cambio, es una descripción del sistema de software. El sistema de software y el sistema del negocio son conceptos diferentes en el mundo real, sin embargo los requisitos del sistema de software usualmente se derivan del modelo del negocio al cual el software brinda soporte.
- ***Modelos estáticos (o estructurales) y dinámicos (o de comportamiento):*** La mayoría de los sistemas poseen una base estructural estática, definida por el conjunto de objetos que constituyen el sistema, con sus propiedades y sus conexiones. Por ejemplo, un banco posee clientes y varias sucursales, en cada sucursal se radican varias cuentas bancarias, cada cuenta bancaria tiene un identificador y un saldo, etc. Por otra parte los sistemas poseen una dinámica definida por el comportamiento que los objetos del sistema despliegan. Por ejemplo, un cliente puede radicar una cuenta en una sucursal del banco y luego puede depositar dinero en su cuenta. Para representar estas dos vistas diferentes pero complementarias del sistema necesitamos modelos estáticos (o estructurales) por un lado y modelos dinámicos (o de comportamiento) por el otro.
- ***Modelos independientes de la plataforma y modelos específicos de la plataforma:*** Algunos modelos describen al sistema de manera independiente de los conceptos técnicos que involucra su implementación sobre una plataforma de software, mientras que otros modelos tienen como finalidad primaria describir tales conceptos técnicos. Teniendo en cuenta esta diferencia, los tipos de modelos que identifica MDD son:
  - ***El modelo independiente de la computación (CIM)*** (en inglés Computation Independent Model). Un CIM es una vista del sistema desde un punto de vista independiente de la computación. Un CIM no muestra detalles de la estructura del sistema. Usualmente al CIM se lo llama modelo del dominio y en su construcción se utiliza un vocabulario que resulta familiar para los expertos en el dominio en cuestión. 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)*** (en inglés, 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: ignora los sistemas operativos, los lenguajes de programación, el hardware, la topología de red, etc. Por lo tanto un PIM puede luego ser implementado sobre diferentes plataformas especificas.
  - ***El modelo específico de la plataforma (PSM)*** (en inglés, Platform Specific Model). Como siguiente paso, un PIM se transforma en uno o más PSM (Platform Specific Models). Un PIM representa la proyección de los PIMs en una plataforma específica. Un PIM puede generar múltiples PSMs, cada uno para una tecnología en particular. Generalmente, los

PSMs deben colaborar entre sí para una solución completa y consistente. Por ejemplo, un PSM para JAVA contiene términos como clase, interfaz, etc. Un PSM para una base de datos relacional contiene términos como tabla, columna, clave foránea, etc.

- ***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.

**Cualidades de los modelos**

El modelo de un problema es esencial para describir y entender el problema, independientemente de cualquier posible sistema informático que se use para su automatización. El modelo constituye la base fundamental de información sobre la que interactúan los expertos en el dominio del problema y los desarrolladores de software. Por lo tanto es de fundamental importancia que exprese la esencia del problema en forma clara y precisa. Por otra parte, la actividad de construcción del modelo es una parte crítica en el proceso de desarrollo. Los modelos son el resultado de una actividad compleja y creativa y por lo tanto son propensos a contener errores, omisiones e inconsistencias. La validación y verificación del modelo es muy importante, ya que la calidad del producto final dependerá fuertemente de la calidad de los modelos que se usaron para su desarrollo. Demos una mirada más detallada a las cualidades que esperamos encontrar en los modelos.

- ***Comprensibilidad***. El modelo debe ser expresado en un lenguaje que resulte accesible (es decir entendible y manejable) para todos sus usuarios.
- ***Precisión***. El modelo debe ser una fiel representación del objeto o sistema modelado. Para que esto sea posible, el lenguaje de modelado debe poseer una semántica precisa que permita la interpretación unívoca de los modelos. La falta de precisión semántica es un problema que no solamente atañe al lenguaje natural, sino que también abarca a algunos lenguajes gráficos de modelado que se utilizan actualmente.
- ***Consistencia***. El modelo no debe contener información contradictoria. Dado que un sistema es representado a través de diferentes submodelos relacionados debería ser posible especificar precisamente cual es la relación existente entre ellos, de manera que sea posible garantizar la consistencia del modelo como un todo.
- ***Completitud***. El modelo debe capturar todos los requisitos necesarios. Dado que en general, no es posible lograr un modelo completo desde el inicio del proceso, es importante poder incrementar el modelo. Es decir, comenzar con un modelo incompleto y expandirlo a medida que se obtiene más información acerca del dominio del problema y/o de su solución.
- ***Flexibilidad***. Debido a la naturaleza cambiante de los sistemas actuales, es necesario contar con modelos flexibles, es decir que puedan ser fácilmente adaptados para reflejar las modificaciones en el dominio del problema.
- ***Re-usabilidad***. El modelo de un sistema, además de describir el problema, también debe proveer las bases para el re-uso de conceptos y construcciones que se presentan en forma recurrente en una amplia gama de problemas.

- ***Corrección***. El análisis de la corrección del sistema de software debe realizarse en dos puntos. En primer lugar el modelo en sí debe ser analizado para asegurar que cumple con las expectativas del usuario. Este tipo de análisis generalmente se denomina "validación del modelo". Luego, asumiendo que el modelo es correcto, puede usarse como referencia para analizar la corrección de la implementación del sistema. Esto se conoce como "verificación del software". Ambos tipos de análisis son necesarios para garantizar la corrección de un sistema de software.

### 2.3.2 ¿Qué es una transformación?

El proceso MDD, descrito anteriormente, muestra el rol de varios modelos, PIM, PSM y código dentro del framework MDD. Una herramienta que soporte MDD, toma un PIM como entrada y lo transforma en un PSM. La misma herramienta u otra, tomará el PSM y lo transformará a código. Estas transformaciones son esenciales en el proceso de desarrollo de MDD. En la figura 2.4 se muestra la herramienta de transformación como una caja negra, que toma un modelo de entrada y produce otro modelo como salida:

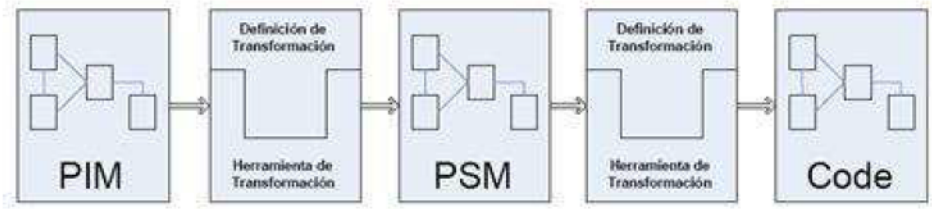


Figura 2.4: Herramienta de transformación como una caja negra

Si abriéramos la herramienta de transformación y mirásemos dentro, podríamos ver qué elementos están involucrados en la ejecución de la transformación. En algún lugar dentro de la herramienta hay una definición que describe como se debe transformar el modelo de entrada para producir el modelo destino. Esta es la definición de la transformación. La figura 2.4 muestra la estructura de la herramienta de transformación. Notemos que hay una diferencia entre la transformación misma, que es el proceso de generar un nuevo modelo a partir de otro modelo, y la definición de la transformación.

Para especificar la transformación, (que será aplicada muchas veces, independientemente del modelo fuente al que será aplicada) se relacionan construcciones de un lenguaje fuente en construcciones de un lenguaje destino. Se podría, por ejemplo, definir una transformación que relaciona elementos de UML a elementos JAVA, la cual describiría como los elementos JAVA pueden ser generados a partir de cualquier modelo UML.

En general, se puede decir que una definición de transformación consiste en una colección de reglas, las cuales son especificaciones no ambiguas de las formas en que un modelo (o parte de él) puede ser usado para crear otro modelo (o parte de él).

Las siguientes definiciones fueron extraídas del libro de Anneke Kepple [KWB 03]:

- *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.

### 2.3.3 HERRAMIENTAS DE SOPORTE PARA MDD

La puesta en práctica del proceso MDD requiere de la disponibilidad de herramientas de software que den soporte a la creación de modelos y transformaciones. La figura 2.5 brinda un panorama de los distintos puntos en los que el proceso MDD necesita ser soportado por herramientas. En particular, necesitamos los siguientes elementos:

- Editores gráficos para crear los modelos ya sea usando UML como otros lenguajes de modelado específicos de dominio;
- Repositorios para persistir los modelos y manejar sus modificaciones y versiones;
- Herramientas para validar los modelos (consistencia, completitud, etc.);
- Editores de transformaciones de modelos que den soporte a los distintos lenguajes de transformación, como QVT o ATL;
- Compiladores de transformaciones, *debuggers* de transformaciones.
- Herramientas para verificar y/o testear las transformaciones.

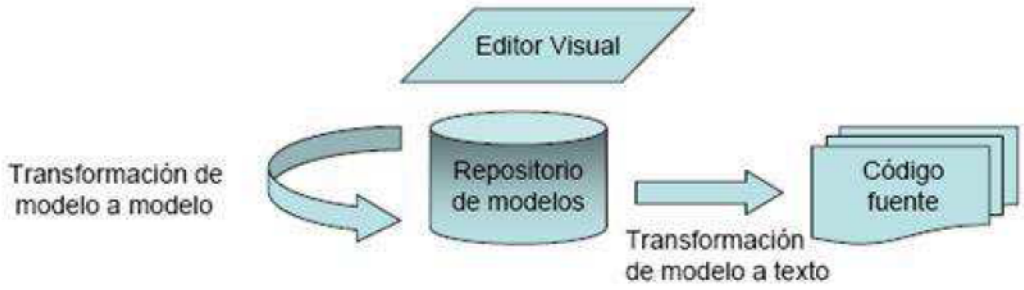


Figura 2.5: Puntos en los que el proceso MDD necesita ser soportado por herramientas

## 2.4 DEFINICIÓN FORMAL DE LENGUAJES DE MODELADO. EL ROL DEL METAMODELO

### *2.4.1 MECANISMOS PARA DEFINIR LA SINTAXIS DE UN LENGUAJE DE MODELADO*

Hace algunos años, la sintaxis de los lenguajes se definía casi exclusivamente usando Backus Naur Form (BNF). Este formalismo es una meta sintaxis usada para expresar gramáticas libres de contexto, es decir, una manera formal de describir cuales son las palabras básicas del lenguaje y cuales secuencias de palabras forman una expresión correcta dentro del lenguaje. Una especificación en BNF es un sistema de reglas de la derivación.

El BNF se utiliza extensamente como notación para definir la sintaxis (o gramática) de los lenguajes de programación. Por ejemplo, las siguientes expresiones BNF definen la sintaxis de un lenguaje de programación simple:

```
<Programa> ::= begin <Comando> end
<Comando> ::= <Asignación>|<Loop>|<Decisión>|<Comando>;<Comando>
<Asignación> ::=  variableName := <Expresión>
<Loop> ::= while <Expresión> do <Comando> end
<Decisión> ::= if<Expresión>then<Comando>else<Comando>endif
<Expresión> ::= ...
```

Este método es útil para lenguajes textuales, pero dado que los lenguajes de modelado en general no están basados en texto, sino en gráficos, es conveniente recurrir a un mecanismo diferente para definirlos.

Por lo tanto, en los últimos años se desarrolló una técnica específica para facilitar la definición de los lenguajes gráficos, llamada *metamodelado*. Veamos en qué consiste esta nueva técnica: usando un lenguaje de modelado, podemos crear modelos; un modelo especifica que elementos pueden existir en un sistema. Si se define la clase Persona en un modelo, se pueden tener instancias de Persona como Juan, Pedro, etc. Por otro lado, la definición de un lenguaje de modelado establece que elementos pueden existir en un modelo. Por ejemplo, el lenguaje UML establece que dentro de un modelo se pueden usar los conceptos Clase, Atributo, Asociación, Paquete, etc. Debido a esta similitud, se puede describir un lenguaje por medio de un modelo, usualmente llamado *metamodelo*. El metamodelo de un lenguaje describe que elementos pueden ser usados en el lenguaje y como pueden ser conectados.

Como un metamodelo es también un modelo, el metamodelo en sí mismo debe estar escrito en un lenguaje bien definido. Este lenguaje se llama metalenguaje. Desde este punto de vista, BNF es un metalenguaje.  En la figura 2.6 se muestra gráficamente esta relación.

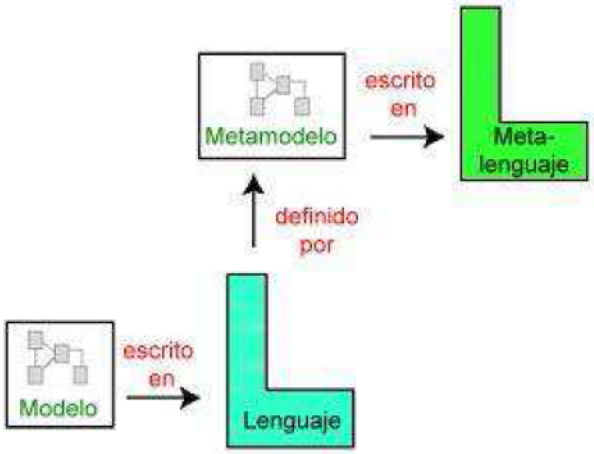


Figura 2.6: Relación entre modelo, lenguaje, metamodelo y metalenguaje

### 2.4.2 La arquitectura de 4 capas de modelado de OMG

El metamodelado es entonces 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.

Los cuatro niveles definidos en esta arquitectura se denominan: M3, M2, M1, M0 y los describimos a continuación. Para entender mejor la relación entre los elementos en las distintas capas, presentamos un ejemplo utilizando el lenguaje UML. En este ejemplo modelamos un sistema de venta de libros por Internet, que maneja información acerca de los clientes y de los libros de los cuales se dispone.

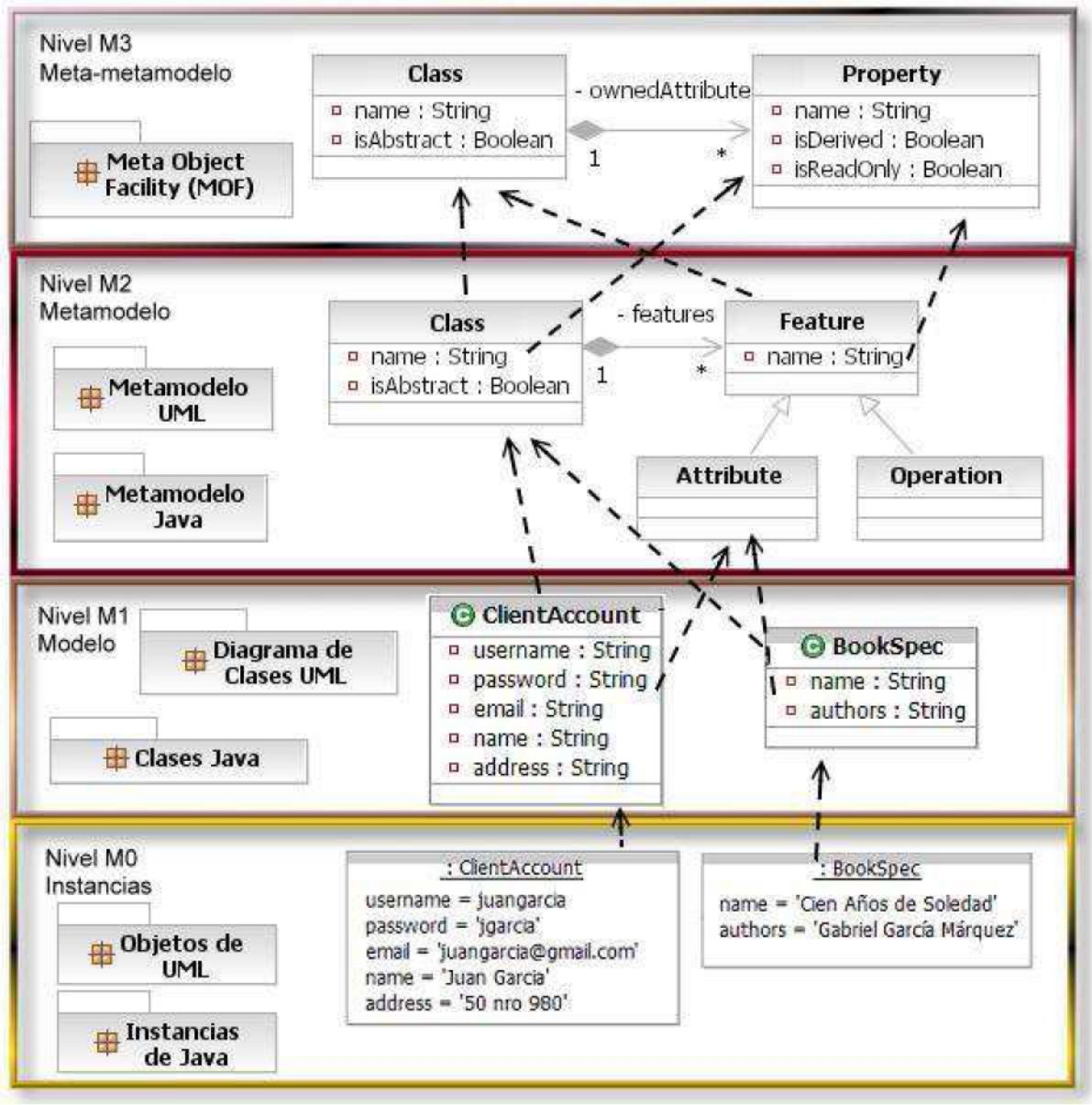


Figura 2.7: Arquitectura en cuatro capas de la OMG

**Nivel M0: Instancias**

En el nivel M0 se encuentran todas las instancias "reales" del sistema, es decir, los objetos de la aplicación. Aquí no se habla de clases, ni atributos, sino de entidades físicas que existen en el sistema.

En la figura pueden verse dos entidades. Un cliente, Juan García, del cual queremos guardar su nombre de usuario, su palabra clave, su nombre real, su dirección postal y su dirección de e-mail. Y un libro con título 'Cien Años de Soledad' de la cual se conoce su autor.

**Nivel M1: Modelo del sistema**

Por encima de la capa M0 se sitúa la capa M1, que representa el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0. Es decir, cada elemento de M0 es una instancia de un elemento de

M1. Sus elementos son modelos de los datos. En el nivel M1 aparecen entonces las entidades Cliente y Libro con sus atributos.

El objeto 'Juan García' puede verse ahora como una instancia de la entidad Cliente y 'Cien Años de Soledad' como una instancia de Libro.

**Nivel M2: Metamodelo**

Análogamente a lo que ocurre con las capas M0 y M1, los elementos del nivel M1 son a su vez instancias del nivel M2. Esta capa recibe el nombre de metamodelo. En la figura se muestra una parte del metamodelo UML. En este nivel aparecen conceptos tales como Clase (Class), Atributo (Attribute) y Operación (Operation).

Siguiendo el ejemplo, la entidad ClientAccount será una instancia de la metaclase Class del metamodelo UML. Esta instancia tiene cinco objetos relacionados a través de la meta asociación feature, por ejemplo una instancia de Attribute con `name="username"` y `type="String"` y otra instancia de Attribute con `name="password"` y `type="String"`.

**Nivel M3: Meta-metamodelo**

De la misma manera podemos ver los elementos de M2 como instancias de otra capa, la capa M3 o capa de meta-metamodelo. Un meta-metamodelo es un modelo que define el lenguaje para representar un metamodelo. La relación entre un meta-metamodelo y un metamodelo es análoga a la relación entre un metamodelo y un modelo.

M3 es el nivel más abstracto, que permite definir metamodelos concretos. Dentro del OMG, MOF es el lenguaje estándar de la capa M3. Esto supone que todos los metamodelos de la capa M2, son instancias de MOF.

No existe otro metanivel por encima de MOF. Básicamente, el MOF se define a sí mismo.

### *2.4.3 El uso del metamodelado en MDD*

Las razones por las que el metamodelado es tan importante en MDD son:

- En primer lugar, necesitamos contar con un mecanismo para definir lenguajes de modelado sin ambigüedades [CESW 04] y permitir que una herramienta de transformación pueda leer, escribir y entender los modelos;
- Luego, las reglas de transformación que constituyen una definición de una transformación describen como un modelo en un lenguaje fuente puede ser transformado a un modelo en un lenguaje destino. Estas reglas usan los metamodelos de los lenguajes fuente y destino para definir la transformación;
- Y finalmente, la sintaxis de los lenguajes en los cuales se expresan las reglas de transformación también debe estar formalmente definida para permitir su

automatización. En este caso también se utilizará la técnica de metamodelado para especificar su sintaxis.

La figura 2.8 muestra como se completa MDD con la capa de metamodelado. La parte baja de la figura es con lo que la mayoría de los desarrolladores trabaja habitualmente. En el centro se introduce el metalenguaje para definir nuevos lenguajes. Un pequeño grupo de desarrolladores, usualmente con más experiencia, necesitarán definir lenguajes y las transformaciones entre estos lenguajes. Para este grupo, entender el metanivel será esencial a la hora de definir transformaciones.

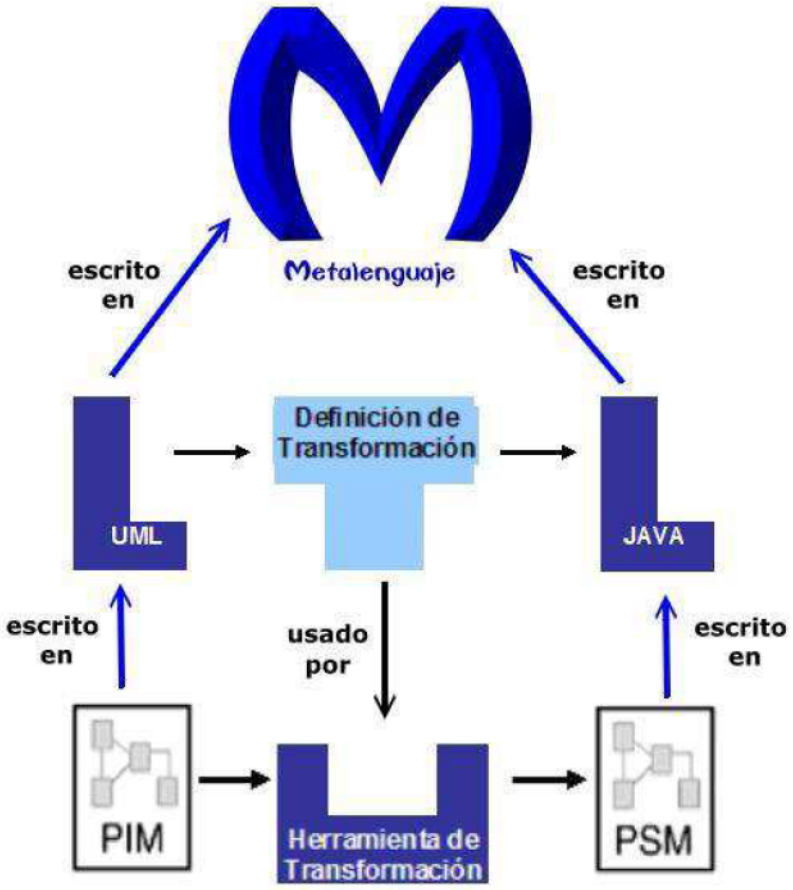


Figura 2.8: MDD y el modelado

### 2.4.4 El lenguaje de modelado más abstracto: MOF

El lenguaje MOF, acrónimo de Meta-Object Facility, es un estándar del OMG para la ingeniería conducida por modelos. Como se vio anteriormente, MOF se encuentra en la capa superior de la arquitectura de 4 capas. Provee un meta-meta lenguaje que permite definir metamodelos en la capa M2. El ejemplo más popular de un elemento en la capa M2 es el metamodelo UML, que describe al lenguaje UML.

Esta es una arquitectura de metamodelado cerrada y estricta. Es cerrada porque el metamodelo de MOF se define en términos de sí mismo. Y es estricta porque cada elemento de un modelo en cualquiera de las capas tiene una correspondencia estricta con un elemento del modelo de la capa superior.

Tal como su nombre lo indica, MOF se basa en el paradigma de Orientación a Objetos. Por este motivo usa los mismos conceptos y la misma sintaxis concreta que los diagramas de clases de UML.

Actualmente, la definición de MOF está separada en dos partes fundamentales, EMOF (*Essential* MOF) y CMOF (*Complete* MOF), y se espera que en el futuro se agregue SMOF (*Semantic* MOF). La figura 2.9 presenta los principales elementos contenidos en el paquete EMOF.

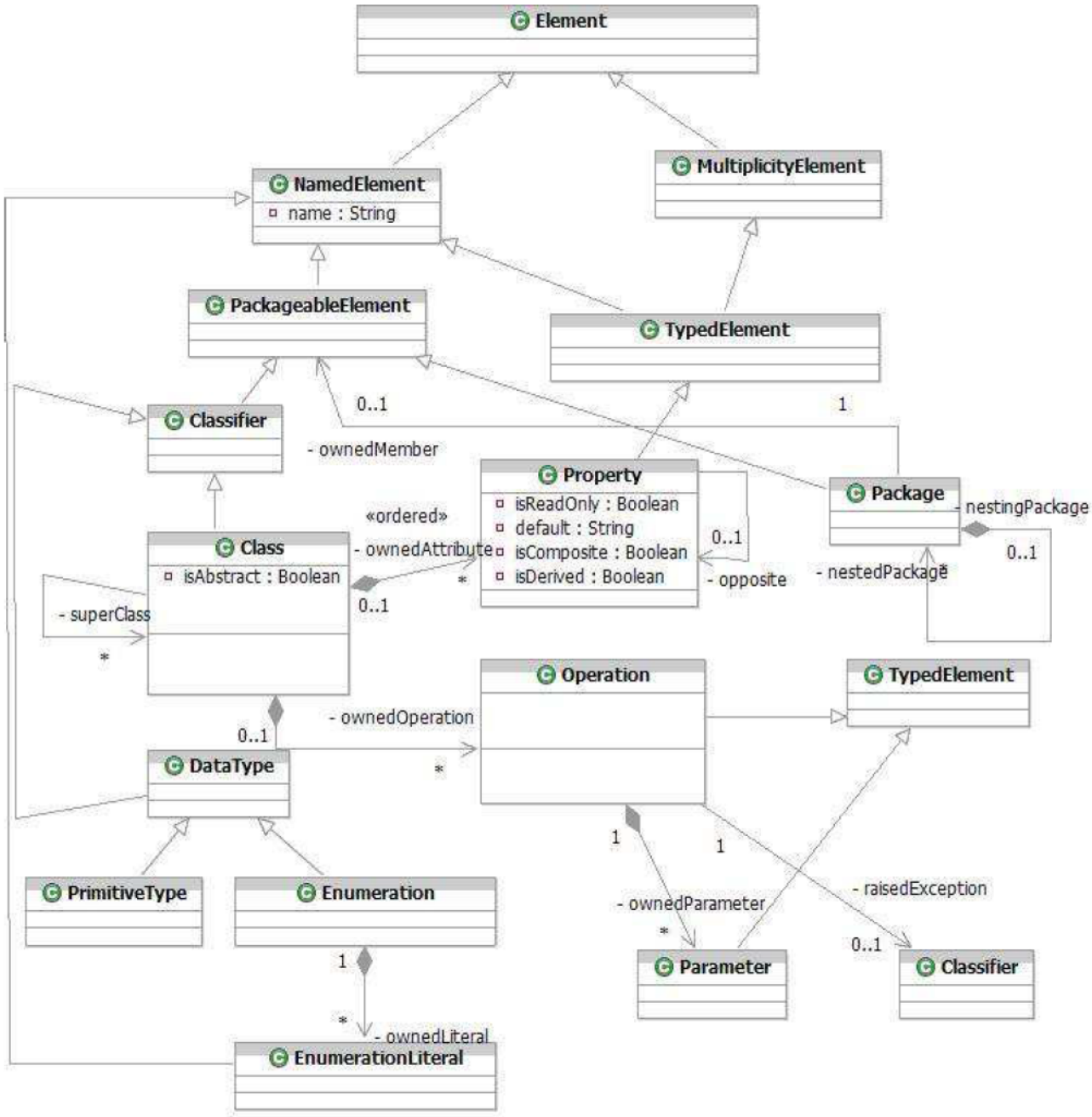


Figura 2.9: Principales elementos contenidos en el paquete EMOF

### 2.4.5 Implementación de MOF: Ecore

El metamodelo MOF está implementado mediante un plugin para Eclipse [Eclipse] llamado Ecore [EMF]. Este plugin respeta las metaclases definidas por MOF. Todas las metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra "E", indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa a la metaclase Class de MOF.

La primera implementación de Ecore fue terminada en Junio del 2002. Esta primera versión usaba como meta-metamodelo la definición estándar de MOF (v1.4). Gracias a las sucesivas implementaciones de Ecore y basado en la experiencia ganada con el uso de esta implementación en varias herramientas, el metalenguaje evolucionó. Como conclusión, se logró una implementación de Ecore realizada en JAVA, más eficiente y con sólo un subconjunto de MOF, y no con todos los elementos como manipulaban las implementaciones hechas hasta ese momento. A partir de este conocimiento, el grupo de desarrollo de Ecore colaboró más activamente con la nueva definición de MOF, y se estableció un subconjunto de elementos como esenciales, llegando a la definición de EMOF, que fue incluida como parte del MOF (v2.0).

MOF y Ecore son conceptualmente muy similares, ambos basados en el concepto de clases con atributos tipados y operaciones con parámetros y excepciones. Además ambos soportan herencia múltiple y utilizan paquetes como mecanismo de agrupamiento.

# 3. Desarrollo Evolutivo y Refactoring

## 3.1 Metodologías ágiles

Los procesos modernos de desarrollo, también llamados metodologías, son evolutivos por naturaleza, requiriendo trabajo iterativo e incremental. Como ejemplos de estas metodologías podemos mencionar Rational Unified Process (RUP), Extreme Programming (XP), Scrum, Dynamic System Development Method (DSDM), Team Software Process (TSP), Agile Unified Process (AUP), Enterprise Unified Process (EUP), por nombrar algunos. 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 ese diseño, 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 releases, más bien que un gran release. Estos métodos y técnicas se han desarrollado, han evolucionado y se han pulido en una forma popular en la industria del software, en lugar de formularse en torres de marfil. Y como resultado, estas técnicas evolutivas y ágiles parecen funcionar muy bien en la práctica.

