Desarrollo de un Mecanismo de Control para el Acceso a Datos

Ingeniería en Computación

Daniel García
Pablo Gagliardi
Beatriz Pereira
Andrea Coronado

Supervisor:
Daniel Calegari
Usuario Responsable:
Álvaro Rodriguez

21 de julio de 2009

Daniel García	dgarcia@paytrue.com
Pablo Gagliardi	pablo.javier.gagliardi@gmail.com
Beatriz Pereira	bpereira@paytrue.com
Andrea Coronado	andreacoro@gmail.com

Creado utilizando LaTeX 2_ε y Microsoft Visio 2007

Revision : 19

Resumen

La programación orientada a objetos es actualmente el paradigma de desarrollo más extendido para el desarrollo de sistemas de información. Al mismo tiempo las bases de datos relacionales continúan siendo, por diversos motivos, el medio de almacenamiento de información por excelencia. En este contexto, la conexión entre la lógica del sistema y el manejador de base de datos presenta un conflicto de paradigmas que ha dado lugar a una gran cantidad de propuestas para su resolución.

La empresa *PayTrue Solutions* , cliente del presente trabajo, desarrolla sistemas de información complejos, usualmente de porte mediano a grande. En lo que respecta a la tecnología de acceso a datos, inicialmente utilizó una herramienta comercial de generación de código para mapeo entidad-relación (LLBLGenPro). Esta herramienta presenta ciertas características que se transformaron en desventajas insostenibles por lo cual su uso se discontinuó. En su lugar se comenzó a utilizar un mecanismo de acceso directo a la base de datos (ADO.Net) con algunas convenciones de programación orientadas a obtener control de correctitud en tiempo de compilación. Esta estrategia también presenta desventajas, en particular que la programación de consultas resulta engorrosa y poco legible.

Los argumentos expuestos motivaron el desarrollo del presente trabajo. En primer lugar se realizó un estudio de los paradigmas y herramientas disponibles en el mercado. Estas herramientas no cumplieron con los objetivos planteados, además de que el tiempo requerido para su adopción no las hacía viables. A raíz de esto se desarrolló un conjunto de herramientas que cumplen con los objetivos planteados, entre ellos: simplificar el acceso a datos, permitir la verificación de la correctitud de las consultas por construcción, realizar un análisis de impacto en las consultas existentes de un cambio en la base de datos y proveer un analizador sistemático de consultas para validar posibles errores que estas pueden producir en tiempo de ejecución.

Índice general

Capítulo 1

Introducción

Este proyecto tiene como cliente a la empresa *PayTrue Solutions* [1], la cual se especializa en el desarrollo de aplicaciones para medios de pago. Las aplicaciones que desarrolla, en su mayoría, son sistemas complejos de gestión y procesamiento de datos, de mediano a gran porte, basados en una arquitectura fuertemente dependiente del modelo de datos. Por esto es importante que se tenga un buen diseño de la capa de acceso a datos y un manejador óptimo de dicha capa.

El desarrollo en *PayTrue Solutions* [1] sigue una metodología modular, permitiendo de esta forma el desarrollo en paralelo del software y la reutilización de componentes. Las aplicaciones resultan principalmente de la composición de una serie específica de módulos, junto a un servidor especialmente desarrollado para resolver la interacción entre los mismos y con los clientes. Además de la separación en módulos se utiliza un diseño basado en capas. Las mismas son visibles también desde la óptica de los módulos, ya que cada uno dispone de la porción que le corresponde, inclusive del modelo de datos.

La capa de acceso a datos (de aquí en adelante DAL por sus siglas en inglés) constituye uno de los aspectos más importantes en el desarrollo de software de *PayTrue Solutions* [1], dado que sus aplicaciones son de alta complejidad requieren de un acceso a datos flexible y performante, teniendo que realizar consultas realmente complejas sobre la base de datos. La funcionalidad principal de la DAL es ofrecer a

las demás capas acceso a los datos que se encuentran almacenados en la base de datos.