La mayoría, por no decir todas, de las metodologías modernas son ágiles, las cuales se caracterizan como evolutivas y altamente colaborativas por naturaleza. Cuando un equipo toma un enfoque colaborativo, sus miembros se esfuerzan activamente por encontrar formas de trabajar juntos efectivamente. De hecho, se intenta asegurar que las partes interesadas como el cliente son miembros activos del equipo. En general, se dice que se conseja adoptar la técnica de comunicación más caliente aplicable a cada situación: preferir la comunicación cara a cara alrededor de una pizarra sobre una llamada telefónica, preferir una llamada telefónica a enviar un e-mail, preferir un e-mail a enviar un documento detallado. Cuanto mejor sea la comunicación y la colaboración en el equipo de desarrollo, mayores van a ser las chances de éxito.

Un proceso es considerado ágil cuando se ajusta a los cuatro valores de la Agile Alliance [AA 01a]. Los valores definen preferencias, no alternativas, alentando un foco en ciertas áreas pero no eliminando otras. En otras palabras, mientras se deben valorar los conceptos mencionados en la siguiente lista en el lado derecho, se deben valorar los conceptos del lado izquierdo aún más. Por ejemplo, los procesos y herramientas son importantes, pero las personas y las interacciones son más importantes. Los cuatro valores son:

- Personas e interacciones ***SOBRE*** procesos y herramientas. El factor más importante que se necesita considerar son las personas y como trabajan juntas. Si no se toma en cuenta este hecho, las mejores herramientas y procesos no serán de ninguna utilidad.
- Software funcionando ***SOBRE*** exhaustiva documentación. El objetivo principal del desarrollo de software es crear software que funcione acorde a las necesidades de las partes interesadas. La documentación aún tiene su lugar, escrita apropiadamente describe cómo y por qué el sistema fue construido, y como se trabajará con el sistema.

- Colaboración con el cliente ***SOBRE*** negociaciones de contratos. Sólo el cliente puede decir lo que quiere. Desafortunadamente, no son buenos en esto. Ellos probablemente no tienen las habilidades para especificar un sistema precisamente, ni van a hacer las cosas bien en un primer momento, y peor aún, probablemente cambiarán su opinión con el tiempo. Tener un contrato con el cliente es importante, pero un contrato no es un sustituto de una efectiva comunicación. Los profesionales IT exitosos trabajan cerca con el cliente, invierten el esfuerzo necesario para descubrir lo que el cliente necesita y educan al cliente a lo largo del camino.
- Responder al cambio ***SOBRE*** seguir un plan. En tanto el trabajo progresa en el sistema, el entendimiento de lo que quieren las partes interesadas cambia, el entorno del negocio cambia, la tecnología subyacente cambia. El cambio es una realidad en el desarrollo de software y como resultado el plan del proyecto y el enfoque total deben reflejar el entorno cambiante.

## 3.2 REFACTORING

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. Un ejemplo de refactoring podría ser renombrar una operación o una variable con otro nombre que indique más claramente su propósito. Para implementar un refactoring que cambia el nombre de una operación se debe cambiar el nombre de la misma y luego cambiar cada invocación a la misma a lo largo de todo el código fuente. Un refactoring no está completo hasta que el código corra igual a como lo hacía antes de aplicar el mismo.

Claramente se necesita una forma sistemática para hacer refactoring de código, incluyendo herramientas y técnicas para hacer eso. Muchos entornos de desarrollo integrados (IDE, Integrated Development Environment) actualmente soportan refactoring de código, lo que es un buen comienzo. Sin embargo, para hacer refactoring en la práctica, se necesita desarrollar un conjunto de tests de integración que validen que el código siga funcionando como antes. No se tendrá la confianza necesaria para hacer refactoring si no se puede asegurar este hecho.

Los desarrolladores que trabajan con metodologías ágiles consideran el refactoring como una técnica primaria de programación. Es tan común hacer un refactoring como introducir una sentencia `if` o `loop` en el código. Se debe hacer refactoring en el código porque se trabaja con mayor productividad cuando el código es de mayor 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.

La adopción de técnicas ágiles y refactoring trae las siguientes ventajas:

- Se minimiza el trabajo en vano. Un enfoque evolutivo permite evitar pérdidas inherentes a las técnicas en cascada cuando cambia un requerimiento. Una temprana inversión en requerimientos detallados, arquitectura y diseño de artefactos es perdida cuando un requerimiento es tardíamente encontrado.
- Se evita rehacer el mismo trabajo. Con las técnicas evolutivas también se necesita hacer algún modelado inicial de los principales requerimientos, los que pueden llevar a rehacer significativa cantidad de trabajo en caso que fueran detectados tardíamente. Sin embargo no se investigan los detalles tempranamente.
- Siempre se tiene un sistema funcionando. Con un enfoque evolutivo, regularmente se produce software funcionando. Cuando se tiene una nueva versión funcionando del sistema cada una o dos semanas, aunque sea en un ambiente para demo, se reducen drásticamente los riesgos del proyecto.
- Siempre se sabe que se tiene el mejor diseño posible. Este es el punto clave de lo que se trata el refactoring: mejorar el diseño en pequeños pasos cada vez.
- Se trabaja en forma compatible con los desarrolladores. Los desarrolladores de aplicaciones trabajan en una forma evolutiva, y si los desarrolladores de bases de datos pretenden formar parte de un equipo de desarrollo moderno, deben trabajar de una forma evolutiva.
- Se reduce el esfuerzo global. Trabajando en una forma evolutiva, se hace sólo el trabajo que realmente se necesita hoy y nada más.

## 3.3 DESARROLLO EVOLUTIVO DE BASES DE DATOS

Aunque las formas evolutivas y ágiles de trabajar han sido fácilmente adoptadas dentro de la comunidad de desarrollo, lo mismo no podemos decir de la comunidad de bases de datos. Muchas de las técnicas orientadas a datos son en cascada por naturaleza, requiriendo la creación de modelos bastante detallados antes que la implementación sea "permitida" comenzar. 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 está el problema: las técnicas comunes de desarrollo de bases de datos no reflejan las realidades de los procesos modernos de desarrollo de software. Esto no debería ser así.

En cuanto al desarrollo evolutivo de bases de datos, podemos decir que en lugar de intentar diseñar el esquema de la base de datos tempranamente en un proyecto, se construye a través de la vida del proyecto, para reflejar los cambios de requerimientos. Es un hecho que los requerimientos cambiarán en tanto el proyecto avance. Los enfoques tradicionales han negado esta realidad fundamental y han intentado gestionar

el cambio. Los profesionales que aplican las técnicas de desarrollo modernas, en lugar de aceptar 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 que las mismas técnicas y herramientas se pueden aplicar al proceso evolutivo de desarrollo de bases de datos.

El segundo paso para profesionales y administradores de bases de datos es adoptar nuevas técnicas que permitan trabajar en una manera evolutiva. A continuación mencionamos brevemente estas técnicas que se desarrollaran durante el presente trabajo y en nuestra opinión la más importante es el refactoring de bases de datos, la cual es nuestro foco. Aunque el refactoring de bases de datos no es una técnica de desarrollo ágil, es una técnica primaria para desarrolladores ágiles. Las técnicas evolutivas en el desarrollo de bases de datos son:

- Refactoring de bases de datos
- Modelado de datos evolutivo
- Tests de regresión de bases de datos
- Gestión de la configuración de los artefactos de la base de datos
- Sandboxes para desarrolladores

### 3.3.1 Refactoring de bases de datos

En el libro Refactoring [FO 99] se discute la idea de 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.

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. 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.

La motivación para aplicar un refactoring a la base de datos es la misma que para el código fuente de una aplicación. Se hace un refactoring a la base de datos para que sea más fácil agregar algo nuevo al mismo. De esta forma de a pequeños, pero continuos pasos se mejora el diseño del esquema de la base de datos, haciéndola más fácil de entender y evolucionar.

### 3.3.2 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 3.1. 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 a través de los mayores requerimientos tempranamente en el proyecto sin tener en cuenta los detalles. Con estos detalles se debe trabajar en el momento que se requiera su implementación.

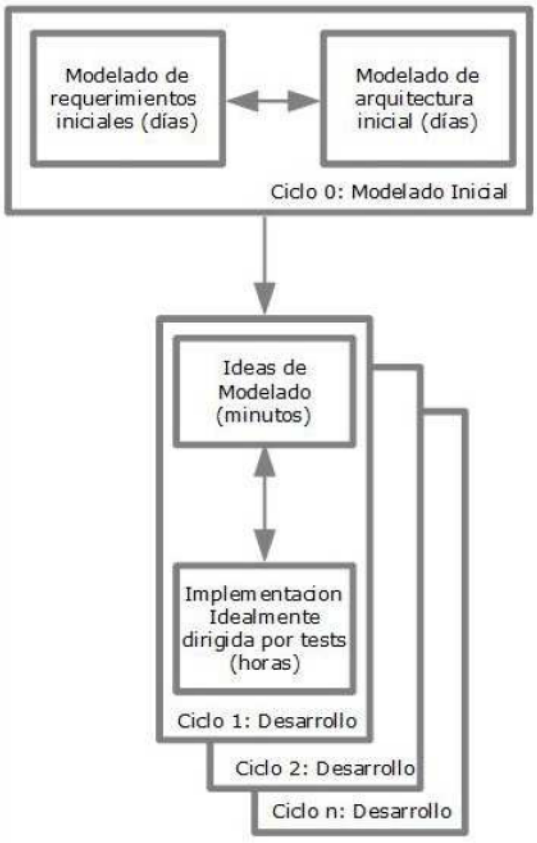


Figura 3.1: El ciclo de vida de Agile Model Driven Development (AMDD)

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.

### 3.3.3 TEST DE REGRESIÓN DE BASES DE DATOS

Para cambiar en forma segura software existente, ya sea aplicando un refactoring o una nueva funcionalidad, se tiene que poder verificar que no se haya modificado un comportamiento existente luego de realizar el cambio. En otras palabras, se necesita poder correr un conjunto completo tests de regresión en el sistema. Si se descubre que algo no funciona como antes, se debe 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) fomentan escribir primero los tests que el código. De esto nos 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. Como describe el siguiente diagrama de actividad UML, los pasos de TFD son los siguientes:

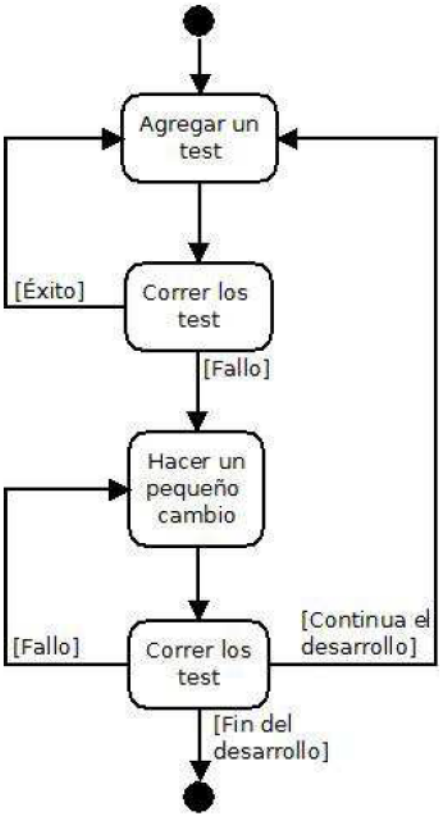


Figura 3.2: Un enfoque Test First Development

1. Agregar un test, básicamente el código necesario para que el test falle.
2. Correr los test, usualmente el conjunto completo de test, o un subconjunto de los mismos, sólo para asegurar que el nuevo test falla.
3. Actualizar el código para que el nuevo test se ejecute correctamente.
4. Correr nuevamente los tests. Si fallan, retornar al paso 3, caso contrario comenzar nuevamente.

Las ventajas principal 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. Del mismo modo que con un conjunto completo de tests para el código fuente de la aplicación nos permite refactoring de código, tener una regresión completa de test para la base de datos permite el refactoring de base de datos.

Test Driven Development (TDD) [BK 03] es la combinación de TFD y refactoring. 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. A medida que se aplica refactoring, se deben correr los test de regresión para verificar que el comportamiento existente antes del refactoring sigue funcionando correctamente.

Una implicación importante es que probablemente sean necesarias varias herramientas para tests de unidad, al menos una por base de datos y una para cada lenguaje de programación usado en las herramientas externas. La familia XUnit de

herramientas, por ejemplo JUnit para Java, VBUnit para Visual Basic, NUnit para .NET, OUnit para Oracle, son libres y consistentes entre ellas.

### 3.3.4 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 que se necesita volver atrás el cambio a un estado previo.

Para permitir el refactoring de la base de datos, se necesita poner los siguientes ítems bajo un control de manejo de la configuración.

- Scripts DDL para crear el esquema de la base de datos
- Scripts de carga, extracción y migración de datos
- Archivos del modelo de datos
- Meta data de mapeos objeto-relacional
- Definiciones de stored procedures y triggers
- Definiciones de vistas
- Restricciones de integridad referencial
- Objetos de la base de datos como secuencias, índices, etc.
- Datos de referencia
- Datos de test
- Scripts para generación de datos de tests
- Scripts para tests

### 3.3.5 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 3.3 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.

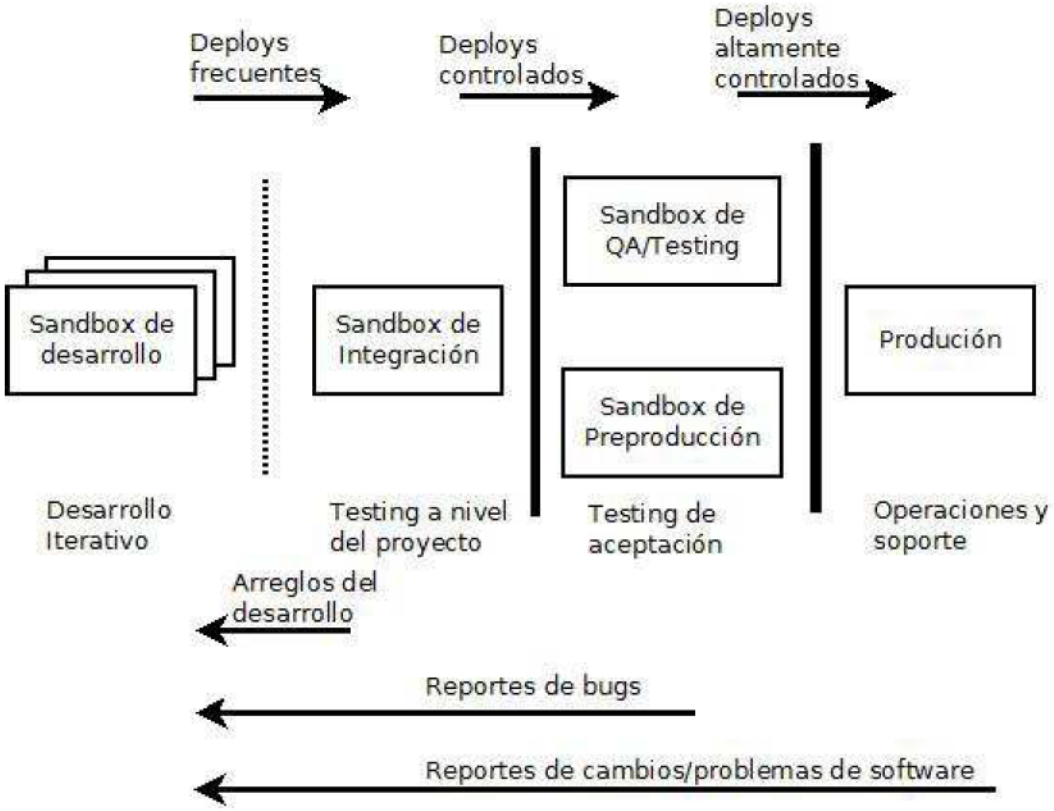


Figura 3.3: Organización lógica de varios *sandboxes*

Para hacer un refactoring exitosamente en un esquema base de datos, los desarrolladores necesitan sus propios *sandboxes* físicos para trabajar en ellos, una copia del código fuente, una copia de la base de datos. Teniendo sus propios ambientes, pueden hacer cambios de manera segura, hacer testing y aceptarlos o descartarlos. Cuando se está de acuerdo que el refactoring a la base de datos es viable, los cambios se promueven en los ambientes compartidos del proyecto, se hace testing y se ponen bajo un control de gestión de la configuración, de ese modo el resto del equipo los conoce. Eventualmente, el resto del equipo promueve su trabajo incluyendo todos los refactorings a la base de datos en los ambientes para demo o preproducción. Estas promociones usualmente ocurren una vez por cada ciclo de desarrollo, pero pueden ocurrir más o menos veces dependiendo del ambiente. Cuanto más frecuentes sean estas promociones, mayores serán las chances de recibir valioso feedback. Finalmente, luego que el sistema pase los tests de aceptación, se hará un deployment (despliegue) con los cambios en producción.

### 3.3.6 *Impedimentos a las técnicas evolutivas en el desarrollo de bases de datos*

El primer impedimento, y el más difícil de sobrellevar, es cultural. Muchos de los profesionales que trabajan hoy en bases de datos comenzaron sus carreras en los años 70 y tempranamente en los 80, cuando el enfoque "codificar y corregir" como metodología de desarrollo era común. La comunidad IT reconoció que este enfoque resultó en código de baja calidad, difícil de mantener, y adoptaron las técnicas de desarrollo estructuradas que aún se siguen usando. A raíz de esas experiencias, la

mayoría de los profesionales de bases de datos creen que las técnicas evolutivas introducidas con la revolución de las tecnologías orientadas a objetos de los años 90 fueron un rejunte de los enfoques "codificar y corregir" de los años 70. Y siendo justos, muchos desarrolladores software en objetos tomaron este camino. Eligieron igualar enfoques evolutivos con baja calidad, pero como la comunidad ágil ha demostrado, este no ha sido el caso. El resultado final es que la mayoría de la bibliografía orientada a datos pareciera enfocar a lo tradicional, procesos en serie del pasado y perdiendo principalmente enfoques ágiles. La comunidad de profesionales de bases de datos tiene mucho por alcanzar.

El segundo impedimento es la falta de herramientas, aunque la comunidad open source (al menos dentro de la tecnología Java) está rápidamente llenando espacios. Aunque mucho esfuerzo ha sido puesto en el desarrollo de herramientas de mapeo objeto-relacional, y algo dentro de de herramientas de testing de bases de datos, hay aún mucho trabajo por hacer. Tomó varios años a las herramientas tradicionales incorporar funcionalidad para refactoring. Y tomará varios años a las herramientas para trabajar con las bases de datos incorporar esta funcionalidad. Claramente, la necesidad de herramientas usables y flexibles que permitan desarrollo evolutivo del esquema de bases de datos existe. La comunidad open source está claramente comenzando a llenar estos espacios y los vendedores de herramientas comerciales seguirán el mismo camino.

# 4. REFACTORING DE BASES DE DATOS

## 4.1 REFACTORING DE BASES DE DATOS

Un refactoring de bases de datos [AM 03] es un simple cambio al esquema de la base de datos que mejora su diseño mientras mantiene su semántica de comportamiento y de datos. En otras palabras, no se puede agregar funcionalidad o cambiar comportamiento existente, ni agregar nuevos datos o cambiar el significado de datos existentes.

Como se mencionó previamente, un refactoring de bases de datos es conceptualmente más complicado que un refactoring de código: un refactoring de código sólo necesita mantener el comportamiento semántico, mientras que un refactoring de bases de datos debe también mantener la semántica de la información.

Otro aspecto crítico en los refactorings de bases de datos es que pueden volverse más complicados aún por el acoplamiento que puede existir en la arquitectura de la base de datos, como muestra la Figura 4.1. 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.

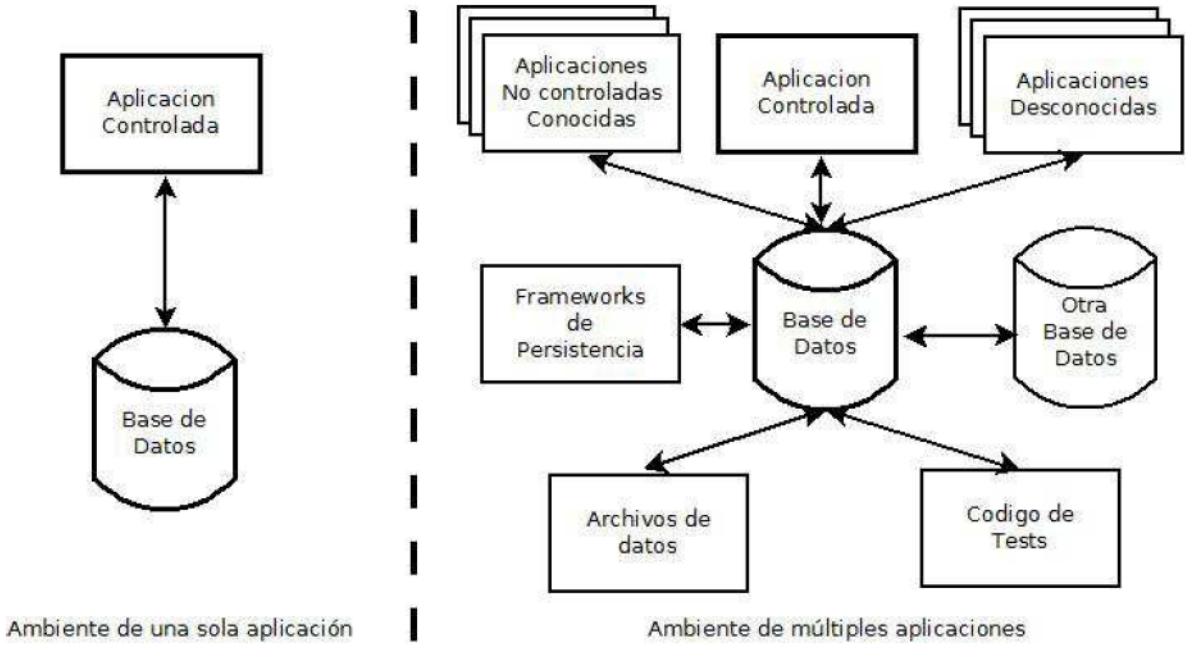


Figura 4.1: Las dos categorías de arquitectura de bases de datos

La arquitectura de una sola aplicación accediendo a la base de datos es la más sencilla. Una sola aplicación interactúa con la base de datos, permitiendo hacer refactoring en paralelo en la aplicación y en la base de datos, y haciendo el deployment

(despliegue) los mismos simultáneamente. 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 (también llamado periodo de deprecación / obsolencia) durante el cual el viejo esquema y el nuevo esquema serán soportados en paralelo.

### 4.1.1 Ambiente de una sola aplicación

Vamos a analizar el caso de cómo sería mover una columna de una tabla a otra en un ambiente de una sola aplicación accediendo a la base de datos. Esta es la situación más simple, porque se tiene completo control sobre el esquema de la base de datos 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 de base de datos al mismo tiempo. No se necesitará brindar soporte al esquema original y al nuevo de la base de datos en paralelo porque una sola aplicación accede a la base de datos.

En este escenario, se sugiere que dos personas trabajen juntas a la par, una persona podría tener habilidades de programación para la aplicación y otra para la base de datos, o idealmente ambos podrían tener ambas habilidades. El refactoring es desarrollado y testeado en los sandboxes de los desarrolladores. Cuando está terminado, los cambios son promovidos a los ambientes de integración del proyecto.

Por ejemplo, veamos el caso para aplicar un refactoring Mover Columna, explicado en detalle al final del presente capítulo. El objetivo de este refactoring, como su nombre indica, es mover una columna de una tabla a otra. Para proceder con el mismo, primero se corren todos los test y se comprueba que pasan correctamente. Luego, se puede escribir un test, aplicando un enfoque Test Driven Development (TDD). Después de correr este test y ver que falla, se introduce el cambio. Una vez realizado el cambio, se deben correr todos los test y comprobar que todo siga funcionando como antes.

Una vez que la aplicación está corriendo nuevamente, se debe realizar un backup de los datos en la columna a eliminar y copiar los datos de esta columna a la nueva columna. Luego se debe eliminar la columna. Se volverán a correr los tests para asegurar que la migración de datos se ha hecho correctamente. Cuando se termina de hacer todo esto, se promueven los cambios al ambiente de integración.

### 4.1.2 Ambientes de múltiples aplicaciones accediendo a la base de datos

Esta situación es más complicada porque las aplicaciones individuales tienen sus respectivos releases en diferentes ocasiones. Para implementar este refactoring en la base de datos, se hace el mismo trabajo que para el ambiente de una sola aplicación, excepto que no se borra la columna de la tabla. En su lugar, ambas columnas quedan

en paralelo durante un periodo de transición para dar tiempo a los equipos de desarrollo para 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.

Hay que tener en cuenta que el periodo de transición debe ser lo suficientemente largo para soportar aplicaciones que no estén en etapa de desarrollo o mantenimiento y que tal vez tengan uno o dos releases al año, o aplicaciones que trabajen con un ciclo de vida tradicional o en cascada, también con pocos releases al año.

Luego del periodo de transición, se remueve la columna original y los triggers, resultando en el esquema final. Se remueven estos objetos solo después del suficiente testing para comprobar que es seguro hacerlo.

### *4.1.3 Manteniendo la semántica*

Cuando se hace un refactoring de base de datos, se debe mantener la semántica de información y de comportamiento. No se debe agregar ni eliminar nada. La semántica de información se refiere al significado de la información dentro de la base de datos, desde el punto de vista de quién consume esa información. Preservar la semántica de la información implica que si se cambia el valor de un dato almacenado en una columna, los clientes de la información no se deberían ver afectados por el cambio. Por ejemplo, podríamos pensar en un refactoring para introducir un formato a una columna que almacena números telefónicos en formato de caracteres, para transformar datos del estilo (0221) 425-5555 y 0221-425-5555 al formato 2214255555. A pesar que el formato ha sido mejorado, requiriendo un simple código para trabajar con el dato, desde el punto de vista práctico la verdadera información no ha cambiado. Nótese que se puede elegir ahora el formato en que los números se mostrarán, solo que ahora no se guardará la información del formato en la base de datos.

Enfocarse en la práctica es un punto crítico en el refactoring de bases de datos. Martin Fowler habla de comportamiento observable cuando habla de refactoring de código, apuntando a que con muchos refactorings no se puede estar completamente seguro que no se ha cambiado la semántica de algún modo, lo único que podemos esperar es que se ha hecho lo mejor que se puede, escribiendo la cantidad que consideramos suficientes de test y que el resultado correcto de correr esos test signifique que la semántica sigue siendo la misma. Un efecto similar surge cuando se trata de preservar la semántica de información en un refactoring de bases de datos. Cambiar (0221) 425-5555 a 2214255555 puede de hecho cambiar la semántica de algún modo que no podamos percibir. Por ejemplo, tal vez exista un reporte que solo trabaje con los las filas que tenga el número telefónico en formato (0XXX) XXX-XXXX, y el reporte confía en este hecho. Ahora el reporte está devolviendo datos de la forma XXXXXXXXXX, haciéndolos difícil de leer, aunque la misma información está siendo devuelta. Cuando el problema es detectado, el reporte necesitará ser actualizado para reflejar el nuevo formato.

Similarmente, con respecto a la semántica de comportamiento, el objetivo es mantener la funcionalidad de caja negra. Cualquier código fuente que trabaje con los aspectos cambiados en la base de datos debe ser modificado para lograr la misma funcionalidad que tenía antes.

Es importante reconocer que un refactoring de base de datos es un subconjunto de transformaciones a la base de datos. Una transformación a la base de datos puede o no cambiar la semántica, un refactoring de base de datos no puede hacerlo. Como ejemplo de transformación podemos citar el hecho de agregar una columna a una tabla. Esta transformación es necesaria para el refactoring Mover Columna.

En principio, agregar una columna podría sonar como un refactoring a la base de datos. Agregar una columna a una tabla no cambia la semántica de una tabla hasta que nueva funcionalidad comience a usar esa columna. Nosotros lo seguiremos considerando una transformación, no un refactoring, porque inadvertidamente puede cambiar el comportamiento de la aplicación. Por ejemplo, introducir una columna en el medio de una tabla puede hacer que un programa que acceda a los a las columnas por su posición en lugar que por su nombre, deje de funcionar.

## 4.2 CATEGORÍAS DE REFACTORING DE BASE DE DATOS

Podemos distinguir seis categorías de refactoring de base de datos:

| **Categoría** | **Descripción** | **Ejemplos** |
|---|---|---|
| Estructurales | Un cambio en la definición de tablas o vistas | Mover una columna de una tabla a otra, separar una columna multipropósito en varias columnas, una para cada propósito |
| Calidad de datos | Un cambio que mejora la calidad de la información contenida en la base de datos | Hacer que una columna que no permita valores nulos, asegurar que los datos de una columna tienen datos en un formato consistente |
| Integridad referencial | Un cambio que asegura que una fila referenciada exista en otra tabla, asegurar que una fila que no es necesitada sea removida apropiadamente | Agregar un trigger que permita borrado en cascada entre dos entidades, código que estaba implementado formalmente fuera de la base de datos |
| Arquitecturales | Un cambio que mejora la manera en que programas externos acceden a la base de datos | Remplazar una operación Java existente en una librería compartida con un procedimiento almacenado en la base de datos. Teniendo esto como un procedimiento almacenado, se permiten aplicaciones no Java |
| Métodos | Un cambio en un método (un procedimiento almacenado, una función, | Renombrar un procedimiento almacenado para hacerlo más fácil de |

| | | |
|---|---|---|
| | un trigger) que mejora la calidad del mismo. Algunos refactorings de código aplicables a métodos de la base de datos | entender |
| Transformaciones a la base de datos (No es refactoring) | Un cambio al esquema de la base de datos que cambie su semántica | Agregar una columna a una tabla existente |

## 4.3 DATABASE SMELLS

Fowler [FO 99] introdujo el concepto de *code smell*, una categoría de problemas comunes en el código que indican que el mismo necesita refactoring. Algunos *code smells* pueden ser sentencias *switch*, métodos largos, código duplicado, sentencias *if* anidadas, etcétera. Similarmente definimos *database smells* comunes que indican que potencialmente se necesitaría un refactoring en la base de datos:

**Columna multipropósito:** Si una columna está siendo usada para varios propósitos, es probable que exista código para asegurar que el dato se está usando de la manera adecuada, usualmente verificando valores en otras columnas. Un ejemplo puede ser una columna usada para almacenar la fecha de nacimiento de una persona si es un cliente, o la fecha de ingreso si es un empleado. Peor aún, ¿cómo se registraría la fecha de nacimiento de un empleado?

**Tabla multipropósito:** Similarmente, cuando una tabla está siendo usada como para almacenar varios tipos de entidades, es probablemente una falla en el diseño. Un ejemplo es una tabla Cliente genérica para almacenar información sobre personas y empresas. El problema con este enfoque es que las estructuras de datos para personas y empresas difieren. Un cliente tendrá nombre y apellido, mientras que una empresa tiene razón social. Una tabla Cliente genérica tendrá valores nulos en algunos campos dependiendo del tipo de entidad en la fila.

**Datos redundantes:** Los datos redundantes son un serio problema en las bases de datos porque cuando un dato es guardado en distintos lugares ocurre la oportunidad para inconsistencias. Por ejemplo, una organización grande puede tener información sobre sus clientes almacenada en diferentes lugares. Si un cliente modifica alguno de sus datos (su dirección por ejemplo), probablemente un sistema no actualice sus datos, debido a que el cliente informará una vez su cambio de domicilio. A priori, no se puede saber cuál es la dirección real del cliente.

**Tablas con muchas columnas:** Cuando una tabla contiene muchas columnas, es un indicativo que la tabla carece de cohesión, que está tratando de almacenar varias entidades. Por ejemplo una tabla de clientes que tenga varias columnas para almacenar tres tipos de direcciones (envío, facturación, estacional) o varios números telefónicos. Esta tabla necesitará ser normalizada agregando una tabla de teléfonos y otra con direcciones.

***Smart column*:** Una columna inteligente (*Smart Column*), es una columna en la cual diferentes posiciones dentro del dato representan diferentes conceptos. Por ejemplo, si los primeros cuatro dígitos del identificador de cliente representan el

identificador de la sucursal que despachará la mercadería solicitada, el identificador del cliente es una *smart column* porque se debe analizar (*parse*) el identificador para descubrir información más granular. Otro ejemplo podría ser usar una columna de texto para almacenar XML, claramente se puede aplicar parsing en el XML para obtener información más granular. Las *smart columns* necesitan ser reorganizadas en sus datos constituyentes para que se pueda tratar fácilmente con ellos como información atómica.

**Miedo al cambio:** Si existe un temor por cambiar algo en el esquema de la base de datos es porque existe temor a que algo deje de funcionar correctamente, como alguna de las aplicaciones que acceden a la misma. Este es la señal más segura que la base de datos necesita refactoring. El miedo al cambio es un buen indicio que se tiene un serio riesgo técnico en las manos, y que seguirá aumentando con el tiempo.

Es importante destacar que porque se detecte un *database smell* no significa que algo esté mal. Cuando se encuentre alguna de estas situaciones se les debe prestar atención, analizarlas y aplicar un refactoring si tiene sentido.

Otro punto a considerar es la necesidad de documentación, que en ciertos casos refleja la necesidad de refactoring. Cuando se encuentra que se necesita escribir documentación de soporte para describir una tabla, columna o procedimiento almacenado, es una buena indicación que se necesita aplicar un refactoring a la parte del esquema para que sea más fácil de entender. Tal vez un simple renombrado sea pueda evitar varios párrafos de documentación. Cuanto más limpio sea un diseño, menor será la documentación que necesitará.

## 4.4 Facilitando el refactoring de bases de datos

Cuanto mayor sea el acoplamiento entre dos módulos, más difícil será aplicar un refactoring. Esto es verdadero para el refactoring de código y más aún para el refactoring de bases de datos. El acoplamiento se convierte en un serio problema cuando se empiezan a considerar aspectos de comportamiento (por ejemplo código). Cómo se menciono anteriormente, el escenario más fácil y menos común es el de una aplicación accediendo a la base de datos, porque la base de datos sólo estará acoplada con ella misma y con la única aplicación externa. Con una arquitectura de múltiples aplicaciones (Figura 4.2), la base de datos estará posiblemente acoplada con código fuente de las aplicaciones, frameworks de persistencia de mapeo objeto-relacional (ORM), otras bases de datos (vía replicación por ejemplo), archivos de esquema de datos, código de testing e incluso con ella misma.

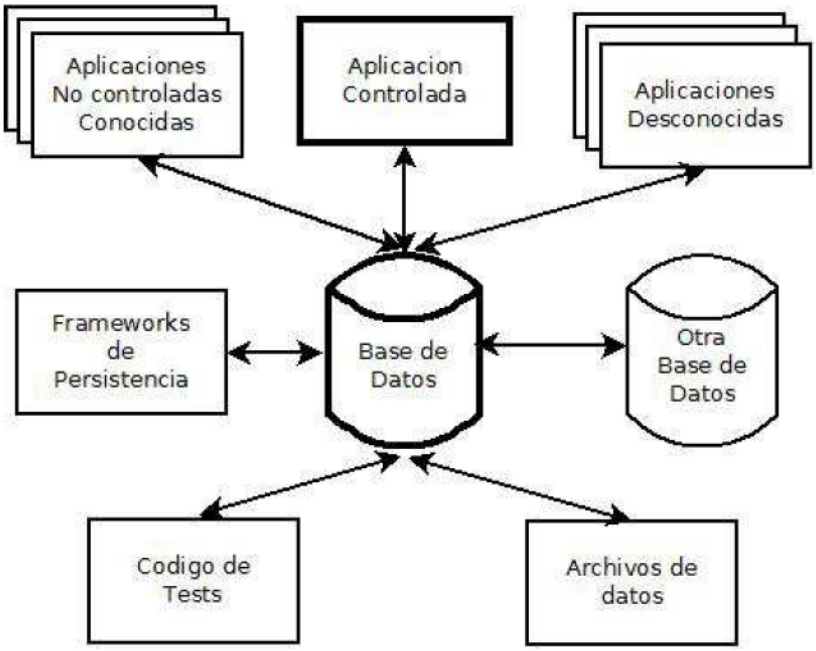


Figura 4.2: Los refactorings de bases de datos están altamente acoplados a los programas externos

Una forma efectiva de decrementar el acoplamiento es encapsulando el acceso a la base de datos. Esto se logra teniendo programas externos que acceden a la base de datos mediante capas de persistencia, como se muestra en Figura 4.3. Una capa de persistencia puede implementarse de varias maneras: mediante objetos de acceso a datos (DAOs, Data Access Objects), que implementan el código SQL necesario, mediante frameworks, procedimientos almacenados e incluso mediante web services. Como se muestra en la figura nunca se puede reducir totalmente el acoplamiento, pero se puede llevar a niveles manejables.

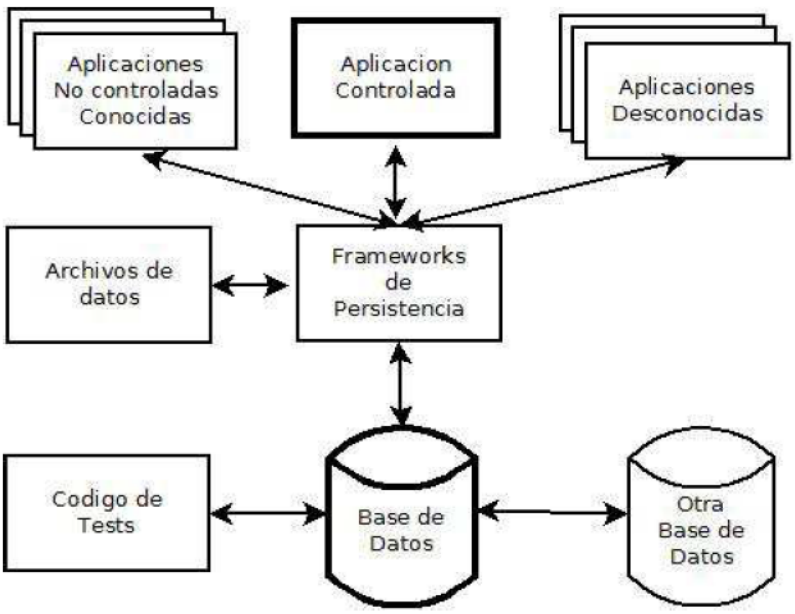


Figura 4.3: Reducción del acoplamiento encapsulando el acceso.

## 4.5 EL PROCESO DE REFACTORING DE BASES DE DATOS

La Figura 4.3 muestra un diagrama de actividad UML con el proceso completo para aplicar un refactoring a la base de datos. El proceso comienza con un desarrollador que está intentando implementar un nuevo requerimiento o corregir un defecto. El desarrollador se da cuenta que el esquema de la base de datos puede necesitar un refactoring. Cuando el desarrollador posiblemente con la opinión de otro miembro del equipo decide aplicar el refactoring, iterativamente trabajan acorde a las actividades:

- Verificar que el refactoring a la base de datos es el adecuado
- Elegir el refactoring más apropiado
- Deprecar el esquema de la base de datos original
- Hacer testing antes, durante y después
- Modificar el esquema de la base de datos
- Migrar los datos
- Modificar el acceso del o los programas externos
- Correr test de regresión
- Controles de versión sobre el trabajo
- Anunciar el refactoring

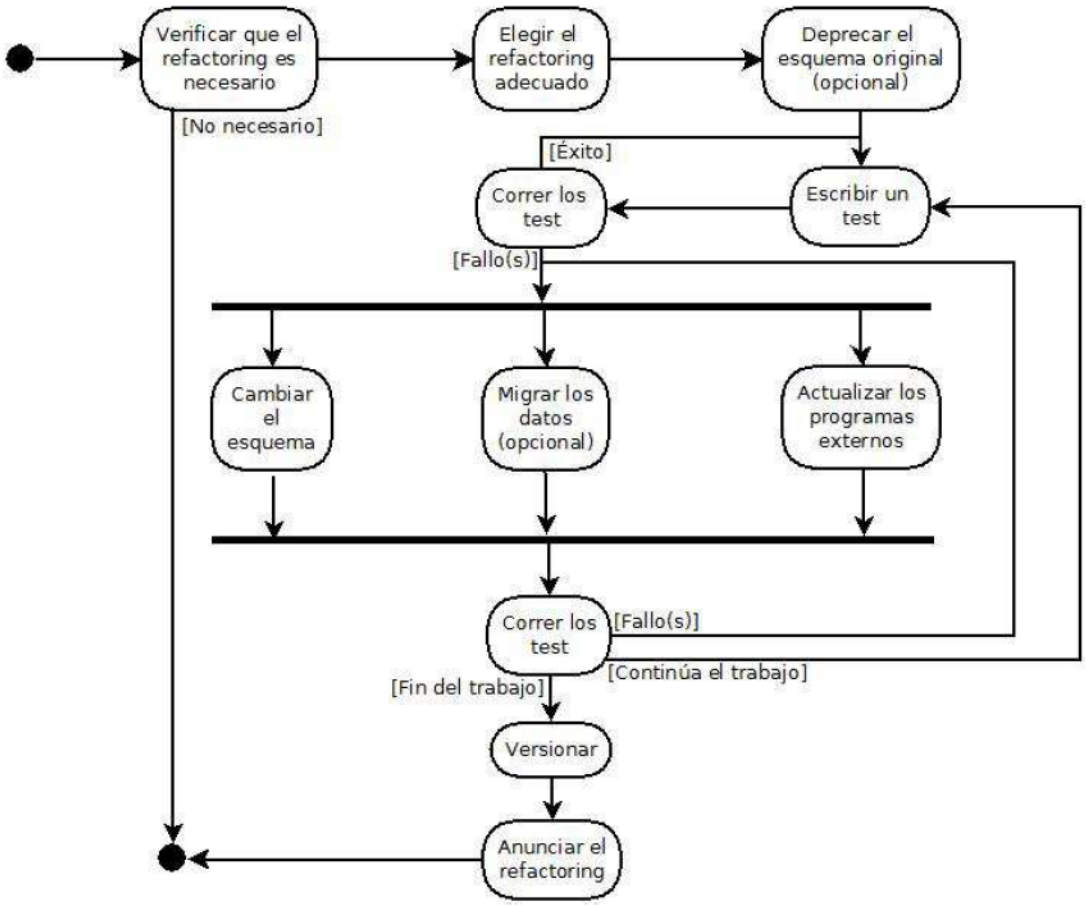


Figura 4.4: El proceso de refactoring de bases de datos

### 4.5.1 VERIFICAR QUE EL REFACTORING A LA BASE DE DATOS ES EL ADECUADO

Hay tres aspectos a considerar:

**¿Tiene sentido el refactoring?**

Tal vez la estructura actual de la tabla es incorrecta. Es común que los desarrolladores estén en desacuerdo o simplemente que no comprendan del todo un diseño existente, ya sea en código fuente o en la base de datos. Esta falta de entendimiento puede llevarlos a creer que ese diseño necesita un cambio cuando en verdad no. El administrador de la base de datos (DBA, database administrator) deberá tener un buen conocimiento de la base de datos del proyecto, otras bases de datos de la organización y sabrá a quién contactar por cuestiones como esta. Luego, se estará en una mejor posición para saber si el esquema existente tiene el diseño correcto. Además el DBA suele tener una visión general de toda la organización, por lo que puede proveer información que no es aparente desde el punto de vista de un proyecto individual.

**¿Es el cambio realmente necesario ahora?**

En este punto es bueno hacerse varias preguntas: ¿Se tiene una buena razón para hacer el cambio en el esquema? ¿Se tiene en claro el requerimiento del negocio que obliga a hacer el cambio? ¿Se tiene en claro la mejora que proporcionará el refactoring al diseño? En base a lo que se haya respondido el equipo y a la incertidumbre que haya en las respuestas, el equipo tomará la decisión de hacer el cambio o dejarlo para más adelante.

**¿Vale la pena el esfuerzo?**

El siguiente paso será evaluar el impacto total del refactoring. Se debe tener un conocimiento del acoplamiento de los programas externos a la parte de la base de datos a la que se aplicará el refactoring. Este conocimiento el equipo lo tiene de haber trabajado con los arquitectos de las aplicaciones, los administradores de las bases de datos operacionales, los desarrolladores de las aplicaciones externas y otros DBAs. Cuando el equipo no está seguro del impacto, es aconsejable informarse con los equipos que trabajan en las aplicaciones externas. El objetivo es asegurar que la implementación del refactoring será un éxito. Si se tendrán que actualizar, testear y hacer el deployment (despliegue) de 50 aplicaciones externas para soportar el refactoring, puede ser que no sea viable continuar. Incluso cuando hay una sola aplicación accediendo a la base de datos, puede estar altamente acoplada a esa porción de la base de datos que se quiere cambiar, haciendo que el esfuerzo no valga la pena. Sin embargo, si el problema de diseño es claramente severo, seguramente un refactoring agilizará desarrollos futuros, de modo que se decidirá la aplicación del refactoring por más que muchos sistemas se vean afectados.

### 4.5.2 Elegir el refactoring de base de datos apropiado

Para determinar cuál es el refactoring más apropiado para cada situación se debe primero analizar y entender el problema al que se está enfrentando. En este punto es crucial la interacción con el equipo. Supongamos que un desarrollador descubre que para un requerimiento se tiene que agregar una columna a una tabla, entonces se sugiere la transformación adecuada, es decir, agregar columna. Sin embargo, no había notado que esa columna con la información que necesitaba ya estaba en la base de datos, en otra tabla, debido a un pobre diseño existente. En este caso, el desarrollador encontró un problema existente (la falta de la columna) pero la solución no fue la más adecuada. Teniendo el resto del equipo un conocimiento sobre el esquema de la base de datos, se procederá con la solución adecuada, que es la de aplicar el refactoring Mover Columna.

### 4.5.3 Deprecar el esquema original de la base de datos

Si múltiples aplicaciones acceden a la base de datos, probablemente se tenga que trabajar bajo la asunción que no se podrá hacer el refactoring y deployment de todos los programas simultáneamente. Se necesitará un período de transición, también llamado periodo de deprecación u obsolencia, para la parte del esquema original que se está cambiando. Durante el periodo de transición se soportarán ambos esquemas, el original y el nuevo en paralelo para que los equipos de las otras aplicaciones puedan aplicar el refactoring y hacer el deployment sus aplicaciones. Los periodos típicos pueden ser de varios trimestres, o hasta años. El tiempo potencial que pueda llegar a tomar la aplicación completa de un refactoring remarca la necesidad de automatizar el proceso tanto como sea posible. Considerando un periodo de un año, la gente en un departamento cambiará, generando un riesgo en los procesos manuales. Habiendo dicho esto, incluso en el caso de una única aplicación accediendo a la base de datos, el equipo aún necesitará un periodo de transición de unos días dentro del sandbox de integración del proyecto.

Consideremos el ciclo de vida de un refactoring de base de datos dentro de un escenario de múltiples aplicaciones. Primero se implementa dentro del alcance de un proyecto, y si es exitoso eventualmente se hará el deployment en producción. Durante el proceso de transición, ambos esquemas existirán, con suficiente código para asegurar que cualquier actualización estará soportada. En este periodo se debe asumir que algunas aplicaciones usarán el esquema original mientras otras usen el nuevo esquema y que ninguna aplicación usará sólo una de las versiones del esquema y nunca las los al mismo tiempo. Más allá de la versión que use cada aplicación, todas las aplicaciones deberán correr apropiadamente. Cuando el periodo de transición haya finalizado, el esquema original y todo el código generado para el soporte de ambas versiones será removido, y la base de de datos deberá ser testeada. En este punto, la asunción será que todas las aplicaciones trabajen con la nueva versión del esquema.

Si tomamos como ejemplo el refactoring Mover Columna, tendremos durante el periodo de transición la columna en la tabla original y la nueva columna en la tabla a la que se deberá mover. En este caso, como código de soporte, necesitaremos dos triggers

que mantengan sincronizadas ambas columnas, uno para cada tabla, que se ejecute luego de cualquier sentencia *insert*, *delete*, o *update*.

Cabe destacar que no todo refactoring a la base de datos requiere un periodo de transición. Por ejemplo, Agregar Constraint o Aplicar Códigos Estándar no requieren un periodo de transición porque simplemente mejoran la calidad de los datos reduciendo la cantidad de datos aceptada en una columna. Sin embargo, menor cantidad de datos aceptados pueden hacer que algunas aplicaciones dejen de funcionar como antes, por lo hay que ser muy cuidadoso en estos casos.

### 4.5.4 TESTING ANTES, DURANTE Y DESPUÉS

Se podrá tener la seguridad que cambiar el esquema de la base de datos si se puede fácilmente validar que la base de datos aún funciona con las aplicaciones antes del cambio, y una buena forma de hacer esto es aplicando la metodología de TDD, Test Driven Development. Con un enfoque TDD se escribe un test y luego la cantidad suficiente de código, usualmente DDL Data Definition Language, para cumplir con el test. Seguramente se necesiten test para:

- Testear el esquema de la base de datos
- Testear la forma que las aplicaciones usen la base de datos
- Validar la migración de datos
- Testear los programas externos

**Testing del esquema de la base de datos**

Debido a que un refactoring a la base de datos afectará el esquema, se necesita escribir test orientados a la base de datos. Aunque esto puede sonar extraño, se pueden validar varios aspectos de la base de datos:

- Procedimientos almacenados y triggers. Los procedimientos almacenados y los triggers deberían ser testeados al igual que cualquier código de la aplicación.
- Integridad referencial. Las reglas de integridad referencial, en particular borrados en cascada en los que filas hijas son borradas cuando su padre es borrado, deben ser validadas. Reglas de existencia o cuantificación entre entidades, pueden ser fácilmente testeadas.
- Definiciones de vistas. Las definiciones de vistas usualmente implementan lógica de negocio. Algunas cosas que se pueden testear son: ¿Funciona correctamente el filtrado? ¿Se devuelven la cantidad correcta de filas? ¿Se devuelven las columnas correctas? ¿Están las filas y las columnas en el orden correcto?
- Valores por defecto. Accidentalmente se pude remover el valor por defecto en una definición de tabla. Se debe verificar que todos los valores por defecto estén siendo correctamente asignados.
- Invariantes de datos. Las columnas suelen tener invariantes, usualmente implementados como constraints. Por ejemplo, una columna numérica puede tener valores del 1 al 7. Este tipo de invariantes debe ser testeado.

El testing para las bases de datos es una técnica reciente, por lo que se deben afrontar varios desafíos:

- Insuficientes habilidades para testing. Este problema se puede sobrellevar mediante capacitación, aplicando pair programming, por ejemplo entre un desarrollador de base de datos y un tester.
- Insuficientes tests de unidad en la base de datos. Por tratarse de una práctica reciente, es probable que en una base de datos no se tengan los suficientes test para el esquema existente. Por más que esto sea desafortunado, no existe mejor momento que el presente para comenzar a escribir tests.
- Insuficientes herramientas para testing de bases de datos. Afortunadamente herramientas como DBUnit para la gestión de datos para tests y SQLUnit para testear stored procedures están disponibles como herramientas Open Source, además de herramientas comerciales.

Volviendo al ejemplo de la aplicación del refactoring Mover Columna, se tienen dos cambios en el esquema durante el periodo de transición que se deben validar. El primero es la creación de la nueva columna. Este cambio puede ser cubierto por las pruebas sobre la migración de datos y por los tests de los programas externos. El segundo cambio a testear son los dos triggers que mantienen las columnas sincronizadas. Se deberían agregar tests que aseguren que si una columna es actualizada en una tabla, el cambio también se refleje en la otra tabla.

### Validación de la migración de datos

Muchos refactorings de bases de datos requieren migración y en ocasiones limpieza de los datos. En el caso del refactoring Mover Columna, se requerirá copiar los datos de la columna original a la nueva columna. En este caso, se debería verificar que cada uno de los datos fue copiado correctamente.

En refactorings como Aplicación de Códigos Estándar y Consolidación de Estrategias de Clave, se "limpian" los datos. Esta lógica debe ser validada. Con el primer refactoring, se pueden convertir códigos del tipo U.S. o USA al valor estándar US. Se debe tener un test que los códigos anteriores no sigan siendo usados y que hayan sido correctamente convertidos al nuevo valor. Para el segundo refactoring, podemos dar un ejemplo que en algunas tablas los clientes son identificados con su ID, mientras que en otras son identificados mediante su CUIT. Se debe elegir una forma para identificar los clientes (por su ID) y luego aplicar refactoring en todas las tablas que identifiquen clientes mediante otra clave para que lo hagan mediante su ID. En este caso, sería interesante escribir tests que verifiquen que las relaciones entre varias filas sigan siendo mantenidas.

### Testing de los programas externos

Los programas externos deben tener un conjunto suficiente de tests como cualquier otro artefacto IT en la organización. Para el refactoring de la base de datos quede exitosamente aplicado, se debe introducir el esquema definitivo luego de período de transición y verificar que todos los programas externos sigan funcionado correctamente. La única forma en la que se puede tener confianza para aplicar un refactoring a la base

de datos es teniendo un conjunto completo de tests de regresión para todas estas aplicaciones. Es probable que no se tengan los tests necesarios para estas aplicaciones, pero como se dijo antes, no existe mejor momento que el presente para comenzar a desarrollarlos. Es aconsejable que haya un test para cada refactoring a la base de datos.

### 4.5.5 MODIFICAR LA BASE DE DATOS

Ya sea mediante la escritura de código o usando una herramienta que genere el mismo, como la propuesta en este trabajo, se debe desarrollar el código que haga el refactoring a la base.

Un punto importante a tener en cuenta es la de asignar un código que identifique al archivo con el script que implementa cada refactoring. Por ejemplo se pueden asignar números correlativos para identificar cada script con un refactoring. Esta estrategia se puede mejorar anteponiendo el código de Sprint o Iteración en la que se encuentra el proyecto, y para los casos en los que haya varios equipos trabajando, también se le puede agregar el identificador del equipo.

Es importante trabajar con pequeños scripts para refactorings individuales por varias razones:

- Simplicidad. Pequeños y enfocados scripts de cambios son más fáciles de mantener y entender que scripts comprendiendo varios pasos. Si se descubre que un refactoring no se podrá aplicar debido a un imprevisto (por ejemplo, no se podrán actualizar la mayoría de la aplicaciones que accedan a la porción del esquema que se aplicará el refactoring), se podrá fácilmente no aplicar el refactoring.
- Exactitud. Se querrá aplicar cada refactoring en el orden apropiado de forma que el esquema evolucione de una forma definida. Los refactorings se pueden construir uno sobre el otro. Por ejemplo se puede renombrar una columna y una semana después moverla a otra tabla. El segundo refactoring dependerá del primero porque el código referenciará al nuevo nombre de la columna.
- Versionado. Diferentes instancias de la base de datos pueden tener diferentes versiones del esquema de la base de datos. Por ejemplo el sandbox de un desarrollador puede estar en la versión 163, el sandbox de integración del proyecto puede estar en la versión 161, el sandbox de QA/Testing puede estar en la versión 155 y el ambiente de producción puede estar en la versión 131. Para migrar el esquema del sandbox de integración a la versión 163 simplemente se deben aplicar los refactoring 162 y 163. Para realizar un seguimiento de los números de versión, se debe introducir una tabla, por ejemplo DatabaseConfiguration, en la que se guarde la versión actual del esquema.

Finalmente, también se debe entregar un script con las sentencias DLL que se ejecutará en cuando finalice el periodo de transición. Para el caso del refactoring Mover Columna, será una sentencia que elimine la columna original, y otras dos para remover los dos triggers.

Un aspecto importante al momento de implementar un refactoring es asegurar que la parte que se está cambiando siga las convenciones de la organización para las bases de datos. Estas convenciones deben ser administradas por el grupo de DBAs.

### 4.5.6 Migración de datos

Muchos refactorings requieren la manipulación de datos en alguna forma. Algunas veces, se necesita mover datos de una locación a otra, o se necesita limpiar los datos de alguna forma.

Similarmente que para la modificación del esquema, se necesitará crear un script para realizar la migración. Este script deberá tener el mismo identificador que el script que realice el cambio en el esquema para facilitar su manejo.

Dependiendo de la calidad de los datos, posiblemente se descubra que además se necesite limpiar los mismos. Esto requerirá aplicar alguno de los refactorings para la calidad de datos. Es una buena práctica no concentrarse en la calidad de los datos cuando se está trabajando en un refactoring estructural. Los problemas en la calidad de datos son comunes en bases de datos bien diseñadas en las que se han permitido degradaciones a lo largo del tiempo.

### 4.5.7 Modificar los programas externos que acceden a la base de datos

Cuando varios programas acceden a la base de datos, se corre el riesgo que alguno de ellos no sea actualizado por los equipos responsables, o peor aún, tal vez no están asignados a ningún equipo en el momento presente. La implicación directa es que alguien tendrá que tomar la responsabilidad de actualizar la aplicación. En muchos casos, los desafíos políticos para actualizar ciertos sistemas serán superiores a los desafíos técnicos para hacerlos.

Idealmente, la organización trabajará en todas las aplicaciones, haciéndolas evolucionar con el tiempo y haciendo los respectivos deployments (o despliegues) en intervalos regulares. Aunque esto pueda ser complicado por varios motivos, el departamento IT de la organización debería esforzarse por asegurar que los sistemas en la organización responden a los cambios necesarios. En estos ambientes se pueden tener periodos de transición relativamente cortos porque se sabe que todas las aplicaciones que acceden a la base de datos evolucionan en intervalos regulares y por lo tanto pueden ser actualizadas para trabajar con la nueva versión del esquema.

Entonces, ¿qué se puede hacer cuando se sabe que alguna de las aplicaciones externas no se actualizará, más allá del motivo? Un camino a seguir es no hacer el refactoring. El otro camino es hacer el refactoring y asignar un periodo de transición largo, de varios años o décadas. De esta forma, las aplicaciones que no se cambiarán podrán seguir accediendo a la base de datos, mientras que el resto de las aplicaciones acceden a un esquema con un diseño mejorado. Esta estrategia tiene la desventaja de que el código de soporte para mantener ambas versiones del esquema existirá por un largo periodo de tiempo, reduciendo el desempeño de la base de datos.

### 4.5.8 Correr los test de regresión

Parte de la implementación del refactoring es testear para asegurar su correcto funcionamiento. Como se indicó antes, se testeará, se hará un cambio, se testeará y así sucesivamente hasta que el refactoring esté completo. Las tareas de testing deberían estar automatizadas tanto como sea posible. Una ventaja significativa del refactoring de bases de datos es que los refactorings al representar pequeños cambios, cuando un test falla, se tiene una buena idea de donde el problema yace, justamente en ese lugar donde se aplicó el cambio.

### 4.5.9 Controles de versión sobre el trabajo

Cuando un refactoring a la base de datos es exitoso, se debe poner todo el trabajo bajo un control de manejo de la configuración, usando una herramienta apropiada para control de versiones. Si se tratan los artefactos de la base de datos de la misma forma que los artefactos del código fuente, es suficiente. Los artefactos a incluir bajo un control de versión son:

- Todos los scripts creados
- Datos para tests
- Casos para test
- Documentación
- Modelos

### 4.5.10 Anunciar el refactoring

Una base de datos es un recurso compartido. Mínimamente, es compartida por el equipo de desarrollo de la aplicación, si no es el caso de múltiples aplicaciones accediendo a la misma. Por lo tanto, se necesita comunicar a las partes interesadas que un refactoring a la base de datos se ha realizado. Tempranamente en el ciclo de vida del refactoring, se necesitará comunicar los cambios dentro del equipo de desarrollo, algo que puede ser como un simple anuncio al equipo en las reuniones stand up. En los ambientes de múltiples aplicaciones, se deberán comunicar los cambios a otros equipos, particularmente cuando se decida promover el refactoring a los ambientes de tests y preproducción. La comunicación puede ser desde un simple email en una lista que la organización use para anunciar los cambios en la base de datos, o puede ser un tema en los reportes o reuniones regulares de estado del proyecto o podría ser un reporte específico.

Un aspecto importante del anuncio será la actualización de toda la documentación relevante. Esta documentación será crítica porque el resto de los equipos necesitan saber cómo la base de datos ha evolucionado. Un enfoque simple puede ser entregar notas del release que sumarice los cambios que se han hecho, listando los refactorings en el orden que se han hecho.

También se deberá actualizar el modelo físico de la base de datos. El modelo físico es el modelo primario que describe el esquema de la base de datos, y usualmente uno

de los pocos modelos mantenidos desde el inicio del proyecto, por eso es necesario que esté al día.

## 4.6 Estrategias de Refactoring de Bases de Datos.

En esta sección del capítulo describimos un conjunto de estrategias a considerar a la hora de llevar a cabo un refactoring de Bases de Datos:

**Pequeños cambios son más fáciles de aplicar**

Es más fácil proceder en pequeños pasos, uno a la vez. Cuanto más grande es el cambio, mayores son las posibilidades de introducir un defecto, y mayor es la dificultad en encontrar dicho defecto.

**Identificar de forma única cada Refactoring**

Durante un proyecto de desarrollo de software, es común tener que aplicar cientos de refactorings y/o transformaciones a la base de datos. Dado que estos refactorings pueden ser dependientes unos de otros por ejemplo, es necesario asegurar que los refactorings se ejecuten en el orden correcto. Para esto se debe poder identificar unívocamente cada refactoring y las dependencias entre ellos. Existen básicamente tres estrategias para lograr esto:

- **Build Number:** El número de build de la aplicación, típicamente un número entero asignado por la herramienta de build. De esta forma la versión de la base de datos puede asociarse directamente con la versión de la aplicación. Esta estrategia puede presentar problemas en escenarios de muchas aplicaciones sobre una única base de datos. Por otra parte pueden existir entregas de aplicación que no incluyan refactorings de la base de datos, esto implica que no se puede garantizar secuencialidad en la generación de IDs para los refactorings.
- **Date/timestamp:** Se usa la fecha y hora actual como clave única del refactoring. Es una estrategia simple, tiene como contrapartida que es necesaria una estrategia extra para asociar el refactoring con el apropiado build de aplicación. Por otra parte puede ser engorroso incluir la fecha y hora en los nombres de los archivos de scripts.
- **Unique Identifier:** Un identificador único tal como GUID, o un valor incremental. La desventaja de usar GUID es que se necesitaría otra estrategia para determinar el orden de los refactorings. Además como en la estrategia anterior, necesitamos otro mecanismo para asociar el refactoring con el build de aplicación.

### Implementar un gran cambio como una composición de otros cambios más pequeños

Cuando tenemos que realizar un cambio importante sobre la base de datos, conviene analizar el cambio como un conjunto de refactorings más pequeños y aplicarlos en el orden correcto. Esto responde al viejo adagio, "¿Como comer a un elefante? Un mordisco a la vez".

Consideremos por ejemplo partir una tabla existente en dos. Para esto se tiene un único refactoring llamado *Split Table*, la realidad es que en la práctica se necesitan aplicar varios refactorings para lograr esto. Por ejemplo, necesitamos aplicar la transformación *Introduce New Table* para agregar la tabla nueva, el refactoring *Move Column* tantas veces como columna tenga la tabla, y potencialmente el refactoring *Introduce Index* para definir la primary key de la nueva tabla. A su vez para aplicar el *Move Column*, se deben aplicar las transformaciones Introduce *New Column* y *Move Data.*

### Usar una tabla de configuración para la base de datos

Es necesario poder identificar la versión actual del esquema para permitir las actualizaciones apropiadas sobre el esquema. La versión del esquema debe reflejar la estrategia de refactoring de la base de datos; por ejemplo si usamos la estrategia *Date/Timestamp* para identificar los refactorings, entonces también deberíamos usar *Date/Timestamp* para identificar la versión actual del esquema.

La manera más fácil de implementar esto es teniendo una tabla que mantenga esta información. Por ejemplo, con una estrategia *Build Number* podríamos definir esta tabla:

```
CREATE TABLE DatabaseConfiguration (SchemaVersion NUMBER NOT NULL);
INSERT INTO DatabaseConfiguration (SchemaVersion) VALUES (0);
```

La estrategia consiste en actualizar la tabla con una nueva versión cada vez que un refactoring es aplicado. Por ejemplo, si se aplica el refactoring número 17 sobre el esquema, DatabaseConfiguration.SchemaVersion debe ser actualizado a 17.

### Preferir triggers en lugar de Vistas o Sincronización Batch

Hemos visto que cuando varias aplicaciones acceden a la base de datos, frecuentemente es necesario un período de transición durante el cual, ambas versiones del esquema, la original y la nueva existen en producción. En consecuencia, es necesario asegurar que sin importar a que versión del esquema accede la aplicación, esta accede a datos consistentes. A continuación mostramos una tabla comparativa de las distintas estrategias de sincronización:

| Estrategia | Ventajas | Desventajas |
|---|---|---|
| Triggers. Uno o más triggers son implementados para actualizar la otra versión del esquema. | Actualización en tiempo real. | - Potencialmente pueden producirse ciclos.<br>- Potencialmente puede degradar la performance.<br>- Se duplican datos. Estos son almacenados en ambas versiones del esquema. |
| Vistas. Se introducen vistas que representan la tabla original, las cuales actualizan ambas versiones del esquema. | Actualización en tiempo real. | - Algunas bases de datos no soportan vistas actualizables.<br>- Puede ser complejo el agregado y la eventual eliminación de la vista. |
| Batch updates. Se ejecuta regularmente un procesamiento en lote. | Los impactos de performance por la sincronización de datos son absorbidos en momentos de bajo uso de la base. | - Alta probabilidad de problemas de integridad referencial.<br>- Suele ser dificultoso determinar el conjunto de actualizaciones de datos a realizar.<br>- Se duplican datos. Estos son almacenados en ambas versiones del esquema. |

### Elegir un Período de Transición suficiente

El DBA debe asignar un período de transición realista para los refactorings, de modo que sea suficiente para todos los otros equipos de aplicación. Una estrategia podría ser establecer un período común de transición para los distintas categorías de refactoring y luego aplicar dicho período de forma consistente. Por ejemplo, los refactorings estructurales pueden tener dos años de período de transición, y los refactorings de arquitectura pueden tener un período de tres años. La primera desventaja de esta aproximación es que se pueden adoptar periodos de transición muy largos, aún cuando la base de datos es accedida por aplicaciones que son desplegadas frecuentemente. Esto puede mitigarse negociando períodos de transición más cortos con el resto de los equipos.

### Simplificar las negociaciones con otros equipos

Una estrategia alternativa para definir el período de transición podría ser por cada refactoring individual, negociando con los responsables de otros equipos que se vean afectados por el refactoring. La ventaja de esta estrategia es que ayuda a una mejor comunicación de los cambios y produce períodos de transición más precisos. Como desventaja, las negociaciones pueden ser lentas y arduas.

### Encapsular el Acceso a la Base de Datos

Cuanto más encapsulado esté el código de acceso a la base de datos, más fácil será implementar el refactoring. Mínimamente, aún si la aplicación tiene código SQL *hard-coded*, este puede estar encapsulado de manera que pueda ser localizado y actualizado

rápidamente. Por ejemplo, podríamos manejar objetos de acceso a datos (DAOs), es decir, clases separadas a las clases de negocios que implementan la lógica de acceso a datos. De este modo, para las clases de negocios *Customer* y *Account* tendríamos las clases *CustomerDAO* y *AccountDAO* respectivamente. Mejor aún, si podemos evitar completamente código SQL generando la lógica de acceso a datos a partir de meta data definida mediante mapeos.

**Tener la posibilidad de crear fácilmente un Entorno de Base de Datos**

Durante todo el proceso de desarrollo hasta la puesta en producción, es necesario que los equipos tengan la posibilidad de crear instancias de la base de datos, frecuentemente con distintas versiones de esquema. La forma más eficaz de hacer esto es con un script de instalación que aplique las sentencias DDL iniciales para crear el esquema de base de datos y luego los scripts de cambios deseados.

## 4.7 CATALOGO DE REFACTORINGS DE BASES DE DATOS

Desde nuestro punto de vista, el refactoring de bases de datos es un punto clave si se quiere aplicar un enfoque evolutivo en el desarrollo de bases de datos. Por este motivo, el objetivo del presente trabajo es desarrollar una herramienta extensible que facilite e incentive la adopción de un proceso evolutivo al desarrollo de bases de datos. La herramienta automatizará las tareas de refactoring de base de datos según los lineamientos establecidos en [AS 06]. Basándose en un modelo de lógico de una base de datos, aplicará el refactoring y generará las sentencias necesarias DDL, DML y SQL para implementarlo en una base de datos existente.

Los detalles de la herramienta se tratarán en el Capítulo 6. Por el momento, presentamos un catálogo de los refactorings elegidos de [AS 06] que implementará la herramienta.

## 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.

# 5. HERRAMIENTAS UTILIZADAS

En el presente capitulo presentaremos una descripción de las principales herramientas utilizadas en el desarrollo de la herramienta propuesta.

## 5.1 ECLIPSE

Eclipse es principalmente una comunidad que fomenta el código abierto. Los proyectos en los que trabaja se enfocan principalmente en construir una plataforma de desarrollo abierta. Esta plataforma puede ejecutarse en múltiples sistemas operativos, lo que convierte a Eclipse en una multiplataforma. Está compuesta por frameworks extensibles, y otras herramientas que permiten construir y administrar software. Permite que los usuarios colaboren para mejorar la plataforma existente y extiendan su funcionalidad a través de la creación de plugins.

Eclipse es también el entorno de desarrollo integrado (IDE, Integraded Development Environment) de código abierto multiplataforma para desarrollar lo que el proyecto llama Aplicaciones de Cliente Enriquecido (RCP, Rich Client Platform), opuesto a las aplicaciones Cliente liviano basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados, como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse y que son usados también para desarrollar el mismo Eclipse. Sin embargo, también se puede usar para otros tipos de aplicaciones cliente.

Esta plataforma es definida en la página Web oficial (www.eclipse.org): "*The Eclipse Platform is an IDE for everything and nothing in particular*" (la Plataforma Eclipse es un IDE para todo y nada en particular), una poderosa herramienta que permite integrar diferentes aplicaciones para construir entornos de desarrollo integrado (IDEs) que pueden ser utilizados para la construcción de aplicaciones Web, Java, C/C++, entre otras, dando a los desarrolladores la libertad de elegir en un entorno multilenguaje y multiplataforma.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas VisualAge. Inicialmente fue un entorno de desarrollo para Java, escrito en Java. Pero luego fue extendiendo el soporte a otros lenguajes de programación. Eclipse ganó popularidad debido a que la plataforma Eclipse proporciona el código fuente de la plataforma y esta transparencia generó confianza en los usuarios.

Eclipse fue liberado originalmente bajo la Common Public License, pero luego fue re licenciado bajo la Eclipse Public License. Según La Free Software Foundation, ambas licencias son licencias de software libre, pero son incompatibles con licencia pública general de GNU (GNU GPL).

Entre los años 2003 y 2004, el Consorcio Eclipse, un consorcio no oficial de empresas fabricantes de software, liderado por IBM, creó la Fundación Eclipse, una entidad legal sin fines de lucro para llevar las riendas y el desarrollo de Eclipse. La Fundación es una organización independiente sin fines de lucro que fomenta una comunidad de código abierto. La misma trabaja sobre un conjunto de productos

complementarios, capacidades y servicios que respalda y se encarga del desarrollo continuo de la plataforma.

La comunidad Eclipse trabaja actualmente en más de 60 proyectos. Los proyectos más importantes engloban otros proyectos, entre ellos podemos mencionar:

- **The Eclipse Project (Eclipse SDK):** es un proyecto de desarrollo de software libre destinado a proporcionar una plataforma de desarrollo de herramientas integradas robusta, completa y comercial. Se subdivide, a su vez, en subproyectos:
  - La propia *plataforma*, que contiene las herramientas Eclipse.
  - *JDT* (Java Development Toolkit): añade a la plataforma un IDE de Java completamente equipado, incluyendo: editores, herramientas de refactoring, compilador y debuger.
  - *PDE* (Plugin Development Environment): es un conjunto de herramientas diseñadas para ayudar al desarrollador de Eclipse en las tareas de desarrollo, prueba, debug, construcción y distribución de plugins.

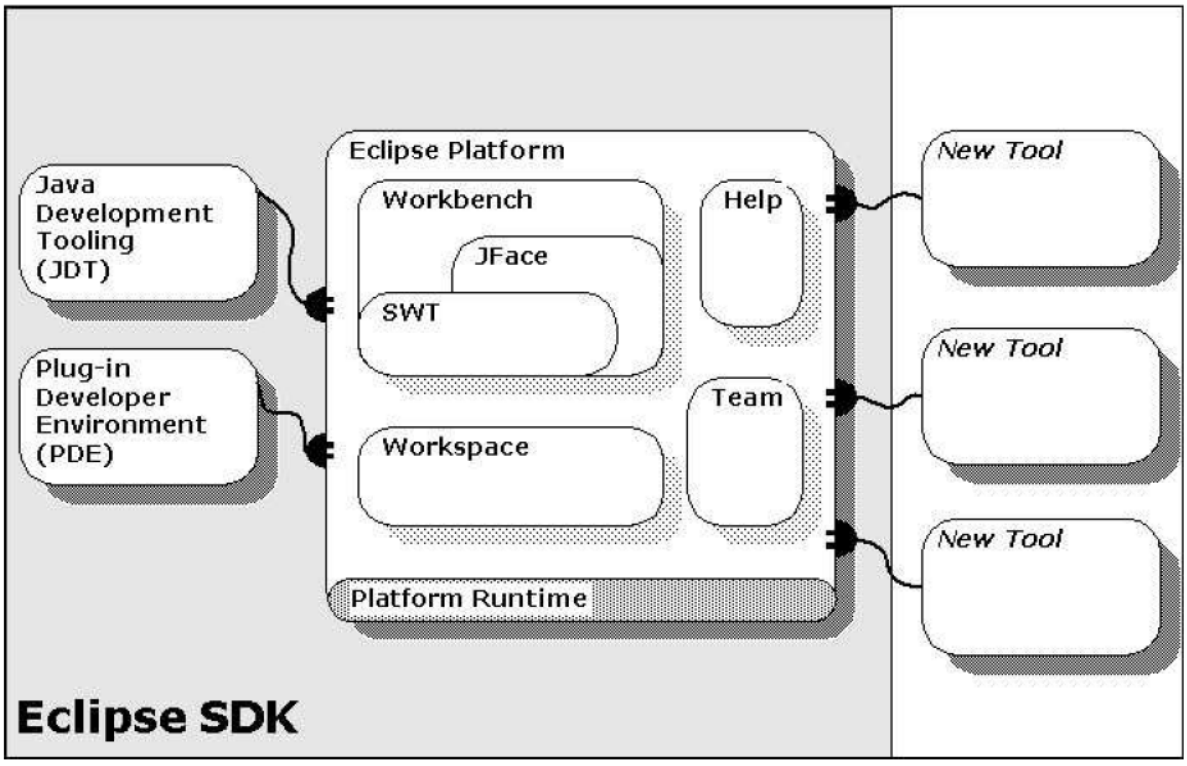


Figura 5.1: El Proyecto Eclipse

- **The Eclipse Tools Project:** su misión es fomentar la creación de una gran variedad de herramientas, proporcionando un punto de coordinación entre los desarrolladores para minimizar la duplicación y promover la interoperabilidad entre los diversos tipos de herramientas.
- **The Eclipse Technology Project:** su misión es proporcionar nuevos canales de comunicación a desarrolladores, profesores y educadores para

participar en la evolución de Eclipse. Está organizado en tres proyectos relacionados:

- *Research*: investigación en dominios relacionados con Eclipse, tales como lenguajes de programación, herramientas y entornos de desarrollo.
- *Incubators*: son pequeños proyectos que añaden capacidades al software base de Eclipse.
- *Education*: estos proyectos se centran en el desarrollo de material de ayuda.

- **The Eclipse Modeling Project:** El proyecto se centra en la evolución y promoción de tecnologías de desarrollo basadas en MDD dentro de la comunidad Eclipse, proporcionando un conjunto unificado de frameworks de modelado, herramientas e implementaciones de los estándares. Este proyecto está compuesto por otros subproyectos relacionados. Cada uno de ellos, también engloba otros proyectos relacionados.
  - *Abstract Syntax development:* El proyecto principal es Eclipse Modeling Framework (EMF), un framework de modelado y generación de código para la construcción de herramientas y otras aplicaciones basadas en un modelo de datos estructurado.
  - *Concrete Syntax development:* El proyecto Graphical Modeling Project (GMP) proporciona componentes generativos e infraestructura de ejecución para el desarrollo de editores gráficos basados en EMF.
  - *Model Transformation:* Engloba dos poryectos principales, M2M y M2T.
    - *Model to Text Transformation (M2T):* un framework extensible para lenguajes de transformación de modelo a modelo, con una ejemplar implementación del núcleo del lenguaje QVT.
    - *Model to Text Transformation (M2T):* se centra en tecnologías para la transformación de los modelos en texto (código fuente normalmente el lenguaje y los recursos que consume). Los proyectos principales en esta categoría son Acceleo, Java Emitter Templates (JET) y Xpand.
  - *Model Development Tools (MDT):* se enfoca en la gran "M" de modelado del proyecto de Modeling. Sus objetivos principales son proveer una implementación de metamodelos estándares de la industria (por ejemplo UML2, OCL, OMD, XSD) y proveer herramientas de desarrollo basadas en estos modelos.
  - *Technology & Research:* el proyecto Generative Modeling Technologies (*GMT*) engloba varios proyectos de investigación en el área de MDD.

### *5.1.1 Arquitectura de la plataforma Eclipse*

Considerándola desde términos de diseño, la Plataforma Eclipse no ofrece gran funcionalidad por sí sola, sino que su valor real yace en el modelo de plugins (unidades mínimas de funcionalidad, explicadas más adelante). Con lo cual, Eclipse está estructurado como un conjunto de subsistemas los cuales son implementados en uno o más plugins que corren sobre una pequeña plataforma de ejecución (Figura 5.2). Dichos subsistemas definen puntos de extensión para permitir agregar funcionalidad a la plataforma. A continuación se describen los principales componentes de la plataforma.

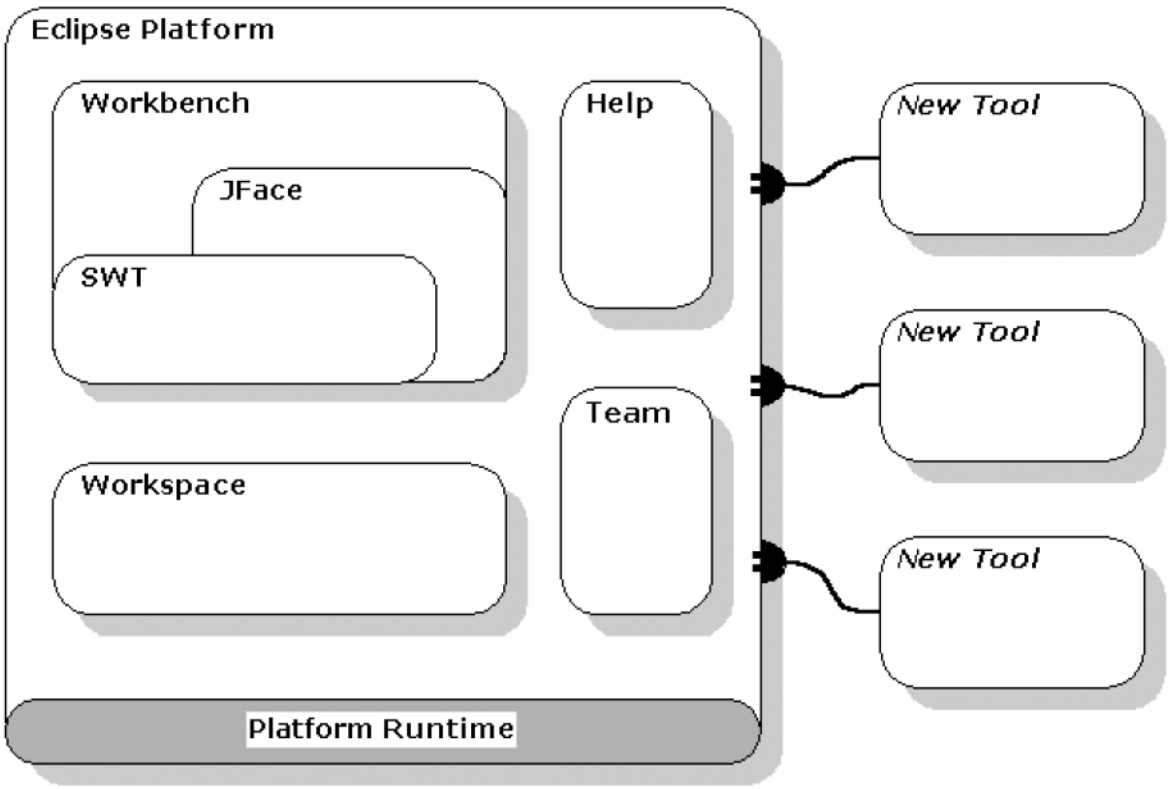


Figura 5.2: Arquitectura de la Plataforma Eclipse

**Plataforma de Ejecución (Platform Runtime)**

Se trata del único componente de Eclipse que no es un *plugin*. Al iniciar la Plataforma de Ejecución se descubren de manera dinámica el conjunto de *plugins* disponibles, se leen sus archivos *manifiest*, y se construye en memoria un registro de *plugins*, que está disponible a través de la API de la Plataforma. La plataforma mantiene el registro de aquellos *plugins* instalados así como de las funcionalidades que proveen. No podrán ser añadidos nuevos *plugins* después del inicio.

Para agregar nuevas funciones al sistema se usa un modelo de extensión común. Los *puntos de extensión* son lugares bien definidos dentro del sistema que permiten ser extendidos por *plugins*. Cuando una herramienta contribuye con una implementación para determinado punto de extensión se dice que agrega una extensión a la plataforma. A su vez, cada *plugin* puede definir sus propios puntos de extensión de tal forma que puedan ser extendidos por otros. Este mecanismo de extensión es la única manera de agregar funcionalidad a la plataforma. Las extensiones están típicamente escritas en Java utilizando la API de la plataforma. Sin embargo, algunos puntos de extensión tienen asociadas extensiones provistas de ejecutables o incluso extensiones que han sido desarrolladas mediante lenguajes script. De forma general, solo un subconjunto del total de la funcionalidad de la plataforma se encuentra disponible para extensiones no realizadas con Java.

Un objetivo muy importante del Runtime es que usuarios finales no sufran desventajas a causa del uso de memoria por aquellos *plugins* que, si bien están instalados, no están siendo usados. De esta manera un *plugin* puede ser instalado y

agregado al registro pero el mismo no será activado a menos que se requiera mediante la actividad del usuario.

### Workspace

Se trata del bloque central, o espacio de trabajo, para los archivos regulares que son específicos de cada usuario, y sobre los que actúan las diferentes herramientas instaladas en la Plataforma.

El *workspace* del usuario consta de uno o más proyectos donde cada uno se mapea en un directorio especificado por el usuario en el sistema de archivos. Cada proyecto contiene los archivos que son creados y manipulados por el usuario. Todos los archivos en el *workspace* son directamente accesibles por programas estándar y herramientas del sistema operativo.

El conjunto de proyectos, archivos y carpetas que son generados por herramientas y almacenados en el sistema de archivos constituyen los recursos. Los recursos del *workspace* están organizados en una estructura de árbol, con los proyectos arriba y los archivos y carpetas por debajo. Existe un recurso especial, el *workspace root*, que funciona como raíz del árbol de recursos. Éste es creado internamente cuando el *workspace* es creado y existirá mientras éste exista.

Los nombres de los recursos son cadenas de caracteres arbitrarias y cualquier cadena es admitida excepto ".metadata" ya que es usada internamente.

La raíz implícita del árbol es el *workspace root* y cada uno de los proyectos es inmediatamente hijo suyo. Los proyectos pueden contener archivos y carpetas, pero no otros proyectos. Los proyectos y carpetas son funcionan como directorios de un sistema de archivos, es decir, si se borra un proyecto o una carpeta también serán borrados todos los recursos que contengan.

### Workbench

Implementa el aspecto visual que permite al usuario navegar por los recursos y utilizar las herramientas integradas. El *workbench* es simplemente un frame donde se presentan varias partes visuales. Estas partes se pueden dividir en dos categorías mayores: editores y vistas.

Los editores permiten al usuario abrir, editar y guardar objetos. El *workbench* incluye un editor sencillo de texto. A partir de puntos de extensión para contribución de nuevos editores se pueden añadir en modo de plugins editores para Java, HTML, ect.

Las vistas proporcionan información sobre aquellos objetos con los que el usuario está trabajando en el *workbench*. Con lo cual, las vistas cambiarán su contenido al cambiar el objeto que seleccione el usuario.

Además de las vistas y editores generales se encuentran las perspectivas. En un *workbench* puede haber una o más perspectivas. Cada una consiste en una agrupación de vistas y editores que se presentan juntos en pantalla para que, desde el punto de vista del usuario, sea una ventana de *workbench* sencilla. Si bien cada proyecto tiene distintas perspectivas solamente se podrá visualizar una a la vez.

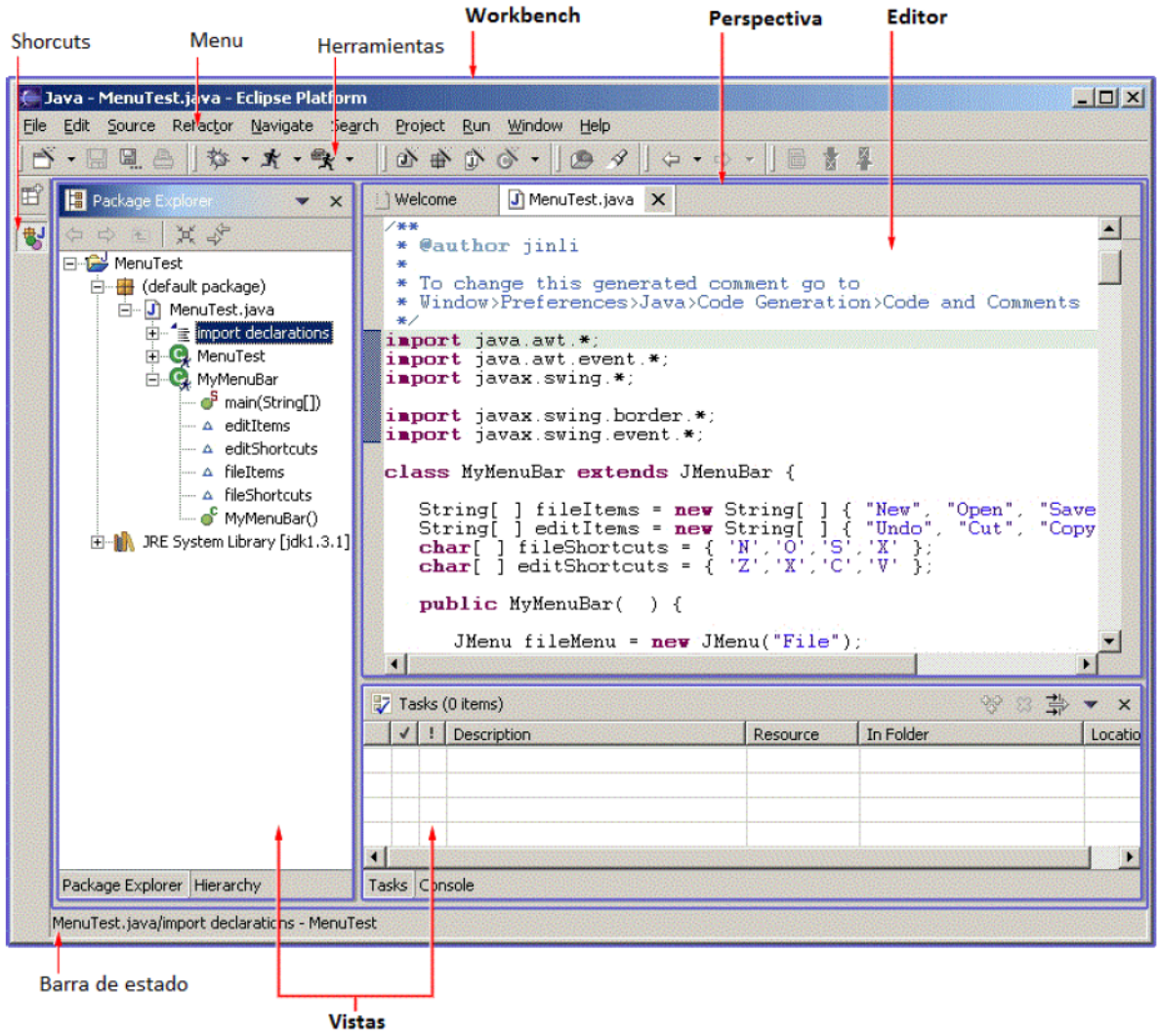


Figura 5.3: El workbench de Eclipse

El ***Standard Widget Toolkit (SWT)*** es una biblioteca de bajo nivel de componentes gráficos y utilidades. Esta biblioteca fue diseñada como alternativa de AWT y Swing, añadiendo la ventaja de que se integra muy bien con el sistema operativo.

Otro componente destacado es ***JFace***, que es una interfaz gráfica implementada sobre SWT que simplifica las tareas de programación poniendo orden en el desarrollo con SWT. JFace es un "UI toolkit", es decir, conjunto de herramientas para el desarrollo de interfaces (*widgets*) que, proporcionando un conjunto de clases, permiten simplificar el desarrollo de características gráficas difíciles de implementar.

### Team Support

El componente Team o Team Support define un modelo de programación en equipo para crear y mantener un registro de las versiones de las aplicaciones que se desarrollen. Este componente permite que diferentes *plugins* de repositorios convivan dentro de la plataforma.

Un repositorio es, en esencia, una base de datos que almacena código y material complementario. Cuando se usa un repositorio el código fuente con el que se trabaja se almacena en la base de datos, no en archivos, y se extrae de él cuando se necesita. Este sistema de almacenamiento ofrece ventajas para los lenguajes orientados a objetos, pues resulta más fácil seguir la pista a las clases, con sus relaciones de herencia, en una base de datos.

Asimismo, el componente Team, añade las vistas que el usuario necesite para interaccionar con cualquier sistema de control de versiones (si hay alguno) que se esté usando.

Eclipse incluye de forma estándar un *plugin* CVS (Concurrent Version System), uno de los sistemas de control de código fuente más usados y muy útil para llevar el control de las versiones con las que se trabaja y conocer en todo momento sus diferencias, pero pueden añadirse repositorios como ChangeMan (Serena), ClearCase (Rational), CM Synergy (Telelogic), PVCS (Merant), StarTeam (Starbase), Subversion (SVN), Git. Independientemente del sistema de control de versión que se use, la interfaz de usuario no cambia.

**Help**

El componente de ayuda (Help) permite a los *plugins* proporcionar documentación HTML que pueda ser presentada contextualmente por el *workbench*. Las facilidades para configurar este contenido HTML en algo estructurado están expresadas en archivos XML externos. Esta separación permite que documentación HTML preexistente pueda ser incorporada directamente en los libros online sin necesidad de ser editada.

También define puntos de extensión que los *plugins* pueden utilizar para contribuir con ayudas u otro tipo de documentación.

Por citar algún ejemplo, la plataforma Eclipse proporciona una "Guía de usuarios del Workbench" y una "Guía de desarrollo de plugins", y también una guía para cada uno de lo *plugins*. Aparte tanto JDT como PDE contribuyen con sus propias ayudas.

**Debug**

El componente de Debug proporciona un modelo genérico de depuración potente, sencilla y muy cómoda de utilizar, junto a una interfaz gráfica genérica de depuración.

Cualquier *plugin* puede aprovechar los mecanismos de debug que proporciona este componente.

Cuando el debugger entra en acción, de forma automática, se abre la Perspectiva Debug, en la que se muestra toda la información relativa al programa que se está depurando. Cuando la Perspectiva Debug está activa, el menú contextual del Editor cambia para mostrar opciones de debug, por ejemplo, ejecutar el programa hasta la línea que tiene el cursor, inspeccionar una variable (o una expresión seleccionada), etc.

Una característica de Eclipse como IDE es el soporte que ofrece Eclipse a HotSwap (cambio en caliente). Esta propiedad, incluida en la Java Platform Debugger Architecture desde la versión 1.4, permite sustituir código o modificarlo mientras se está ejecutando

un programa, sin necesidad de pararlo. Podemos establecer breakpoints (puntos de interrupción) en el código, ejecutarlo, comprobar el estado de las variables en los breakpoints cuando éstos se alcancen, modificar el código o introducir código nuevo, grabar los cambios y continuar la ejecución del programa. Los fallos pueden localizarse y analizarse al vuelo, sin vernos obligados a salir de la aplicación, cambiar el código, recompilar y comenzar una nueva sesión de depuración.

### 5.1.2 PLUGINS

Un *plugin* es la unidad mínima de funcionalidad de Eclipse que puede ser distribuida de manera separada. Herramientas pequeñas se escriben como un único *plugin*, mientras que en las complejas la funcionalidad está en varios *plugins*. Excepto un pequeño núcleo de la plataforma Eclipse, el Platform Runtime, el resto de la funcionalidad de la plataforma Eclipse está implementada como *plugin*.

Los *plugins*, por lo general, están escritos en Java. Cada uno está formado por código Java, archivos de lectura y otros recursos como imágenes, catálogos de mensajes, bibliotecas de código nativo, etc.

Existen algunos *plugins* que no contienen nada de código, por ejemplo: el *plugin* que proporciona ayuda en forma de páginas HTML. Todos los recursos que componen el *plugin* se encuentran en un directorio del sistema de archivos del sistema operativo, o en una URL de un servidor. Existe un mecanismo que permite que un *plugin* pueda ser sintetizado a partir de distintos fragmentos, cada uno en su propio directorio o en su propia URL. Este mecanismo es utilizado para distribuir diferentes paquetes de idiomas para un *plugin* que soporte diversos idiomas (internacionalización).

Para añadir *plugins* a la plataforma existe un único modo, los puntos de extensión. En conformidad con la orientación a objetos, un punto de extensión no deja de ser una interfaz que puede ser implementada por algún desarrollador dispuesto a extender la plataforma.

**Arquitectura y plugin manifest**

Cada *plugin* tiene un archivo denominado de *manifiest* (plugin.xml) en el cual se declaran sus interconexiones con otros *plugins*. La interconexión sigue un modelo muy simple: un *plugin* declara un número de los denominados puntos de extensión, y un número de extensiones para uno o más puntos de extensión de otros *plugins*.

Al iniciar el Platform Runtime se descubren de manera dinámica el conjunto de *plugins* disponibles, se leen sus archivos *manifest*, y se construye en memoria un registro de *plugins*. La Plataforma enlaza cada extensión por el nombre con sus declaraciones de puntos de extensión. No pueden ser añadidos nuevos *plugins* después del inicio.

Los archivos *manifest* de los *plugins* están en formato XML. Un punto de extensión puede declarar tipos de elementos XML adicionales para ser utilizados en las extensiones. Esto permite el paso de datos entre *plugins*. Además, la información del archivo *manifest* está disponible desde el registro de *plugins* sin haber activado los

*plugins* o haber cargado algo de su código. Esto es fundamental para soportar un gran número de *plugins*, a pesar de que sólo un número muy pequeño de ellos sean utilizados por el usuario. Esto marca una diferencia notable, en cuanto a consumo de recursos, entre los IDEs comerciales y Eclipse. Hasta que el código del *plugin* no es cargado, el efecto que tiene sobre el entorno de ejecución es nulo.

Los *plugins* no sólo extienden o amplían la plataforma base, también pueden extender, a su vez, otros *plugins* que hayan definido sus propios puntos de extensión. Es decir, un *plugin* puede hacer públicas interfaces que otros *plugins* pueden implementar. Las implementaciones de las interfaces (llamadas extensiones) mostradas por los puntos de extensión se realizan típicamente en Java, aunque algunos puntos de extensión pueden acomodar extensiones proporcionadas por ficheros ejecutables nativos o componentes ActiveX; incluso pueden programarse en lenguajes de script. El principal obstáculo con el cual se enfrentan las extensiones no realizadas en Java es la falta de acceso a la funcionalidad completa de la plataforma Eclipse.

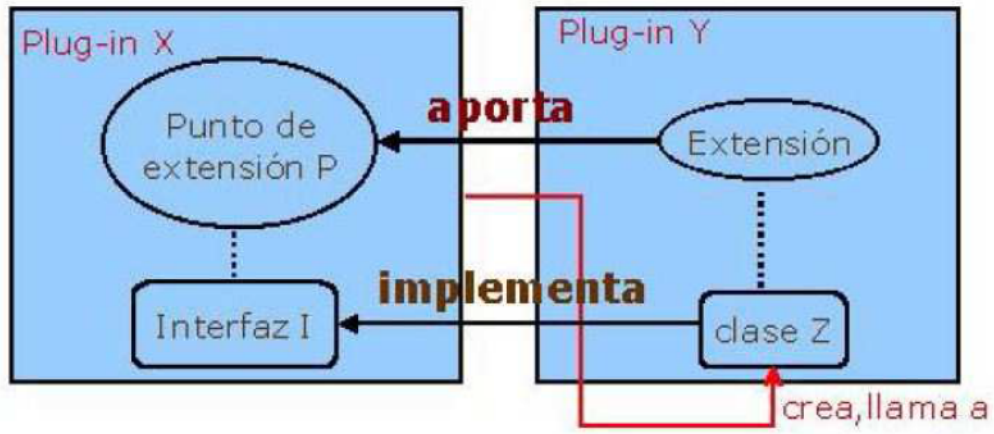


Figura 5.4: Esquema de la extensión de un *plugin* mediante otro *plugin*

Como representa la Figura 5.4, la arquitectura de *plugins* de Eclipse viene marcada por los puntos de extensión y los aportes de los *plugins* a la plataforma así como entre ellos. En el ejemplo de la Figura 5.4, el *plugin* X declara el punto de extensión P y la interfaz I asociada a P. Mientras que el *plugin* Y implementa la interfaz I con su clase Z y por tanto, aporta la clase Z al punto de extensión P. Con lo cual, el *plugin* X será el que instancia la clase Z y el encargado de llamar a sus I-métodos.

### Activación de los plugins

La Plataforma se ejecuta en una única invocación de la máquina virtual Java (JVM). A cada *plugin* se le asigna su propio Java classloader, usado para cargar las clases del *plugin* así como los recursos del mismo.

Un *plugin* es activado cuando su código realmente necesita ser ejecutado. Una vez activado, un *plugin* utiliza el registro de *plugins* para descubrir y acceder a las extensiones que contribuyen a sus puntos de extensión. Por ejemplo, el *plugin* que

declara el punto de acceso de preferencias de usuario puede descubrir todas las preferencias de usuario contribuidoras y mostrarlas para construir un cuadro de diálogo con todas ellas. Esto puede ser realizado, sin necesidad de cargar el código de esos *plugins*, simplemente consultando el registro de *plugins*. Esto nos permite eliminar largos tiempos de arranque y es una solución de gran escalabilidad.

Es posible saber, qué *plugins* están activos en un momento determinado, para ello, es necesario lanzar la ayuda de Eclipse (desde el menú principal, seleccionado la entrada "Help About Eclipse Platform"). Desde la ventana que se despliega, se puede ver la lista de *plugins*.

Una vez activado, un *plugin* permanece activo hasta que la Plataforma finaliza. Cada *plugin* tiene su propio subdirectorio donde poder almacenar datos específicos sobre él; lo que permite mantener datos de sesión entre ejecuciones.

## 5.2 ECLIPSE MODELING FRAMEWORK (EMF)

El proyecto EMF es un framework para modelado, que permite la generación automática de código para construir herramientas y otras aplicaciones a partir de modelos de datos estructurados.

EMF comenzó como una implementación del metalenguaje MOF. En los últimos años se usó para implementar una gran cantidad de herramientas lo que permitió mejorar la eficiencia del código generado. Actualmente el uso de EMF para desarrollar herramientas está muy extendido. Se usa, por ejemplo, para implementar XML Schema Infoset Modelo (XSD), Servicio de Data Objects (SDO), UML2, y Web Tools Platform (WTP) para los proyectos Eclipse. Además EMF se utiliza en productos comerciales, como Omondo, EclipseUML, IBM Rational y productos WebSphere.

EMF permite usar un modelo como el punto de partida para la generación de código, e iterativamente refinar el modelo y regenerar el código, hasta obtener el código requerido. Aunque también prevé la posibilidad de que el programador necesite modificar ese código, es decir, se contempla la posibilidad de que el usuario edite las clases generadas, para agregar o editar métodos y variables de instancia. Siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración.

Como se dijo anteriormente, la generación de código es posible a partir de la especificación de un modelo. De esta manera, permite que el desarrollador se concentre en el modelo y delegue en el framework los detalles de la implementación.

El código generado incluye clases Java para manipular instancias de ese modelo como así también clases adaptadoras para visualizar y editar las propiedades de las instancias desde la vista "propiedades" de Eclipse. Además se provee un editor básico en forma de árbol para crear instancias del modelo. Y por último, incluye un conjunto de casos de prueba para permitir verificar propiedades.

El código generado por EMF es eficiente, correcto y fácilmente modificable. El mismo provee un mecanismo de notificación de cambios de los elementos, una implementación propia de operaciones reflexivas y persistencia de instancias del

modelo. Además provee un soporte básico para rehacer y deshacer las acciones realizadas. Por último, establece un soporte para interoperabilidad con otras herramientas en el framework de Eclipse, incluyendo facilidades para generar editores basados en Eclipse y RCP.

### 5.2.1 El Meta-metamodelo

En EMF los modelos se especifican usando un meta metamodelo llamado Ecore. Ecore es una implementación de eMOF (Essential MOF). Ecore en sí, es un modelo EMF y su propio metamodelo. Existen algunas diferencias entre Ecore y EMOF, pero aun así, EMF puede leer y escribir serializaciones de EMOF haciendo posible un intercambio de datos entre herramientas. La Figura 5.5 muestra las clases más importantes del meta metamodelo Ecore.

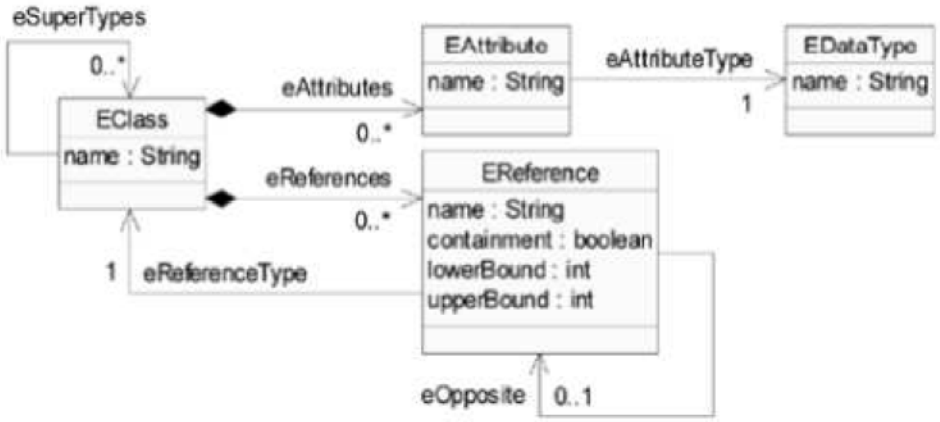


Figura 5.5: Parte del meta metamodelo Ecore

### 5.2.2 Especificación de un metamodelo y su editor

Un metamodelo puede especificarse con un editor gráfico para metamodelos Ecore (por ejemplo, el editor gráfico para Ecore que provee GMF), o como un documento XML, como un diagrama de clases UML o como interfaces de Java con Anotaciones. EMF provee asistentes para interpretar ese metamodelo y convertirlo en un modelo EMF, es decir, en una instancia del meta-metamodelo Ecore, que es el metamodelo usado por EMF (Figura 5.6).

Ecore respeta las metaclases definidas por EMOF: todas las metaclases mantienen el nombre del elemento que implementan y agregan como prefijo la letra "E", indicando que pertenecen al metamodelo Ecore. Por ejemplo, la metaclase EClass implementa la metaclase Class de EMOF. La principal diferencia está en el tratamiento de las relaciones entre las clases. MOF tiene a la asociación como concepto primario, definiéndola como una relación binaria entre clases. Tiene finales de asociaciones, con la propiedad de navegabilidad. En cambio, Ecore define solamente EReferences, como un rol de una

asociación, sin finales de asociación ni Association como metaclases. Dos EReferences pueden definirse como opuestas para establecer una relación navegable para ambos sentidos. Existen ventajas y desventajas para esta implementación. Como ventaja puede verse que las relaciones simétricas, como por ejemplo "esposoDe", implementadas con Association, son difíciles de mantener ya que debe hacerse consistentemente. En cambio con Ecore, al ser sólo una referencia, ella misma es su opuesto, es decir, se captura mejor la semántica de las asociaciones simétricas, y no es necesario mantener la consistencia en el otro sentido. Los modelos son entonces, instancias del metamodelo Ecore. Estos modelos son guardados en formato XMI (XML Metadata Interchange), que es la forma canónica para especificar un modelo.

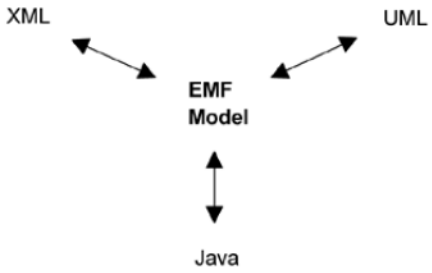


Figura 5.6: Puntos de partida para obtener un modelo EMF

En la Figura 5.7 puede verse el metamodelo del lenguaje Relacional el editor gráfico para Ecore que provee GMF

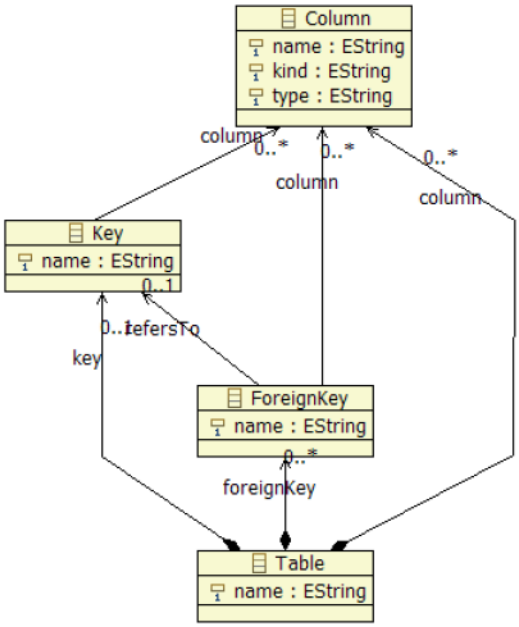


Figura 5.7: Metamodelo del lenguaje Relacional

A partir de un modelo representado como instancias de Ecore, EMF puede generar el código para ese modelo. EMF provee cuatro operaciones para los modelos instancias de Ecore: *Generate Model Code*, *Generate Edit Code*, *Generate Editor Code* y *Generate Test Code*. Mediante estas operaciones, EMF permite generar cuatro plugins.

Con *Generate Model Code* se genera un plugin que contiene el código Java de la implementación del modelo, es decir, un conjunto de clases Java que permiten crear instancias de ese modelo, hacer consultas, actualizar, persistir, validar y controlar los cambios producidos en esas instancias. Por cada clase del modelo, se genera dos elementos en Java: una interface y la clase que la implementa. Todas las interfaces generadas extienden directa o indirectamente a la interface EObject. La interface EObject es el equivalente de EMF a `java.lang.Object`, es decir, la base de todos los objetos en EMF. EObject y su correspondiente implementación EObjectImpl proveen la base para participar en los mecanismos de notificación y persistencia. Con *Generate Edit Code* se genera un plugin con las clases necesarias por el editor. Contiene un conjunto de clases adaptadoras que permitirán visualizar y editar propiedades de las instancias en la vista "propiedades" de Eclipse. Estas clases permiten tener una vista estructurada y permiten la edición de los objetos de modelo a través de comandos. Con *Generate Editor Code* se genera un editor para el modelo. Este plugin define además la implementación de un asistente para la creación de instancias del modelo.

Finalmente, con *Generate Test Code* se generan casos de prueba, que son esqueletos para verificar propiedades de los elementos. También tiene una operación que permite generar todo el código anteriormente mencionado en un solo paso.

En resumen, el código generado es limpio, simple, y eficiente. La idea es que el código que se genera sea lo más parecido posible al que el usuario hubiera escrito, si lo hubiera hecho a mano. Pero por ser generado, se puede tener la confianza que es correcto. EMF establece un soporte para interoperabilidad con otras herramientas en el framework de Eclipse ya que genera código base para el desarrollo de editores basados en Eclipse y RCP. Como se mencionó, el generador de código de EMF produce archivos que pretenden que sean una combinación entre las partes generadas y las partes modificadas por el programador. Se espera que el usuario edite las clases generadas, para agregar o editar métodos, variables de instancia. Siempre se puede regenerar desde el modelo cuando se necesite, y las partes agregadas serán preservadas durante la regeneración. EMF usa los marcadores `@generated` en los comentarios JavaDoc de las interfaces, clases y métodos generados para identificar las partes generadas. Cualquier método que no tenga ese marcador se mantendrá sin cambios luego de una regeneración de código. Si hay un método en una clase que está en conflicto con un método generado, la versión existente tendrá prioridad.

En la Figura 5.8 puede verse el editor generado por EMF. A la derecha se encuentra la vista *Outline*, que muestra el contenido del modelo que se está editando con una vista de árbol. Cuando se selecciona un elemento en la vista *Outline* se muestra también seleccionado en la primera página del editor, que tiene una forma de árbol similar.

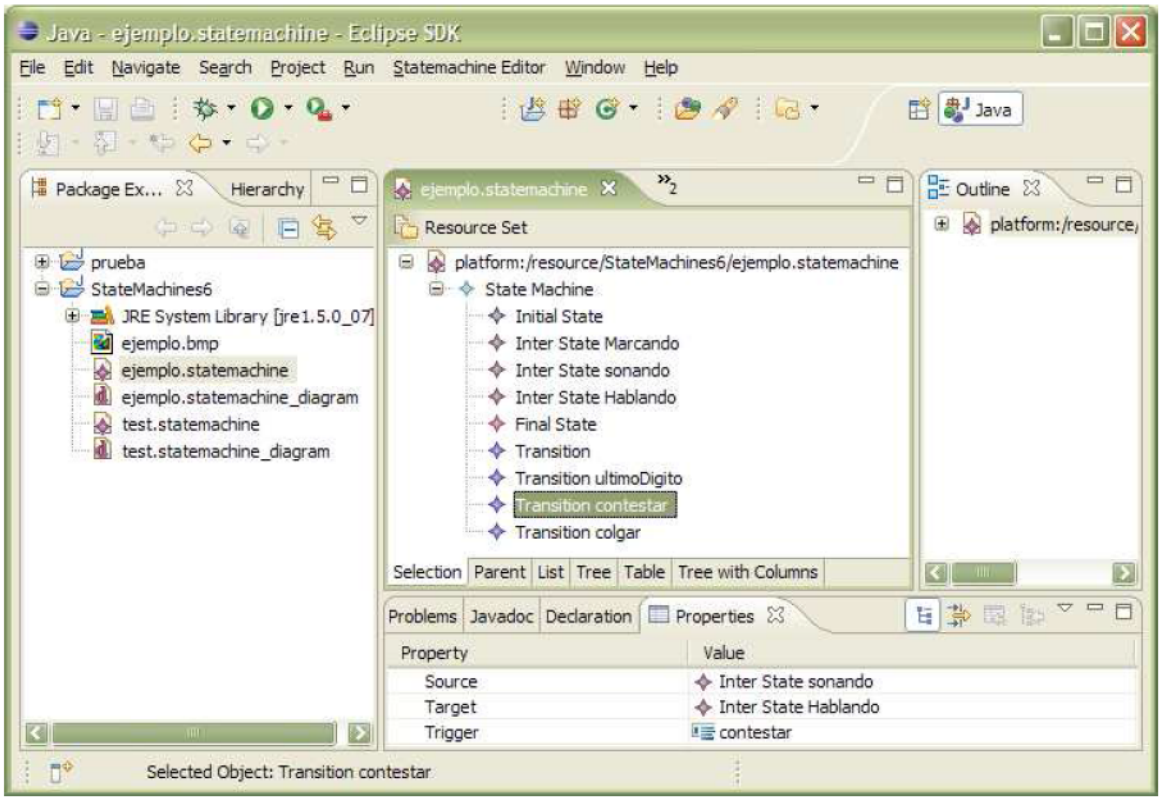


Figura 5.8: Editor generado con EMF

Independientemente de la vista donde se seleccione el elemento, al seleccionarlo se muestran sus propiedades en la vista de propiedades. La vista de propiedades permite editar los atributos del elemento y las referencias a otros elementos del modelo (permite seleccionar de una lista de posibles). Se pueden arrastrar con el mouse los elementos para moverlos, cortarlos, copiarlos y pegarlos, borrarlos, y estas acciones soportan deshacer y rehacer. Las otras páginas del editor, Parent, List, Tree, Table, TableTree permiten otras visualizaciones de los elementos, como en forma de tabla y en forma de lista.

## 5.3 QUERIES, VIEWS AND TRANSFORMATIONS (QVT)

QVT (Queries, Views and Transformations) es el lenguaje estándar de OMG para expresar transformaciones de modelos. QVT se ha convertido en una especificación voluminosa y compleja, por lo que, respecto a su descripción, mostraremos sólo una vista general de su arquitectura y propiedades mediante algunos ejemplos.

El OMG inicialmente publicó un Request for Proposal (RFP) para transformaciones Modelo a Modelo (M2M), en Abril de 2002 [QVTR]. Recién en Noviembre de 2005 fue conocida la especificación adoptada. Esta tardanza puede explicarse por la complejidad del problema que se abarca.

Aunque se pueda tener buenas ideas acerca de cómo escribir transformaciones M2M en Java, por ejemplo, es claro que los escenarios M2M reales requieren técnicas más

sofisticadas. El RFP de QVT solicitó propuestas dirigidas a este nivel de sofisticación, aún pensando que tales requisitos no estaban muy explorados ni conocidos en ese momento. Otro problema fue que la experiencia previa existente con transformaciones M2M era prácticamente nula en los comienzos. Esto no fue una buena posición de partida desde la cual generar un estándar, que idealmente es una consolidación de tecnologías existentes.

Esta situación fue agravada por el hecho de que ocho grupos diferentes remitieron propuestas iniciales. La mayoría de estas propuestas diferían marcadamente entre sí, por lo que no hubo bases claras para la consolidación. Como resultado, llevó una considerable cantidad de tiempo encontrar coincidencias y aún hoy, el resultado obtenido especifica tres lenguajes QVT diferentes que sólo se conectan débilmente, justificando la naturaleza híbrida del estándar –relacional y operacional- que intenta abarcar las diferentes respuestas al RFP.

La especificación de QVT, como dijimos, tiene una naturaleza híbrida, relacional (o declarativa) y operacional (o imperativa). Comprende tres diferentes lenguajes M2M: dos lenguajes declarativos llamados *Relations* y *Core*, y un tercer lenguaje, de naturaleza imperativa, llamado *Operational Mappings*. Esta naturaleza híbrida fue introducida para cubrir diferentes tipos de usuarios con diferentes necesidades, requisitos y hábitos. Esta estrategia no resulta sorprendente si se consideran las numerosas propuestas iniciales generadas por el RFP, cada una con distintas expectativas de uso y funcionalidad.

Entonces, la especificación de QVT define tres paquetes principales, uno por cada lenguaje definido: QVTCore, QVTRelation y QVTOperational, como puede observarse en la Figura 5.9. Estos paquetes principales se comunican entre sí y comparten otros paquetes intermedios (Figura 5.10).

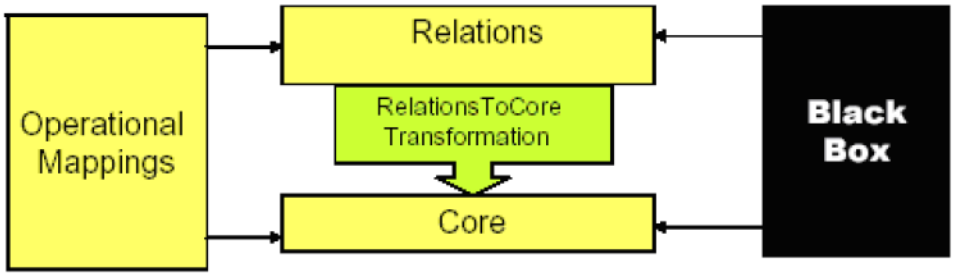


Figura 5.9: Relaciones entre metamodelos QVT

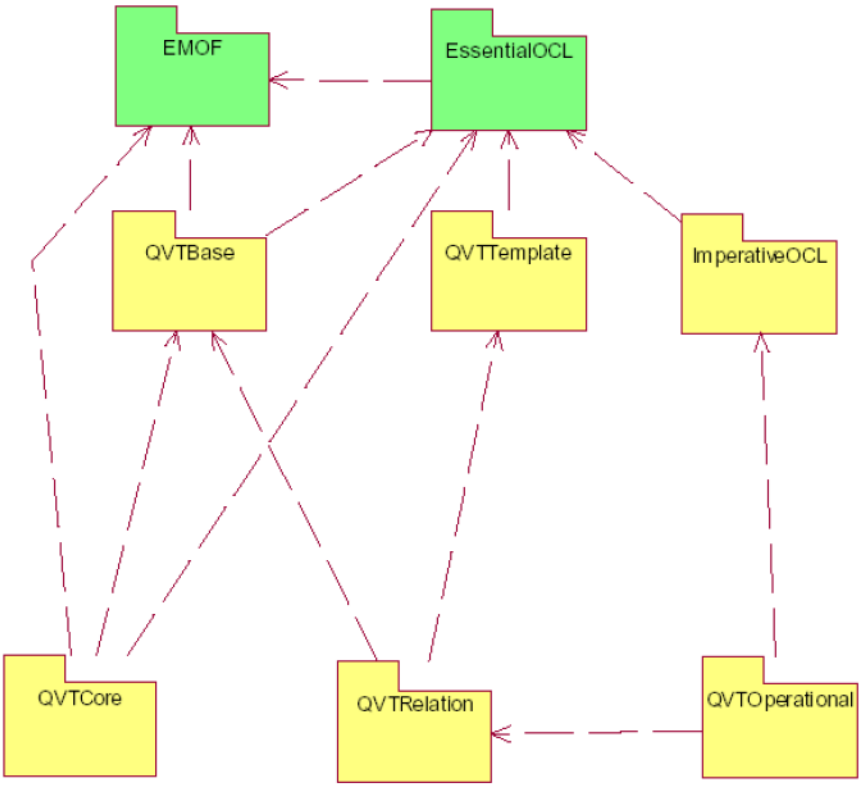


Figura 5.10: Dependencias de paquetes en la especificación QVT

El paquete QVTBase define estructuras comunes para transformaciones. El paquete QVTRelation usa expresiones de patrones template definidas en el paquete QVTTemplateExp. El paquete QVTOperational extiende al QVTRelation, dado que usa el mismo framework para trazas. Usa también las expresiones imperativas definidas en el paquete ImperativeOCL. Todos los paquetes QVT dependen del paquete EssentialOCL de OCL 2.0, y a través de él también dependen de EMOF (EssentialMOF).

### *5.3.1 QTV DECLARATIVO*

La parte declarativa de QVT está dividida en una arquitectura de dos niveles. Las capas son:

- Un lenguaje *Relations* amigable para el usuario, que soporta pattern matching complejo de objetos y creación de template para objetos. Las trazas entre elementos del modelo involucrados en una transformación se crean implícitamente. Soporta propagación de cambios, ya que provee un mecanismo para identificar elementos del modelo destino. En este lenguaje, una transformación entre modelos se especifica como un conjunto de relaciones que deben establecerse para que la transformación sea exitosa. Es una especificación declarativa de las relaciones entre modelos MOF.
- Un lenguaje *Core* definido usando extensiones minimales de EMOF y OCL. Las trazas no son automáticamente generadas, se definen explícitamente

como modelos MOF, y pueden crearse y borrarse como cualquier otro objeto. El lenguaje *Core* no soporta pattern matching para los elementos de modelos. Esta propuesta absolutamente minimal lleva a que el *Core* sea el "assembler" de los lenguajes de transformación.

No profundizamos en la descripción de este lenguaje debido a que no lo utilizamos en nuestra implementación.

### *5.3.2 QVT OPERACIONAL*

Además de sus lenguajes declarativos, QVT proporciona dos mecanismos para implementaciones de transformaciones: un lenguaje estándar, *Operational Mappings*, e implementaciones no-estándar o *Black-box*.

El mecanismo de caja negra (*black-box),* permite implementaciones opacas (escritas en otro lenguaje existente) de parte de transformaciones. Una implementación caja negra no tiene relación explicita con el lenguaje *Relations* de QVT declarativo, pero cada caja negra podría implementar una Relation, la cual es responsable de mantener las trazas entre los elementos relacionados.

Por su parte, el lenguaje *Operational Mappings* se especificó como una forma estándar para proveer implementaciones imperativas. Proporciona una extensión del lenguaje OCL mediante el agregado de nuevas construcciones con efectos laterales que permiten un estilo más procedural, y una sintaxis que resulta familiar a los programadores.

Este lenguaje puede ser usado en dos formas diferentes. Primero, es posible especificar una transformación únicamente en el lenguaje *Operational Mappings*. Una transformación escrita usando solamente operaciones Mapping es llamada Transformación Operacional. Alternativamente, es posible trabajar en modo híbrido. El usuario tiene entonces que especificar algunos aspectos de la transformación en el lenguaje *Relations* (o *Core*), e implementar reglas individuales en lenguaje imperativo a través de operaciones Mappings.

Una *Transformación Operacional* representa la definición de una transformación unidireccional, expresada imperativamente. Tiene una signatura indicando los modelos involucrados en la transformación y define una operación entry, llamada *main*, la cual representa el código inicial a ser ejecutado para realizar la transformación. La signatura es obligatoria, pero no así su implementación. Esto permite implementaciones de caja negra definidas fuera de QVT.

El ejemplo que sigue muestra la signatura y el punto de entrada de una transformación llamada Uml2Rdbms, que transforma diagramas de clase UML en tablas RDBMS.

```
transformation Uml2Rdbms (in uml: UML, out rdbms: RDBMS) {
//el punto de entrada para la ejecución de la transformación.
main () {
uml.objectsOfType (Package) -> map packageToSchema ();
}
...
}
```

La signatura de esta transformación declara que un modelo *rdbms* de tipo RDBMS será derivado desde un modelo *uml* de tipo UML. En el ejemplo, el cuerpo de la transformación (*main*) especifica que en primer lugar se recupera la lista de objetos de tipo Paquete y luego se aplica la operación de mapeo (mapping operation) llamada packageToSchema() sobre cada paquete de dicha lista. Esto último se logra utilizando la operación predefinida map() que itera sobre la lista.

Una *Mapping Operation* es una operación implementando un mapping entre uno o más elementos del modelo fuente, en uno o más elementos del modelo destino. Una *Mapping Operation* se describe sintácticamente mediante una signatura, una guarda (su cláusula *when*), el cuerpo del mapping y una poscondición (su cláusula *where*). La operación puede no incluir un cuerpo (es decir, que oculta su implementación) y en ese caso se trata de una operación de caja negra (*black-box*).

Una *Mapping Operation* siempre refina una relación, donde cada dominio se corresponde con un parámetro del mapping. El cuerpo de una operación mapping se estructura en tres secciones opcionales. La sección de inicialización es usada para crear los elementos de salida. La intermedia, sirve para asignarle valores a los elementos de salida y la de finalización, para definir código que se ejecute antes de salir del cuerpo.

La operación mapping que sigue define como un paquete UML se transforma en un esquema RDBMS.

```
mapping Package::packageToSchema() : Schema
when { self.name.startingWith() <> "_"}
{
  name := self.name;
  table := self.ownedElement->map class2table();
}
```

La relación implícita asociada con esta operación mapping tiene la siguiente estructura:

```
relation REL_PackageToSchema {
  checkonly domain:uml (self:Package)[]
  enforce domain:rdbms (result:Schema)[]
  when { self.name.startingWith() <> "_" }
}
```

### Sintaxis Abstracta del Lenguaje Operational Mappings

En esta sección presentamos los principales elementos que definen la sintaxis abstracta del lenguaje Operational Mappings.

La Figura 5.11 muestra el Paquete QVT Operational para Transformaciones Operacionales, del cual detallamos la metaclase OperationalTransformation:

***OperationalTransformation*** representa la definición de una transformación unidireccional, expresada imperativamente.

**Superclases**
`Module`
**Atributos**
`isBlackbox: Boolean (from Module)`

*Indica que la transformación es opaca*

`isAbstract: Boolean (from Class)`

*Indica que la transformación sirve para la definición de otras transformaciones*

**Asociaciones**

`entry: EntryOperation [0..1]`

*Una operación actuando como punto de entrada para la ejecución de la transformación operacional.*

`modelParameter: ModelParameter [*] {composes, ordered}`

*Indica la signatura de esta transformación operacional. Un parámetro de modelo indica un tipo de dirección (in/out/inout) y un tipo dado por un tipo de modelo (ver la descripción de clase ModelParameter).*

`refined: Transformation [0..1]`

*Indica una transformación relacional (declarativa) que es refinada por esta transformación operacional.*

`relation: Relation [0..*] {composes, ordered}`

*El conjunto ordenado de definiciones de relaciones que tienen que están asociadas con las operaciones mapping de esta transformación operacional.*

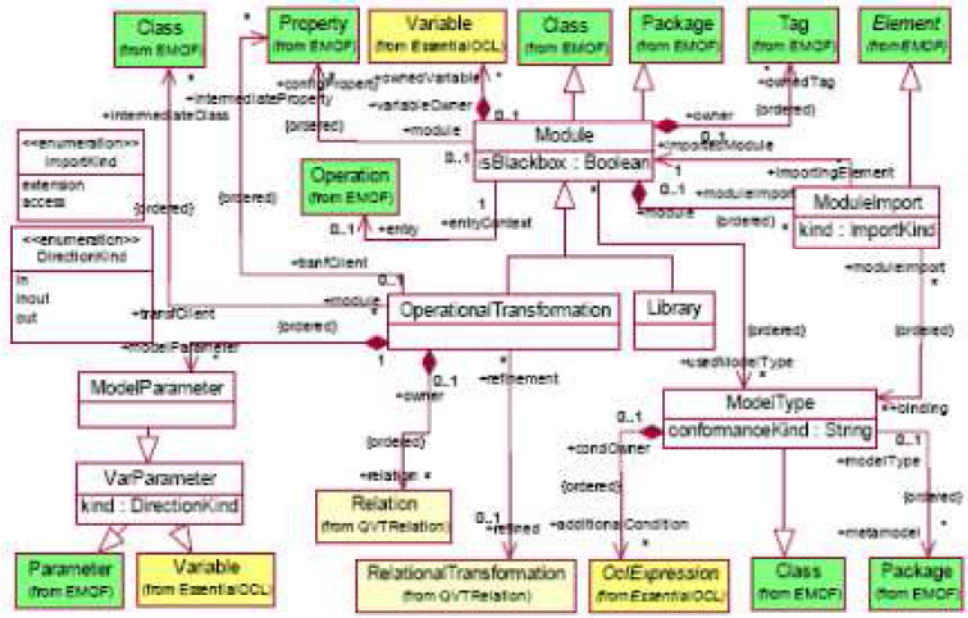


Figura 5.11: Paquete QVT Operational – Transformaciones Operacionales

La Figura 5.12 presenta la jerarquía de las Operaciones Imperativas del Paquete QVT Operational, de la que detallamos la metaclase MappingOperation.

***MappingOperation*** es una operación implementando un mapping entre uno o más elementos del modelo fuente, en uno o más elementos del modelo destino.

**Superclases**

`ImperativeOperation`

**Atributos**

`isBlackbox: Boolean`

*Indica si el cuerpo de la operación está disponible. Si isBlackbox es verdadero esto significa que la definición debería ser proporcionada externamente a fin de que la transformación pueda ser ejecutada.*

**Asociaciones**

`inherited: MappingOperation [*]`

*Indica la lista de operaciones mappings que son especializados.*

`merged: MappingOperation [*]`

*Indica la lista de operaciones mappings que son mezclados.*

`disjunct: MappingOperation [*]`

*Indica la lista de operaciones mappings potenciales para invocar*

`refinedRelation: Relation [1]`

*Indica la relación refinada. La relación define la guarda (when) y la poscondición para la operación mapping.*

`when: OclExpression [0..1] {composes}`

*La pre-condición o guarda de la operación.*

`where: OclExpression [0..1] {composes}`

*La post-condición o guarda de la operación.*

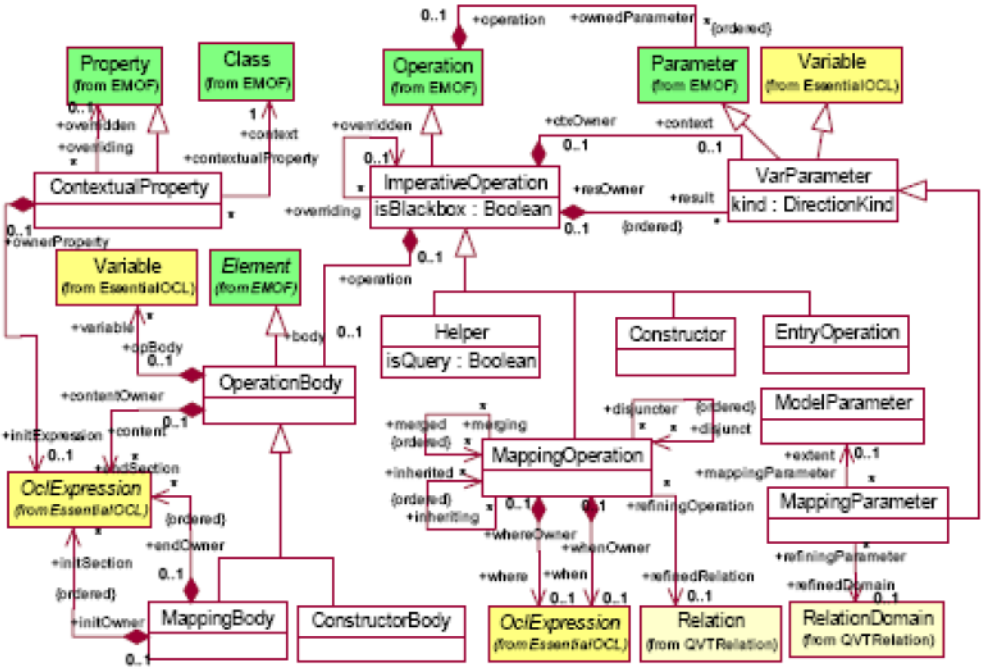


Figura 5.12: Paquete QVT Operational – Operaciones Imperativas

Finalmente, la Figura 5.13 muestra el Paquete OCL Imperativo que extiende a OCL proveyendo expresiones imperativas y manteniendo las ventajas de la expresividad de OCL.

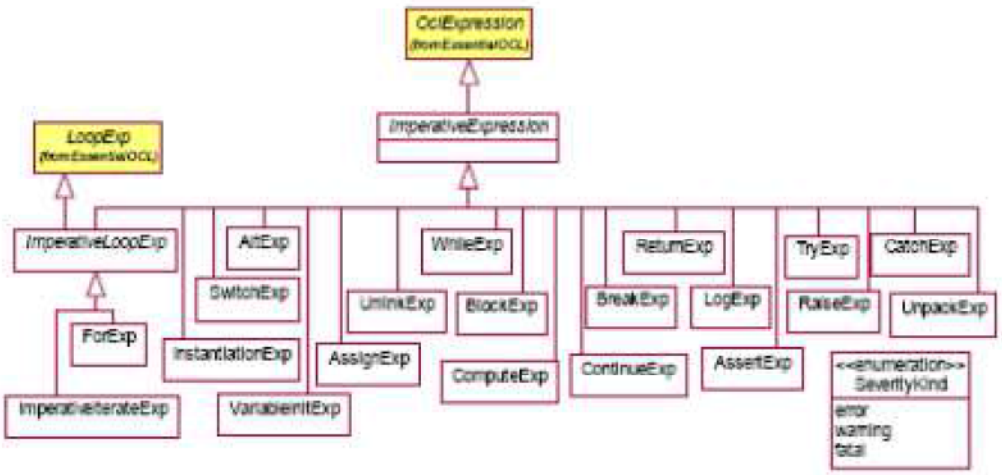


Figura 5.13: Paquete OCL Imperativo

***ImperativeExpression*** es la raíz abstracta de esta jerarquía sirviendo como base para la definición de todas las expresiones con efectos laterales definidas en esta especificación. Tales expresiones son *AssignExp, WhileExp, IfExp*, entre otras. La Superclase de *ImperativeExpression* es *OclExpression*. Podemos notar que en contraste con las expresiones OCL puras, libres de efectos laterales, las expresiones imperativas en general, no son funciones. Por ejemplo, son constructores de interrupción de ejecución como *break*, *continue*, *raise* y *return* que producen un efecto en el flujo de control de las expresiones imperativas que las contienen.

## 5.4 JAVA EMITTER TEMPLATE (JET)

Las transformaciones modelo a modelo crean su modelo destino como una instancia de un metamodelo específico, mientras que el destino de una transformación modelo a texto (M2T) es simplemente un documento en formato textual, generalmente de tipo String.

El proyecto Model to Text (M2T) se centra en la generación de artefactos textuales a partir de modelos. Su objetivo es triple:

- Proporcionar implementaciones de estándares de la industria y los motores M2T estándares de facto en Eclipse.
- Proporcionar herramientas de desarrollo para estos lenguajes.
- Proporcionar infraestructura común para estos lenguajes.

Java Emitter Template (JET) es un subproyecto de Eclipse Modeling dentro de M2T centrado en simplificar el proceso de generación automática de código (Java, XML, JSP o SQL en nuestro caso) a partir de plantillas. JET se utiliza típicamente en la implementación de un generador de código. Un generador de código es un componente importante en Model Driven Development (MDD). El objetivo de MDD es describir un sistema de software utilizando modelos abstractos (como Ecore en nuestro caso), y luego refinar y transformar estos modelos en código. Aunque es posible crear modelos abstractos y convertirlos manualmente en el código, el verdadero poder de MDD proviene de la automatización de este proceso. Tales transformaciones acelerar el proceso de MDD, y dan lugar a la calidad del código mejor. Las transformaciones pueden capturar las mejores prácticas de los expertos, y se puede asegurar que un proyecto consistente emplea estas prácticas.

Sin embargo, las transformaciones no siempre son perfectas. Las mejores prácticas a menudo depende del contexto, lo que es óptimo en un contexto no puede serlo en otro. Las transformaciones pueden abordar este problema mediante la inclusión de algún mecanismo para la modificación de usuario final del generador de código. Esto se suele hacer mediante el uso de plantillas para crear artefactos, y permite a los usuarios sustituir sus propias implementaciones de estas plantillas si es necesario. Este mismo es el papel de JET.

JET proporciona una librería tags estándar que hacen posible la creación de transformaciones completas sin tener que recurrir a Java y ni las API de Eclipse. A su vez, JET se puede expandir para apoyar etiquetas personalizadas, permitiendo que el usuario defina su propia librería de tags. Para esto se tienen interfaces Java y puntos de extensión de Eclipse para la declaración de estas librerías de tags personalizadas.

También proporciona una API para Eclipse y la interfaz de usuario para invocar tales transformaciones, y un editor de plantillas JET.

### 5.4.1 El funcionamiento de JET

Básicamente, el funcionamiento de JET parte de un modelo estructurado, como XML, UML, o un modelo basado en EMF y luego este modelo es "atravesado" por la plantilla o template que se ejecuta, creando Strings, cuyo destino puede ser por ejemplo, un archivo o la salida estándar.

JET funciona a partir de plantillas muy similares a las Java Server Pages (JSP), que es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Al igual que en esta tecnología, los templates JET son traducidos a una clase Java para luego ser ejecutadas por medio de una clase generadora creada por el usuario.

Un aspecto de plantillas JET que al principio es confuso es que la generación de texto tiene dos etapas: traducción y generación. El primer paso es la traducción de la plantilla a una clase de implementación de plantilla. El segundo paso se utiliza esta clase de implementación de plantilla para generar el texto.

Por ejemplo, si el objetivo con JET es generar código fuente de Java, que puede ser confuso que el paso de traducción plantilla también se traduce en código fuente Java. El código fuente no es el texto generado. El código fuente que es el resultado de la etapa de traducción es simplemente otra forma de la plantilla.

Una plantilla JET se traduce a una clase de implementación de plantilla, al igual que una página JSP es traducida a un `Servlet`. El segundo paso, donde la clase de implementación de plantilla genera texto, equivale a la creación de Servlets, la ejecución de alguno de sus métodos (como `doGET`, por ejemplo) y devolver HTML.

Las clases principales de JET pueden dividirse en dos grupos:

- Las clases de bajo nivel que tratan los aspectos de la traducción de una plantilla a una clase de implementación de plantilla. La clase `JETCompiler` encapsula todas estas clases de bajo nivel en para proporcionar una API simple para la traducción de la plantilla.
- Las clases de alto nivel que trabajan con `JETCompiler` para lograr la tarea del usuario, como por ejemplo `JETBuilder` o `JETEmitter` que traducen cada archivo template en la clase generadora.

`JETCompiler` es la clase principal para la traducción de la plantilla. Esta clase se encarga de traducir las plantillas en el código fuente Java de la clase de implementación de plantilla. La traducción real se delega en otras clases que colaboran con esta. Los clientes crean un objeto `JETCompiler` para una plantilla determinada y luego llaman al método de *parsing* seguido por el método que genera el código fuente Java de la clase de implementación de plantilla.

`JETBuilder` genera las traducciones de todas las plantillas en el proyecto que se han modificado desde la generación anterior.

`JETEmitter` cuenta con una API de alto nivel para los usuarios del paquete JET. El método `generate` de esta clase combina la traducción de la plantilla y la generación de texto en un solo paso. Al cuidar de los detalles profundos de la traducción y compilación del código fuente Java de la clase de implementación de la plantilla, `JETEmitter` le permite centrarse en la salida final del generador. De esta forma abstrae el paso de la traducción y permite simular que directamente se puede generar texto desde una plantilla.

La siguiente figura muestra los pasos para llegar al resultado de la ejecución del template usando `JETEmitter`.

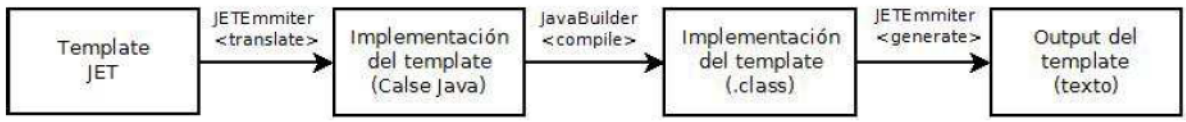


Figura 5.14: Ejecución de un template con JET

# 6. La Herramienta

En este capítulo se abordan distintos aspectos relacionados a la herramienta desarrollada. Se comienza con la descripción de los objetivos de la misma y su alcance, siguiendo con un ejemplo de uso aplicando el refactoring RenameTable en un caso concreto. Luego se describen los aspectos principales de su arquitectura y diseño, y cada una de las componentes de la herramienta. La sección finaliza marcando algunas conclusiones y propuestas de mejoras.

## 6.1 Descripción General

En [AS 06], los autores Scott W. Ambler y Pramod J. Sadalage definen un catálogo de Refactorings de Base de Datos. Los mismos son presentados en forma de patrones; para cada uno de ellos indican 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. Como vimos en los capítulos 3 y 4, la premisa en cada refactoring es lograr una coexistencia entre ambas versiones de la base de datos durante un período de transición determinado, de manera que los distintos clientes de la base de datos cuenten con un margen de tiempo razonable para adaptar sus aplicaciones al modelo nuevo. Entonces,

> *"El objetivo de la herramienta 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"*

Consideramos que esto será de suma utilidad para los desarrolladores de base de datos por todos los beneficios que se desprenden de un proceso automatizado, como evitar trabajo repetitivo, aumentar la fiabilidad, evitar errores humanos, aumentar la productividad y mejorar la calidad del proceso entre otros.

Optamos por encuadrar nuestra herramienta en el marco del Desarrollo de Software Dirigido por Modelos MDD, pues este paradigma promete mejorar el proceso de construcción de software basándose en un proceso guiado por modelos y soportado por potentes herramientas, siendo la transformación entre modelos el motor de MDD. Conceptos como Modelos y Transformaciones se ajustan adecuadamente para las necesidades de la herramienta. Los modelos representarán el lenguaje relacional y las bases de datos concretas. Las transformaciones entre modelo y de modelo a texto proveerán el soporte para modificar un modelo y generar los scripts correspondientes para implementar un refactoring.

La herramienta consiste en un conjunto de plugins Eclipse. La plataforma Eclipse brinda:

- Un conjunto de herramientas orientadas al desarrollo de software basado en MDD como respuesta a las especificaciones de la OMG. Estas componen el proyecto Eclipse model-to-model transformation (M2M) que es un subproyecto del proyecto mas general Eclipse Modeling Project, como se

detalló en el capítulo 5. Estas herramientas se utilizan tanto para la especificación de meta modelos (Ecore) como para la transformación entre instancias de los mismos y consultas a los mismos (ATL, MOFScript, etc.).

- Un conjunto de frameworks basados en MDD para la creación de poderosos editores, que permiten la creación y edición de instancias de nuestros modelos (EMF, GEF y GMF).
- Un entorno de desarrollo basado en las tecnologías Rich Client Platform, SWT y JFace que permite la creación de poderosas interfaces de usuario (UI).

La interacción de la herramienta con el usuario consiste en un *wizard* que permitirá 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 deberá ingresar distintos datos dependiendo del refactoring elegido. Por ejemplo, si selecciona el refactoring *Rename Table*, entonces se deberá suministrar el nombre de la tabla a renombrar y el nuevo nombre para la tabla. 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. En la siguiente sección mostramos en detalle cómo se resuelve el refactoring *Rename Table*.

## 6.2 Ejemplo: Rename Table

Supongamos que el usuario cuenta con el siguiente modelo de base de datos, y desea renombrar la tabla Movie por Film:

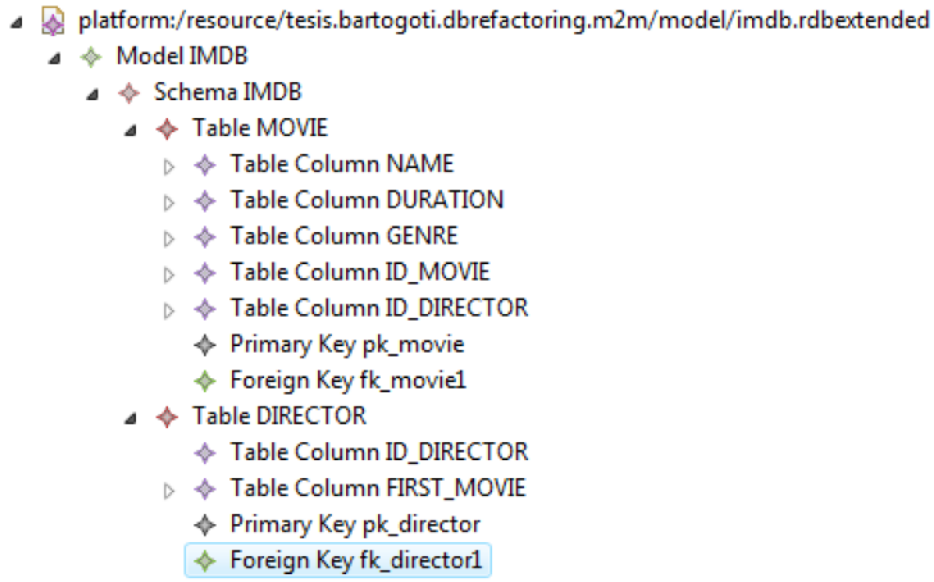


Figura 6.1: Esquema de base de datos definido como instancia de `rdbExtended`

Se ingresa a la herramienta por el Menú DB Refactoring, o el icono con la llave en la barra de herramientas, o CTR + 6.


Figura 6.2: Ícono y entrada de menú de la herramienta

En la primera página se deberá seleccionar el modelo al cual se le aplicará el refactoring y el tipo de refactoring, *Rename Table* en este caso:

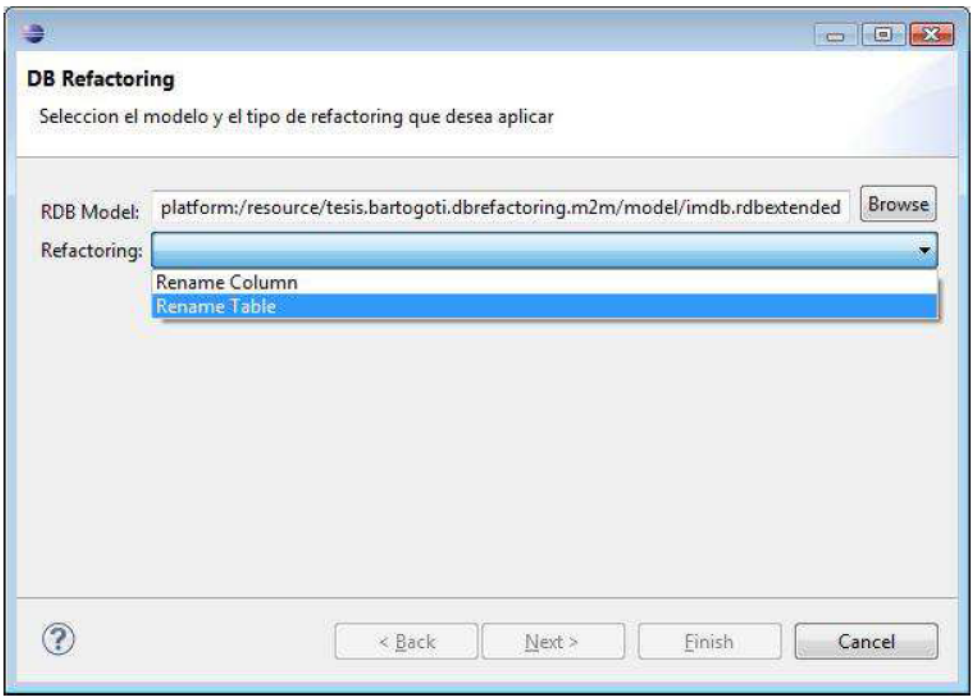


Figura 6.3: Primera página del wizard de la herramienta

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:

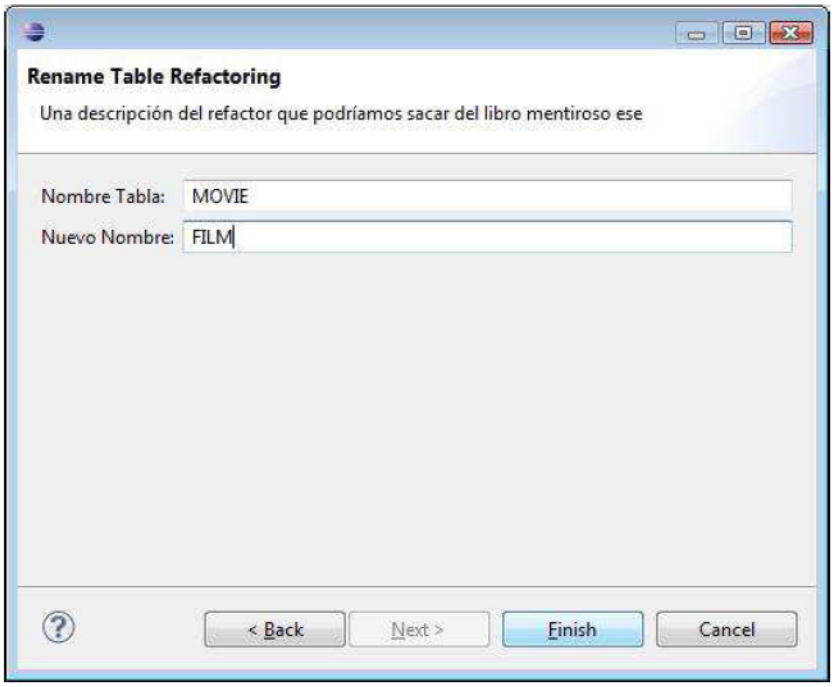


Figura 6.4: Segunda página del wizard de la herramienta para el refactoring *Rename Table*

Para terminar click en *Finish* y la herramienta mostrará un mensaje indicando que el refactoring fue aplicado correctamente:

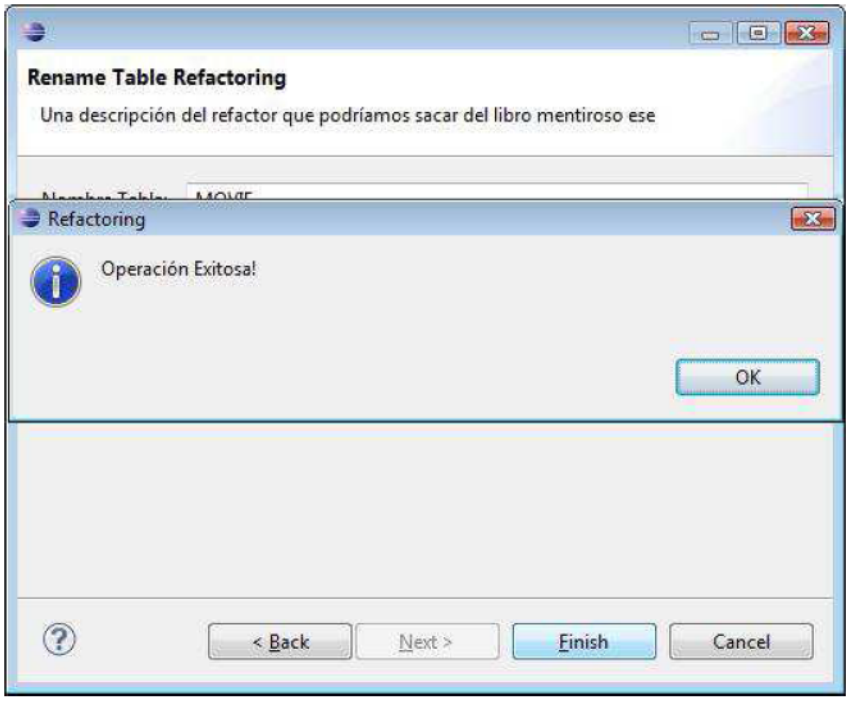


Figura 6.5: Resultado exitoso del refactoring *Rename Table*

Luego el usuario podrá inspeccionar el modelo con el refactoring aplicado:

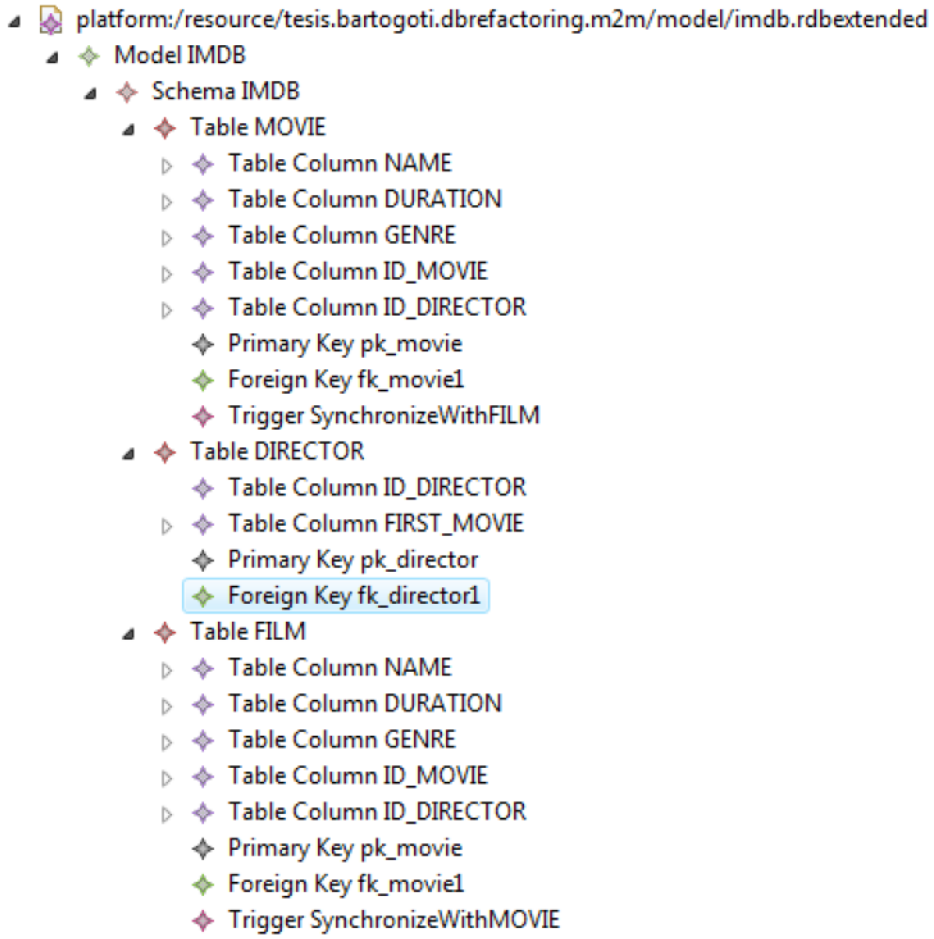


Figura 6.6: Modelo con el refactoring aplicado

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

- 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);
```

## 6.3 Arquitectura y Diseño

### *6.3.1 El Metamodelo y su Editor*

Una de las primeras y 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`.

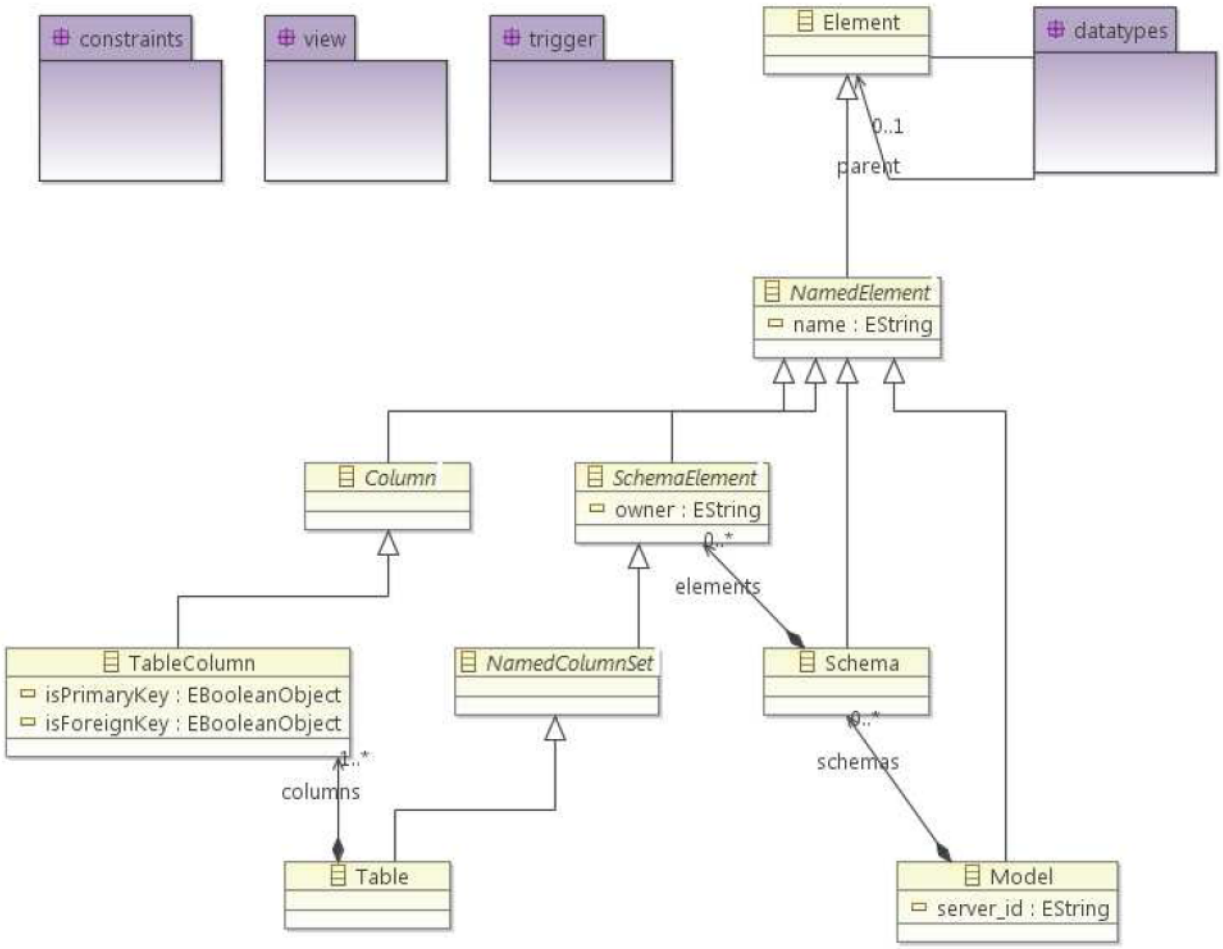


Figura 6.7: El metamodelo rdb.ecore: *Elements*

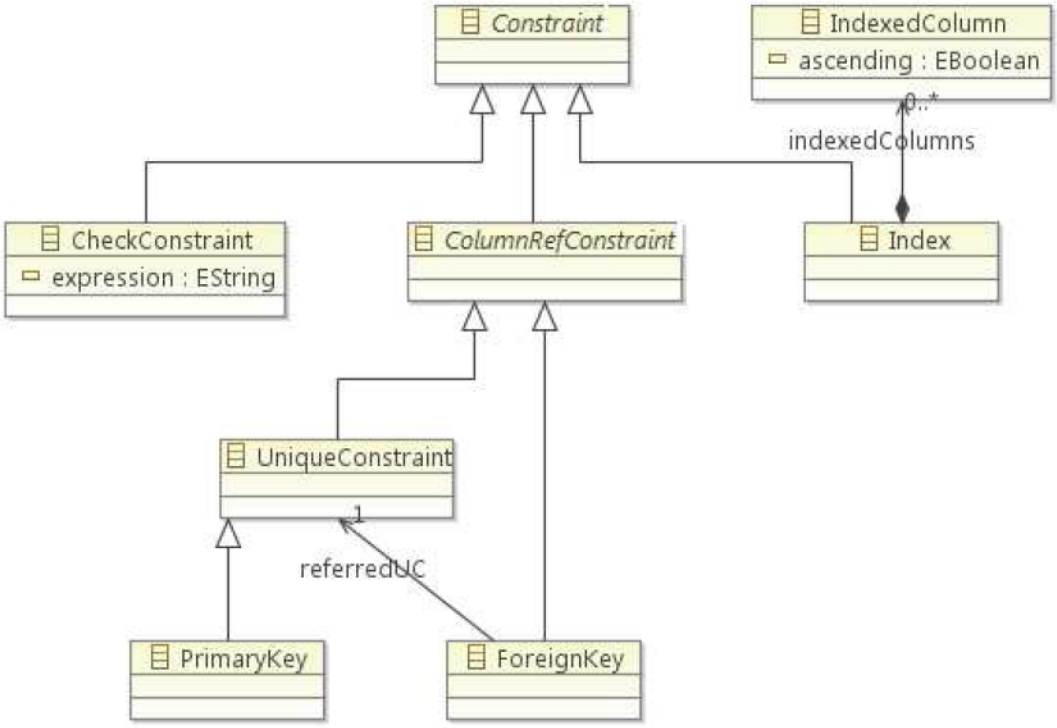


Figura 6.8: El metamodelo rdb.core: *Constraints*

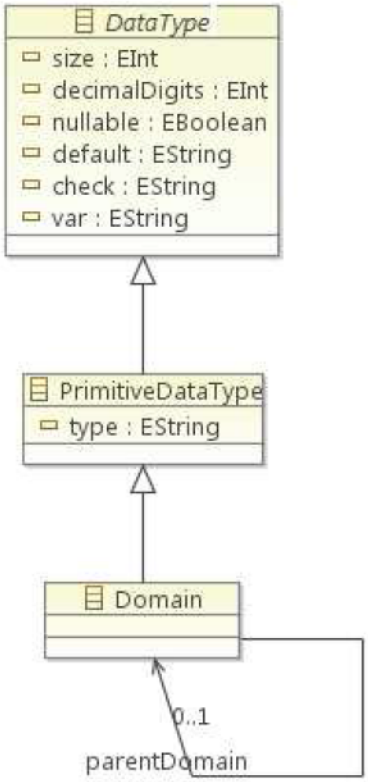


Figura 6.9: El metamodelo rdb.core: *DataTypes*

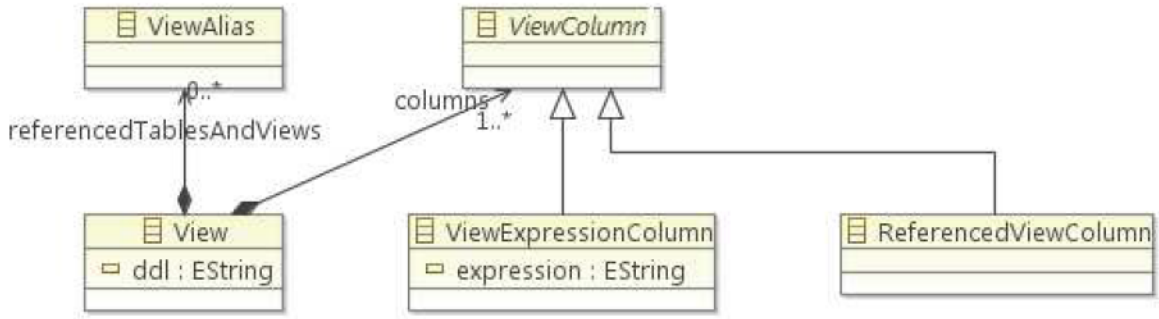


Figura 6.10: *Views*

El mismo resulta ser muy completo, y 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 propuestos en la sección 4.7, el *Trigger*. Por tal motivo, se decidió extender el metamodelo dando origen a otro metamodelo que denominamos **`rdbExtended.ecore`**, con la siguiente URI `http://rdbExtended.ecore/rdbExtended`.

Así quedó definido el elemento Trigger en el nuevo metamodelo:


Figura 6.11: El elemento Trigger

El siguiente paso consistió en poder definir un editor que permita crear instancias del metamodelo. Para esto el framework Eclipse EMF provee asistentes para crear editores simples. Así, se generan los siguientes plugins que permitirán crear instancias de `rdbExtended.ecore`:

| **Nombre** | **ID** |
|---|---|
| RdbExtended Model | rdbExtended |
| RdbExtended Edit Support | rdbExtended.edit |
| RdbExtended Editor | rdbExtended.editor |

Entonces, para acceder al editor del metamodelo, desde la ventana principal de Eclipse, través del menú de *File* → *New* → *Others*... y luego se listará RdbExtended Model, entre todos los metamodelos cargados en la plataforma.

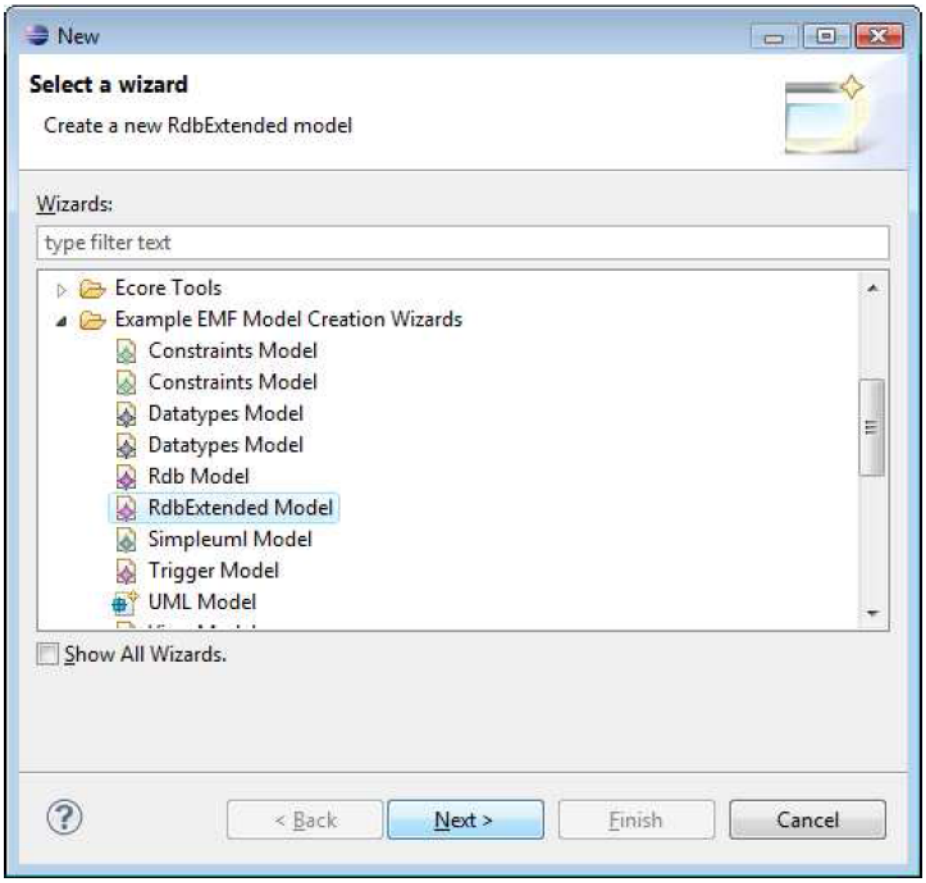


Figura 6.12: Creación de una instancia de `rdbExtended`

En el siguiente paso se indica un nombre y una ubicación para el nuevo modelo, por ejemplo `imdb.rdbExtended`:

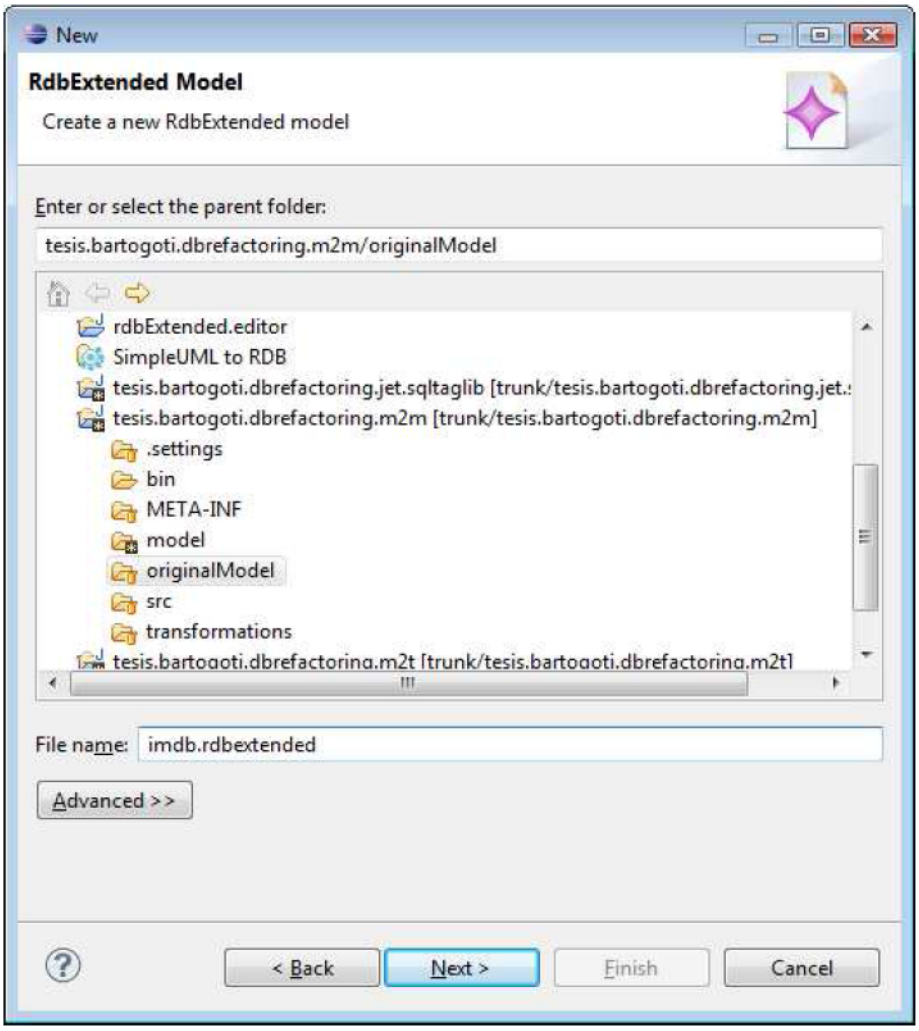


Figura 6.13: Nombre y ubicación de la instancia de `rdbExtended`

Finalmente, con el editor abierto se podrán crear nuevas instancias de las clases definidas en el metamodelo, como Tablas, Columnas y demás.

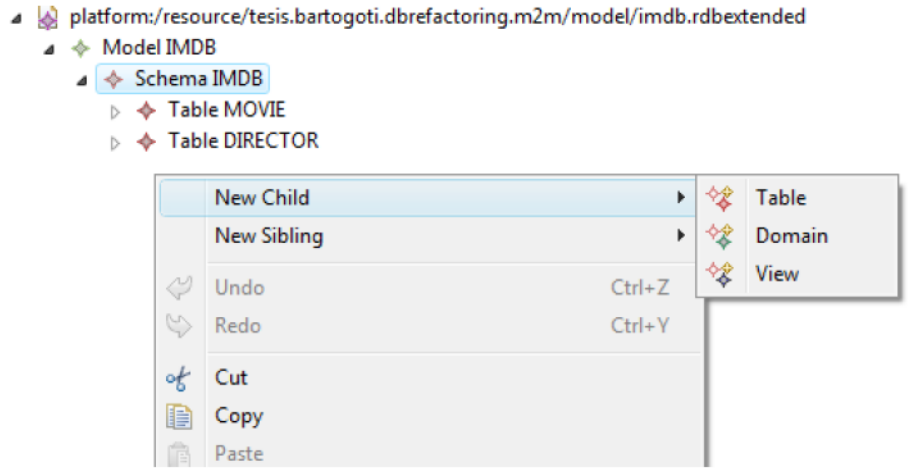


Figura 6.14: Edición de la instancia de `rdbExtended`

### 6.3.2 ESTRUCTURA BÁSICA DE LA HERRAMIENTA

La herramienta provista se compone de cuatro plugins:

| Nombre | ID | Descripción General |
|---|---|---|
| DBRefactoring UI | tesis.bartogoti.dbrefactoring.ui | Define los aspectos de interfaz de usuario, es decir, la entrada de menú y el wizard. |
| DBRefactoring M2M QVTO | 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. |
| DBRefactoring M2T JET | 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. |
| JET2 SQL TagLib | tesis.bartogoti.dbrefactoring.jet.sqltaglib | Este plugin definine una librería de tags jet para generar código SQL. Desde los templates de JET se usan tags de esta librería como <createTable ...>, <createTrigger...>, etc. |

La interacción entre los plugins es la siguiente:

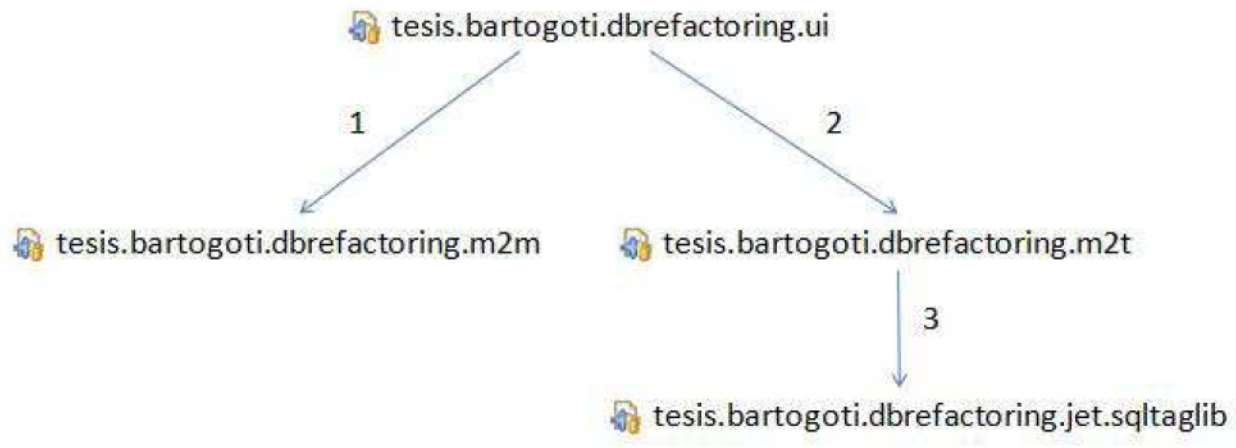


Figura 6.15: Dependencias entre plugins de la herramienta

El control principal lo lleva el plugin UI, una vez que el usuario ingresa todos los datos para el refactoring y hace click en *Finish*, se ejecuta primero la transformación de modelo a modelo definida por el plugin M2M y luego la tranformación de modelo a texto

definida en M2T. Este último hace uso de del plugin SQL Tag Lib para simplificar la escritura de código SQL.

### 6.3.3 DBREFACTORING UI

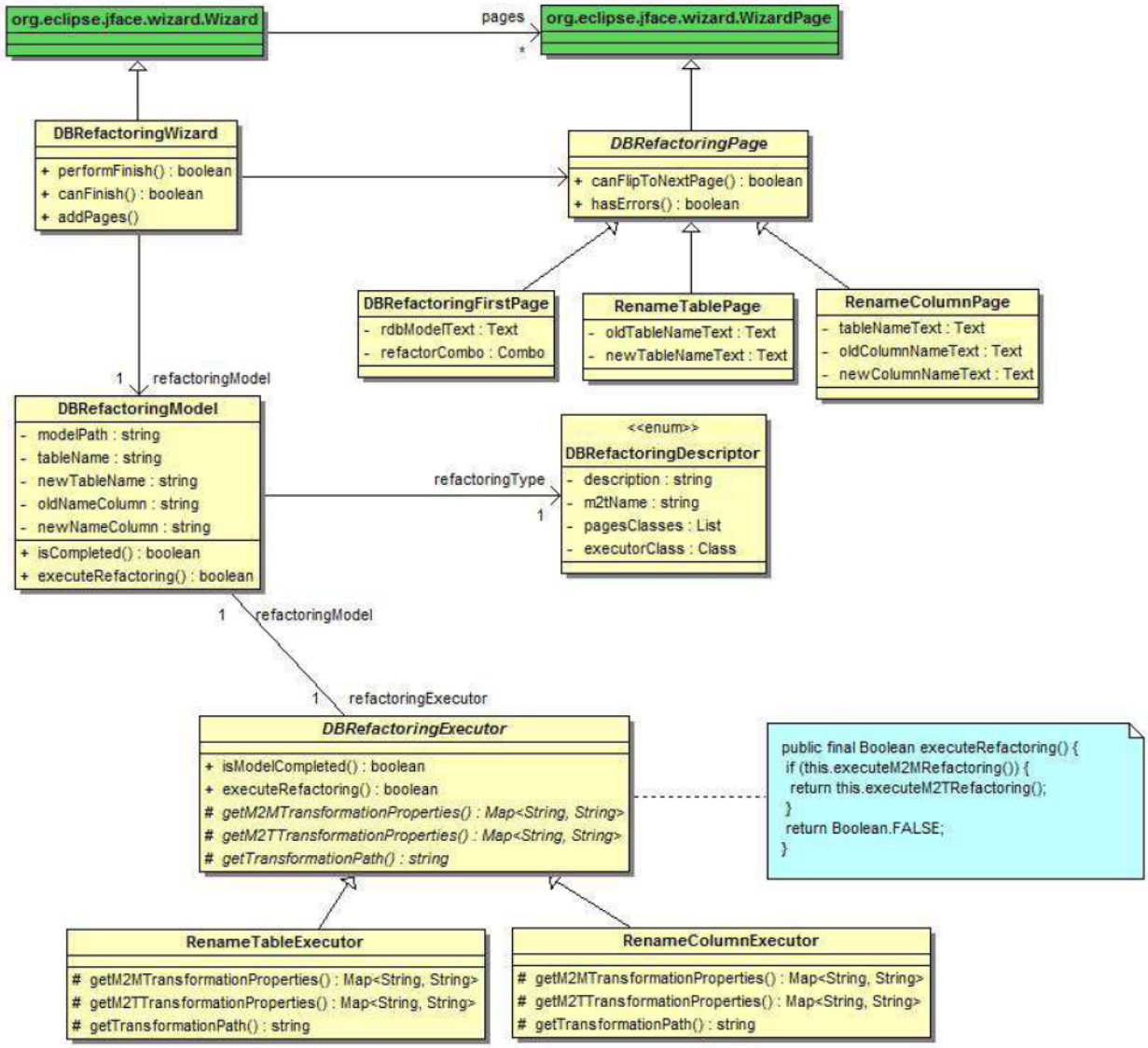


Figura 6.16: Diagrama de clases del plugin `DBRefactoirng UI`

Las clases `Wizard` y `WizardPage` son parte del framework de Eclipse.

Las clases `DBRefactoringWizard` y sus páginas, las subclases de `DBRefactoringPage`, definen la parte visual de la herramienta e implementan la interacción con el usuario y el manejo de errores. La clase `DBRefactoringModel` funciona como modelo de la vista, aquí se guardan todos los datos necesarios para la ejecución del refactoring. La clase abstracta `DBRefactoringExecutor` es la encargada de ejecutar ambas transformaciones, primero de modelo a modelo, y luego de modelo a texto, tal como se indica en la nota.

El enumerativo `DBRefactoringDescriptor` es de vital importancia. Cada refactoring que soporte la herramienta deberá tener un literal en este enumerativo. Por ejemplo para el *Rename Table* se tiene el siguiente literal:

```
RENAME_TABLE(
    "Rename Table",
    "renameTable",
    Arrays.asList(RenameTablePage.class), RenameTableExecutor.class);
```

Con esta información la herramienta puede incorporar automáticamente un nuevo refactoring sin necesidad de modificar siquiera una línea de código. Esto hace que el diseño sea escalable.

A continuación se detalla el significado de cada uno de los atributos del enumerativo.

- **`description`** (`"Rename Table"`): Es el texto que usará la herramienta para referirse a este refactoring; se usa por ejemplo para popular el combo de Refactoring en la primera página del wizard
- **`m2tName`** (`"renameTable"`): Es el nombre del template JET que resuelve la transformación de modelo a texto para generar los scripts SQL. Es decir, que en el plugin M2T debería existir un template con nombre `{m2tName}.jet`
- **`pagesClasses`** (`Arrays.asList(RenameTablePage.class)`): Es la lista de páginas que debe mostrar el wizard luego de la primera página para este refactoring. Estas páginas implementan la captura de datos necesarios para el refactoring. Normalmente debería alcanzar con una sola página, pero para permitir escalabilidad se maneja con una lista
- **`executorClass`**: Es la clase encargada de ejecutar las transformaciones una vez que el usuario hace *Finish*. Esta clase debe extender de `DBRefactoringExecutor` que es la que provee el corazón de la funcionalidad. En las subclases sólo debemos implementar operaciones sencillas como *getTransformationPath()* que indica el path del archivo de transformación *.qvto*

Entonces por cada refactoring que se quiera agregar a la herramienta, se deberá:

- Agregar un literal al enumerativo `DBRefactoringDescriptor`.
- Implementar las páginas especificas del refactoring (atributo `pagesClasses` del descriptor).
- Implementar la clase ejecutora (atributo `executorClass` del descriptor).
- Implementar la transformación QVT en el plugin M2M.
- Implementar la transformación JET en el plugin M2T.

### 6.3.4 DBREFACTORING M2M QVTO

Este plugin sirve simplemente como un contenedor de las transformaciones `.qvto`. A modo de ejemplo mostramos la transformación `renameTable.qvto`:

```
/*
 * Refactoring RenameTable:
 * Es una In-Place Transformation pues lo cambio se hacen sobre el mismo modelo
input.
 */
modeltype RDB uses 'http://rdbExtended.ecore/rdbExtended';
modeltype CTR uses 'http://rdbExtended.ecore/constraints';
modeltype TGR uses 'http://rdbExtended.ecore/trigger';

transformation rdb2rdb(inout model: RDB);

--Nombre de la tabla a renombrar, se establece en el launcher: Configutation Tab
configuration property oldTableName : String;

--Nombre nuevo de la tabla, se establece en el launcher: Configutation Tab
configuration property newTableName : String;

main() {
	--model.rootObjects()->selectOne(t |
t.oclIsTypeOf(Model)).oclAsType(Model).map model2RModel();
	model.rootObjects()[RDB::Model]->map model2Model();
}

mapping inout RDB::Model::model2Model() {
	self.schemas->map schema2Schema();
}

mapping inout RDB::Schema::schema2Schema() {
	init {
		--obtenemos la primera (uso de !) tabla de nombre oldTableName
		var oldTable :=
self.elements[Table]![name=oldTableName].oclAsType(Table);
		--La siguiente es la expresión equivalente no abreviada:
		--var oldTable := self.elements->selectOne(t | t.oclIsTypeOf(Table)
and t.oclAsType(Table).name = oldTableName);
		--creamos la nueva tabla clonandola de la vieja.
		var newTable := oldTable.deepclone().oclAsType(Table);
		--Creamos los triggers que sincronizan las ambas tablas
		var synchroWithNewTableTrigger := object Trigger { name :=
'SynchronizeWith' + newTableName };
		var synchroWithOldTableTrigger := object Trigger { name :=
'SynchronizeWith' + oldTableName };
		--Obtenemos las UCs (Unique Constraints) de la tabla vieja y la
nueva
		var ucsFromOldTable := this.getUCsFromTable(oldTable);
		var ucsFromNewTable := this.getUCsFromTable(newTable);
	}

	synchroWithNewTableTrigger.body :=
this.renameTableTriggerBody(newTableName);
	synchroWithNewTableTrigger.events +=
RDB::trigger::EventManipulationType::insert;
	synchroWithNewTableTrigger.events +=
RDB::trigger::EventManipulationType::update;

	synchroWithOldTableTrigger.body :=
this.renameTableTriggerBody(oldTableName);
	synchroWithOldTableTrigger.events +=
RDB::trigger::EventManipulationType::insert;
	synchroWithOldTableTrigger.events +=
RDB::trigger::EventManipulationType::update;
```

```
		newTable.name := newTableName;
		newTable.triggers += synchroWithOldTableTrigger;
		oldTable.triggers += synchroWithNewTableTrigger;
		self.elements += newTable;
		self.elements[Table]->map table2Table(ucsFromOldTable, ucsFromNewTable);

}

mapping inout RDB::Table::table2Table(in ucsFromOldTable:
OrderedSet(UniqueConstraint), in ucsFromNewTable: OrderedSet(UniqueConstraint)) {
		self.foreignKeys->map foreingKey2ForeingKey(ucsFromOldTable,
ucsFromNewTable);
}

mapping inout CTR::ForeignKey::foreingKey2ForeingKey(in ucsFromOldTable:
OrderedSet(UniqueConstraint), in ucsFromNewTable: OrderedSet(UniqueConstraint))
		when { ucsFromOldTable -> includes(self.referredUC) } {
		self.referredUC := ucsFromNewTable![name=self.referredUC.name];
}

helper getUCsFromTable(in table: RDB::Table): OrderedSet(CTR::UniqueConstraint) {
		return table.primaryKey.oclAsType(UniqueConstraint) -> asOrderedSet()->
			union(table.uniqueConstraints)->asOrderedSet();
}

/*
 * Retorna el cuerpo del trigger.
 */
helper renameTableTriggerBody(in tableName : String): String {
		return 'REFERENCING OLD AS OLD NEW AS NEW
		FOR EACH ROW DECLARE
		BEGIN
			IF UPDATING THEN
				findAndUpdateIfNotFoundCreate' + tableName + ';' +
			'END IF;
			IF INSERTING THEN
				createNewInto' + tableName + ';' +
			'END IF;
			IF INSERTING THEN
				deleteFrom' + tableName + ';' +
			'END IF;
		END;
		/';
}
```

### 6.3.5 DBREFACTORING M2T JET

Este plugin funciona simplemente como un contenedor de las transformaciones JET. El template principal se llama `main.jet`, el cual deriva la transformación en otro template según el refactoring solicitado, por ejemplo, `renameTable.jet`:

```
<c:setVariable select="/contents/schemas/elements" var="elements"/>
<%-- Se crea la tabla con el nuevo nombre --%>
<sql:createTable name="'{$newTableName}'">
	<%-- Se crean las columnas --%>
	<c:iterate select="$elements[@name='{$newTableName}']/columns"
var="column">
		<sql:column name="$column/@name"
			type="$column/type/@type" size="$column/type/@size"
			nullable="$column/@nullable" default="$column/@default" />
	</c:iterate>

	<%-- Se crean las Primary Key --%>
	<c:set select="$elements[@name='{$newTableName}']"
name="pkColumns"><c:iterate
select="$elements[@name='{$newTableName}']/primaryKey/includedColumns"
var="pkCol" delimiter=","><c:get select="$pkCol/@name" /></c:iterate></c:set>
	<sql:primaryKey name="$elements[@name='{$newTableName}']/primaryKey/@name"
		columns="$elements[@name='{$newTableName}']/@pkColumns" />

	<%-- Se crean las Foreign Key --%>
	<c:iterate select="$elements[@name='{$newTableName}']/foreignKeys"
var="foreignKey">
		<c:set select="$foreignKey" name="fkColumns"><c:iterate
select="$foreignKey/includedColumns" var="fkCol" delimiter=","><c:get
select="$fkCol/@name" /></c:iterate></c:set>

		<c:set select="$foreignKey" name="fkRefColumns"><c:iterate
select="$foreignKey/referredUC/includedColumns" var="refCol" delimiter=","><c:get
select="$refCol/@name" /></c:iterate></c:set>

		<sql:foreignKey name="$foreignKey/@name"
columns="$foreignKey/@fkColumns"
			referencedTable="$foreignKey/referredUC/parent/@name"
referencedColumns="$foreignKey/@fkRefColumns" />

	</c:iterate>

	<%-- Se crean las UniqueKey --%>
	<c:iterate select="$elements[@name='{$newTableName}']/uniqueConstraints"
var="uniqueKey">
		<c:set select="$uniqueKey" name="ukColumns"><c:iterate
select="$uniqueKey/includedColumns" var="ukCol" delimiter=","><c:get
select="$ukCol/@name" /></c:iterate></c:set>

		<sql:uniqueKey  name="$uniqueKey/@name"
columns="$uniqueKey/@ukColumns"/>

	</c:iterate>
</sql:createTable>

<%-- Se crean los índices en la nueva tabla --%>
<c:iterate select="$elements[@name='{$newTableName}']/indices" var="index">
```

```
      <c:set select="$index" name="indexColumns"><c:iterate
select="$index/indexedColumns" var="indexCol" delimiter=","><c:get
select="$indexCol/refColumn/@name" /></c:iterate></c:set>

      <sql:createIndex tableName="'{$newTableName}'"
columnsWithOrder="$index/@indexColumns" name="$index/@name"/>
</c:iterate>

<%-- Se crea el trigger de sincronización en la nueva tabla --%>
<sql:createTrigger name="$elements[@name='{$newTableName}']/triggers/@name"
            tableName="'{$newTableName}'"
            events="$elements[@name='{$newTableName}']/triggers/@events">

      <c:get select="$elements[@name='{$newTableName}']/triggers/@body"/>
</sql:createTrigger>

<%-- Se crea el trigger de sincronización en la vieja tabla --%>
<sql:createTrigger name="$elements[@name='{$oldTableName}']/triggers/@name"
            tableName="'{$oldTableName}'"
            events="$elements[@name='{$oldTableName}']/triggers/@events">

      <c:get select="$elements[@name='{$oldTableName}']/triggers/@body"/>
</sql:createTrigger>

<%-- Se copian los datos de la vieja tabla a la nueva tabla --%>
<sql:copyData fromTable="'{$oldTableName}'" toTable="'{$newTableName}'" />

<%-- Se hace que las FK que referenciaban a la tabla vieja apunten a la tabla
nueva --%>
<c:iterate select="$elements/foreignKeys" var="fk">
      <%-- Caso si la FK apunta a una UK de la tabla nueva --%>
      <c:iterate select="$elements[@name='{$newTableName}']/uniqueConstraints"
var="uc">
            <c:if test="($fk/referredUC/@name = $uc/@name)">

                  <sql:dropForeignKey name="$fk/@name"
tableName="$fk/parent/@name"/>

                  <c:set select="$fk" name="baseColumns"><c:iterate
select="$fk/includedColumns" var="includedCol" delimiter=","><c:get
select="$includedCol/@name" /></c:iterate></c:set>
                  <c:set select="$uc" name="referencedColumns"><c:iterate
select="$uc/includedColumns" var="includedCol" delimiter=","><c:get
select="$includedCol/@name" /></c:iterate></c:set>

                  <sql:addForeignKey name="$fk/@name"
                  baseTable="$fk/parent/@name" baseColumns="$fk/@baseColumns"
                  referencedTable="{$newTableName}"
referencedColumns="$uc/@referencedColumns"/>
            </c:if>
      </c:iterate>

      <%-- Caso si la FK apunta a la PK de la tabla nueva --%>
      <c:setVariable select="$elements[@name='{$newTableName}']/primaryKey"
var="pk"/>
      <c:if test="($fk/referredUC/@name = $pk/@name)">

                  <sql:dropForeignKey name="$fk/@name"
tableName="$fk/parent/@name"/>
```

```
                    <c:set select="$fk" name="baseColumns"><c:iterate
select="$fk/includedColumns" var="includedCol" delimiter=","><c:get
select="$includedCol/@name" /></c:iterate></c:set>
                    <c:set select="$pk" name="referencedColumns"><c:iterate
select="$pk/includedColumns" var="includedCol" delimiter=","><c:get
select="$includedCol/@name" /></c:iterate></c:set>
                    <sql:addForeignKey name="$fk/@name"
                    baseTable="$fk/parent/@name" baseColumns="$fk/@baseColumns"
                    referencedTable="'{$newTableName}'"
referencedColumns="$pk/@referencedColumns"/>
      </c:if>
</c:iterate>
```

### 6.3.6 JET2 SQL TAGLIB

Este plugin define una librería de tags jet que saben escribir código SQL. Se implementó de forma totalmente independiente del metamodelo `rdbExtended`. Su funcionalidad es muy concreta: *escribir código SQL*.

De esta forma se convierte en una abstracción muy importante dado que podría ser usado fuera del alcance de este proyecto, por cualquier transformación JET que pretenda escribir código SQL.

Los tags definidos son los siguientes:

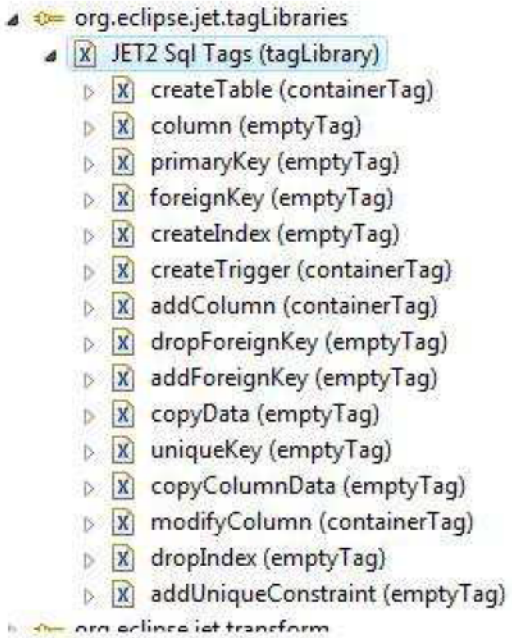


Figura 6.17: Librería de tags implementada

El siguiente es un fragmento de la estructura básica de la herramienta:

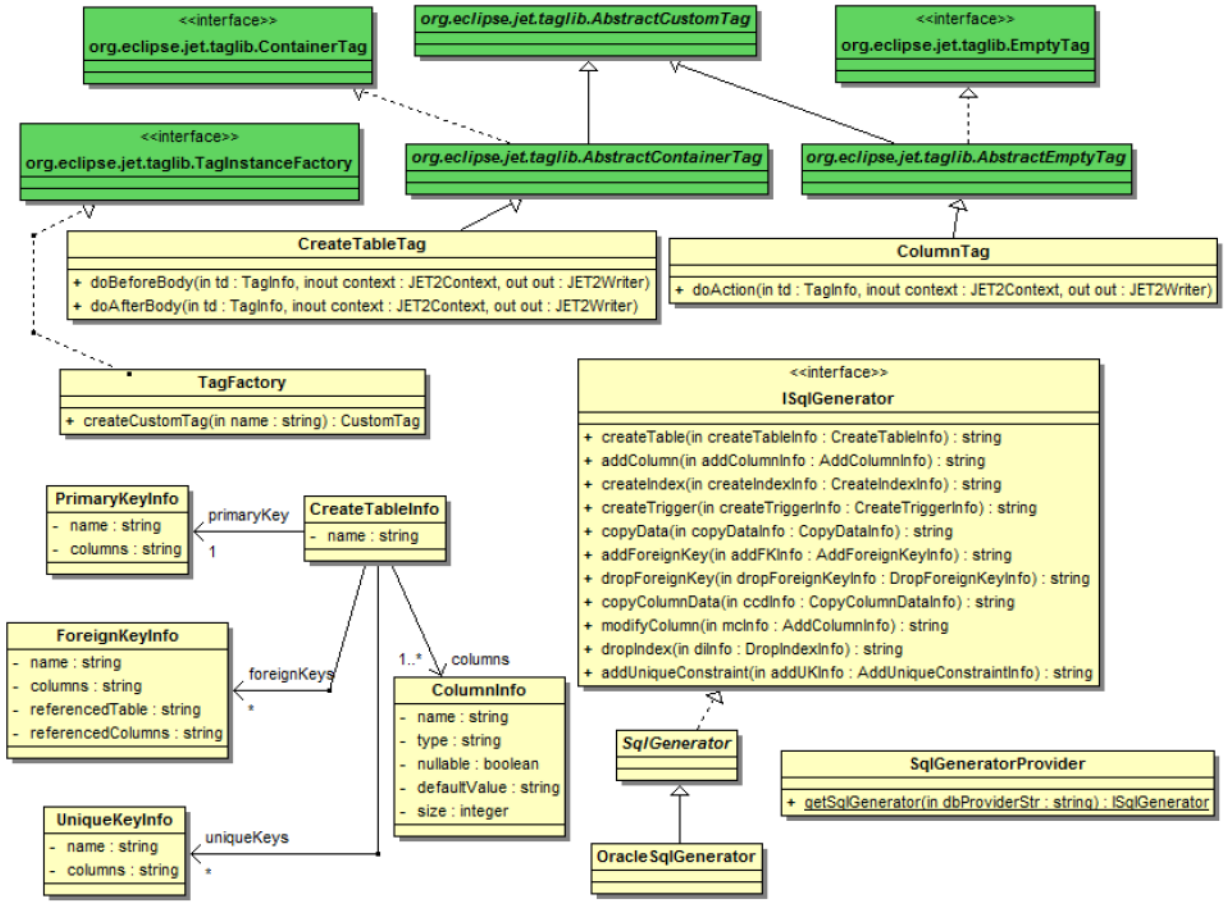


Figura 6.18: Diagrama de clases del plugin `SQL Taglib`

Las clases `ContainerTag`, `AbstractCustomTag`, `EmptyTag`, `TagInstanceFactory`, `AbstractContainerTag` y `AbstractEmtyTag` pretenecen al framework JET. La implementación de la librería consiste en extender el framework para proveer nuevos tags, en la figura se muestran sólo dos: `CreateTableTag` y `ColumnTag`. Estos tags deben resolver la escritura de código SQL. La clase `TagFactory` sirve para crear y retornar cada tag a partir de un nombre. Por ejemplo dado el nombre `createTable`, retornará una instancia de `CreateTableTag`.

El grupo de clases `XInfo`, como `CreateTableInfo` y `ColumnInfo`, colaboran con las clases de tags. Estas mantienen toda la información necesaria para crear la sentencia SQL. Por ejemplo, `ColumnInfo` tiene propiedades como `name`, `type`, `nullable`, `defaultValue` y `size`, que son los parámetros que se indican cuando se usa el tag.

Por otro lado tenemos la interface `ISqlGenerator` con su implementación abstracta `SqlGenerator` y una implementación concreta `OracleSqlGenerator`. Esta última es la que en definitiva escribe código SQL para Oracle. Con esta estructura, el framework puede fácilmente extenderse para generar código para otros proveedores de bases de datos. Por ejemplo, podríamos agregar la clase hermana `MySQLGenerator`.

A continuación el fragmento de código de `CreateTableTag` encargado de escribir código SQL:

```
String dbProvider = (String) context.getVariable("db.provider");
ISqlGenerator sqlGenerator = SqlGeneratorProvider
```

```
            .getSqlGenerator(dbProvider);
out.write(sqlGenerator.createTable(ctInfo));
```

La clase `SqlGeneratorProvider` sirve para seleccionar el generador apropiado según el parámetro `db.provider`. No se mencionarán más detalles técnicos sobre la implementación porque escapa un poco al objetivo de este trabajo, pero cabe destacar que el diseño es simple y escalable. Se podrían agregar nuevos tags a la librería sin mucho esfuerzo. Así como también queda abierta para proveer implementaciones para distintos proveedores de bases de datos.

## 6.4 A FUTURO

Con este desarrollo, se pretende principalmente mostrar que es posible automatizar las tareas de refactorings de bases de datos propuestas en [AS 06]. Sin embargo, se debería seguir trabajando sobre la herramienta si se pretende integrar a un proyecto de desarrollo real.

A continuación enumeramos algunas propuestas de mejora:

- Actualmente la herramienta implementa cinco refactorings: *RenameTable, RenameColumn, RenameView, ReplaceOneToManyWithAssociativeTable* y *makeColumnNonNullable*. Es deseable extender la herramienta con más implementaciones de refactorings de base de datos.
- La herramienta por el momento sólo genera código para ORACLE. Sería provechoso brindarle al usuario la posibilidad de generar código para distintos proveedores de bases de datos, al menos los más populares del mercado. Como explicamos en la sección anterior, el plugin JET2 SQL TagLib está diseñado para que fácilmente puedan agregarse distintas implementaciones para distintos motores de bases de datos.
- Otro punto a mejorar puede ser la usabilidad. Si bien la UI es simple e intuitiva, podría ser más amigable para el usuario. Por ejemplo, para el *Rename Table* se podría ofrecer un combo para que el usuario seleccione la tabla a renombrar.
- Actualmente se utilizan editores básicos EMF para trabajar con modelos de bases de datos, instancias de `rdbExtended`. Se podría hacer uso de GMF para generar editores gráficos de manera que sea más amigable trabajar con los modelo de bases de datos.
- Hoy la herramienta sólo resuelve la primera etapa del refactoring. Sería de gran utilidad extender la misma de manera que sirva para completar la implementación del refactoring una vez cumplido el período de transición.

# 7. Trabajos Relacionados

Existen varias herramientas que implementan la propuesta de Refactoring de Bases de datos. En su mayoría, la motivación principal de las mismas es proveer un mecanismo para aplicar en la base de datos cambios (o refactorings), ya sea en datos o en estructura, mediante un archivo externo que puede ser un archivo con un script SQL, código Java o XML. Todos estos cambios o refactorings en la base de datos, se aplicarán para una determinada versión de la base de datos. Este es el punto en que estas herramientas juegan un papel fundamental. Serán las encargadas de correr los archivos correspondientes a la versión actual y registrar que la base de datos está en una versión determinada. De esta forma, será muy simple conocer la versión actual de la base de datos. Llevar un control de versiones sobre la base de datos es indispensable, sobre todo cuando se cuenta con varios ambientes por los cuales cada versión de la base de datos será instalada: desarrollo, testing, preproducción y finalmente producción.

Si bien la herramienta propuesta en este trabajo se enfoca en la generación de código SQL, el uso de estas herramientas es imprescindible para completar el proceso de refactoring de base de datos. El objetivo principal de estas herramientas es facilitar a un equipo que trabaja bajo una metodología ágil las etapas del proceso de refactoring de bases de datos, principalmente dando soporte al control de versiones en los distintos ambientes que de la base de datos. La herramienta propuesta en este trabajo abarca otra etapa dentro del proceso, la generación de código SQL que implementará el refactoring, automatizando una parte fundamental del proceso de refactoring de bases de datos. De este modo, el uso de la herramienta propuesta genera un complemento ideal para las herramientas analizadas que acompañan al proceso de refactoring de bases de datos.

A continuación se presenta un breve análisis de las principales herramientas.

**Liquibase**

Es una herramienta para la gestión de cambios en la base de datos. Propone un control de versiones para la base de datos al igual que cuando se desarrolla código. Se basa en la premisa que todos los cambios de base de datos se almacenan en una forma legible, rastreable y protegida bajo un control de versiones. Los cambios o refactorings se escriben en un formato XML, definido por la herramienta. Esta es una de las características más importantes, ya que este XML puede verse como un DDL (Data Definition Language) abstracto, independiente de cada motor de base de datos, o mejor aún, como un DSL (Domain Specific Language) para modificaciones a una base de datos.

A partir de este XML genera el código SQL para múltiples motores de base de datos, lo ejecuta y lleva un control de la versión de la base de datos. Además, dentro de cada XML se pueden incluir archivos SQL o Java para implementar un cambio no provisto por la herramienta. También provee soporte para hacer rollback a una determinada versión.

Además posee integración con herramientas Maven o Ant, de modo que cada refactoring puede aplicarse cuando hace el build de la aplicación.

Solamente puede integrarse con aplicaciones desarrollada en Java.

### Flyway

Se define como una herramienta para migración de bases de datos en un ambiente Java. Su objetivo es llevar un control de versiones en la base de datos. En cada ambiente de la base de datos crea una tabla donde lleva registro de la versión de la base de datos. Cada migración, puede estar implementada en un archivo SQL o en una clase Java. Para esto Flyway posee una API para facilitar la implementación de la misma. También posee soporte para los principales motores de bases de datos actuales.

### DBdeploy

Es una herramienta similar a Flyway, lleva un control de la versión de la base de datos. Permite correr archivos SQL y tiene soporte para varios motores de bases de datos. Se integra al proceso puede usar en ambientes Java o .NET.

### Otras herramientas

Otras herramientas como `migrate4j`, `migratedb` y `dbmaintain` trabajan bajo la misma idea, pero son más limitadas en funcionalidad que las anteriores. Básicamente proveen una forma de ejecutar cambios en la base de datos, ya sean archivos SQL o escribir estos cambios en un archivo XLM (como Liquibase, pero más limitado). Este último caso es la propuesta de `migratedb`.

A continuación se muestran las diferencias entre las mencionadas herramientas:

| | **Liquibase** | **Flyway** | **DBdeploy** | **Migrate4j** | **Migratedb** | **DBmaintain** |
|---|---|---|---|---|---|---|
| **Formatos de Entrada** | | | | | | |
| SQL | (1) | ✓ | ✓ | | | ✓ |
| Java | (1) | ✓ | | ✓ | | |
| Groovy | ✓ | | | | | |
| XML | ✓ | | | | ✓ | |
| DDL abstraction (DSL) | ✓ | | | | | |
| **Ejecución** | | | | | | |
| API Java | ✓ | ✓ | | ✓ | | ✓ |
| Maven plugin | ✓ | ✓ | | | | ✓ |
| Ant Task | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Línea de comandos | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

| **Bases de Datos** | | | | | | |
|---|---|---|---|---|---|---|
| Oracle | ✓ | ✓ | ✓ | | ✓ | ✓ |
| SQL Server | ✓ | ✓ | ✓ | | ✓ | ✓ |
| DB2 | ✓ | ✓ | ✓ | | ✓ | ✓ |
| MySQL | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| PostgreSQL | ✓ | ✓ | ✓ | | ✓ | ✓ |
| H2 | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Hsql | ✓ | ✓ | ✓ | | ✓ | ✓ |
| Derby | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Otras Características** | | | | | | |
| Creación de metadata para tablas | ✓ | ✓ | ✓ | | | ✓ |
| Seguridad entre clusters | ✓ | ✓ | | | | |
| Validación por checksum | ✓ | ✓ | | | | ✓ |
| Remplazo de variables (placeholders) | ✓ | ✓ | ✓ | | | |
| Soporte para múltiples esquemas | | ✓ | | | | ✓ |
| Limpiar esquemas existentes | | ✓ | | | | ✓ |
| Salida de SQL | ✓ | | ✓ | | ✓ | |
| Disponible en repositorio Maven Central | ✓ | ✓ | | | | ✓ |
| **Licencia** | Apache v2 | Apache v2 | BSD | LGPL v3 | BSD | Apache v2 |

(1) Archivos SQL y clases Java se puede utilizar indirectamente a través de referencias en las migraciones XML

# 8. Conclusiones

En el presente trabajo se hizo un importante énfasis en seguir un enfoque evolutivo para el desarrollo de la base de datos, al igual que lo siguen todos los equipos modernos de desarrollo de aplicaciones.

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, se enfatiza en el refactoring, una forma disciplinada de reestructurar el código en pequeños pasos evolutivos para mejorar la calidad del diseño del mismo. El refactoring es una pieza fundamental en un proceso de desarrollo evolutivo. Un refactoring de código mantiene el comportamiento semántico del código, no agrega ni elimina funcionalidad del mismo. Similarmente un refactoring de bases de datos es un cambio simple al esquema de la base de datos que mejora su diseño mientras mantiene la semántica de comportamiento e información. El refactoring de bases de datos es una técnica fundamental que permite a los profesionales de bases de datos tomar un enfoque evolutivo para el desarrollo de bases de datos.

Sin embargo, se deben tener en cuenta varios aspectos antes de aplicar un refactoring a la base de datos. Uno de ellos es el acoplamiento. Cuanto más acoplada esté la base de datos con el resto de los componentes de software de la organización, más dificultoso será aplicar refactoring, desde el punto de vista que habrá que actualizar el resto de los sistemas a esta nueva versión de la base de datos.

Por este motivo, cuando se determina que hay que aplicar un refactoring, es importante evaluar el costo del refactoring al momento de aplicarlo. Tal vez el costo de aplicar el refactoring sobrepasa el beneficio, o tal vez el esquema ya tiene el mejor diseño. En caso que se decida aplicar, es necesario soportar la versión original y la nueva versión del esquema luego de haber aplicado el refactoring por un periodo de transición lo suficientemente largo para permitir el deployment (o despliegue) en producción a todas las aplicaciones que acceden a esa porción del esquema.

Para implementar el refactoring, es altamente recomendable tomar un enfoque Test Driven Development para detectar cualquier comportamiento cambiado durante el proceso. De esta forma, cuando se detecta mediante un test que algo no está funcionando como antes, será sencillo detectar que modificación afecto ese comportamiento, debido a que cada refactoring implica sólo pequeños cambios.

En el punto de la implementación del refactoring nos centramos para realizar un aporte en una metodología evolutiva para el desarrollo de bases de datos. Para facilitar el proceso de de aplicar un refactoring al a base de datos, se propone una herramienta extensible que genera el código DDL, DML y SQL necesario para implementar un refactoring, manteniendo la semántica del esquema de bases de datos.

La herramienta es un complemento ideal para las herramientas mencionadas en el *Capítulo 7*, que aportan en la organización del proceso evolutivo con tareas como control de versiones, ejecución de scripts, etc.

La herramienta trabaja con un modelo lógico 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, genera 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 se utilizó 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 se propone el metamodelo `rdbExtended`. El metamodelo describe la sintaxis abstracta de un esquema de bases de datos y constituye la base para el procesamiento automatizado de los modelos.

De esta forma, 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.

# REFERENCIAS BIBLIOGRÁFICAS


[AA 01a] Manifesto for Agile Software Development, http://agilemanifesto.org/, 2001

[AA 01b] Principles: The Agile Alliance, http://agilemanifesto.org/principles.html, 2001

[AM 02] Ambler, Scott W. Agile Modeling: Best Practices for the Unified Process and Extreme programming. New York: John Wiley & Sons, 2002

[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

[CESW 04] Tony Clark, Andy Evans, Paul Sammut, James Willans. Applied Metamodelling. A Foundation for Language Driven Development. http://www.ceteva.com/book.html, 2004

[CR 08] Eric Clayberg, Dan Rubel. Eclipse Plugins, Third Edition. The Eclipse Series, Addison Wesley, 2008

[CWM 03] Common Warehouse Metamodel™ (CWM™) Specification, v1.1 http://www.omg.org/cgi-bin/doc?formal/03-03-02

[DBdeploy] DBdeploy http://dbdeploy.com/

[DBmaintain] DBmaintain http://www.dbmaintain.org/

[DM 79] De Marco, Tom. Structured Analysis and System Specification. Englewood Cliffs, NJ, 1979

[EC] Eclipse http://www.eclipse.org/

[EC 03] Eclipse Platform Technical Overview, Object Technology International, Inc., Febrero 2003: http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[EC PR] Eclipse projects http://www.eclipse.org/projects/

[EMF] EMF http://www.eclipse.org/projects/project.php?id=modeling.emf

[EMP] Eclipse Modeling Project. http://www.eclipse.org/modeling/

[Flyway] Flyway http://flywaydb.org/

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

[FO 03] Martin Fowler, P. Sadalage. Evolutionary Database Design. http://martinfowler.com/articles/evodb.html, 2003

[GR 09] Gronback, Richard C. Eclipse Modeling Project: A Domain Specific Language (DSLI) Toolkit. The Eclipse Series, Addison Wesley, 1999

[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

[Liquibase] Liquibase http://www.liquibase.org/

[M2M] Eclipse Model to Model (M2M) transformation. http://www.eclipse.org/m2m

[M2T] Eclipse Model to Text (M2T) transformation. http://www.eclipse.org/m2t

[Migrate4j] Migrate4j http://migrate4j.sourceforge.net/

[Migratedb] Migratedb http://migratedb.sourceforge.net/

[MOF] Object Management Group. "OMG's Meta Object Facility". http://www.omg.org/mof/

[PDE] Eclipse Plugin Development Environment http://www.eclipse.org/projects/project.php?id=eclipse.pde

[PGP 08] Claudia Pons, Roxana Giandini, Gabriela Pérez. Desarrollo de Software Dirigido por Modelos. Facultad de Informática, Universidad Nacional de La Plata. La Plata, 2008

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

[QVTo] QVT Operational http://www.eclipse.org/projects/project.php?id=modeling.mmt.qvt-oml

[QVTR] Query/View/Transformation Request for Proposal, OMG, April 2002