En el presente documento nos enfocaremos en dicha capa, considerando en particular su aplicación sobre bases de datos relacionales.

1.1. Problemática del cliente y soluciones planteadas

A continuación se presentan los problemas que se han identificado en el cliente:

- Complejo mantenimiento de la DAL
  El mantenimiento de la DAL se planteó como uno de los principales problemas del cliente. Incluir una nueva consulta en los sistemas de cliente implica un costo en horas/hombre relativamente alto, así como también la modificación de consultas existentes.

- Código de la DAL heterogéneo
  La forma de desarrollo de la DAL que utiliza el cliente puede provocar que el código de la misma termine siendo heterogéneo. Esto sucede especialmente en proyectos de gran porte donde participan decenas de desarrolladores, estas condiciones propician la gradual degradación estructural del código de la DAL.

- Costoso análisis de impacto
  El cliente ha planteado la problemática de que en etapas maduras de los proyectos que encara, le es altamente costoso efectuar un análisis de impacto de cambios en el modelo de datos.

- Disociación entre la DAL y el modelo de datos
  El modelo de datos, mantenido por el cliente con la herramienta QDesigner [5], no tiene ningún punto de contacto tangible con la DAL, esto implica que puedan ser escritas consultas que referencien a objetos erróneos de la base de datos.

La problemática descripta líneas más arriba, acarrea la necesidad de contar con una herramienta que englobe varias características fundamentales:

- Generación automatizada del código correspondiente a la lógica de acceso a datos.
- Verificación de las consultas en tiempo de compilación.
- Homogeneizar el código generado de modo que sea legible y de fácil interpretación.
- Disponibilizar toda la información del modelo de datos y sus consultas en una ubicación centralizada de forma de facilitar su uso.

Es por esto que se hace cada vez más notoria la necesidad de contar con dicha herramienta hecha a medida que cubra todas las necesidades y requisitos de la empresa, de modo de encontrar de un mecanismo de trabajo que permita optimizar el proceso de desarrollo.

1.2. Objetivos

El objetivo principal de nuestro proyecto es desarrollar un mecanismo de control para el acceso a datos de la plataforma de desarrollo de *PayTrue Solutions* [1], de forma tal de alcanzar una serie de objetivos que apuntan a facilitar la programación y el mantenimiento del software.

Entre las diversas tecnologías de acceso a datos, la escogida inicialmente por *PayTrue Solutions* [1] fue el mapeo entidad-relación. Se utilizó una herramienta comercial de generación de código para mapeo entidad-relación llamada LLBL GenPro [7] . Esta herramienta presenta ciertas características que se transformaron eventualmente en desventajas insostenibles por lo cual su uso se discontinuó, pasando a utilizar directamente ADO.NET [11] con algunas convenciones de programación orientadas a obtener control de correctitud en tiempo de compilación en cuanto a los nombres de los objetos de la base de datos. Esta estrategia tiene también varias desventajas,

en particular que la programación de consultas resulta engorrosa y poco legible.

Es aquí donde se ve la importancia de la contribución de nuestro proyecto, en lograr tener todas las necesidades cubiertas en una única herramienta hecha a medida.

Para lograr el desarrollo se marcaron los siguientes objetivos:

- Generación de código

  A partir de consultas en lenguaje SQL, lograr la generación de código de consultas, logrando homogeneizar el código generado, de modo que este sea legible para facilitar su mantenimiento. Las consultas soportadas permiten que se puedan utilizar todas las funcionalidades que brinda Oracle [2], según la funcionalidad actual de *PayTrue Solutions* [1].

- Generación de metadata del modelo de datos y de consultas

  Permitiendo consultar las características de cualquier objeto en un momento dado. Esta metadata contará con la información de nombres de consultas, objetos de la base de datos utilizados en cada consulta y ubicación dentro del código generado de cada una de las consultas.

- Aplicación de reglas

  Proveer de un manejador de reglas que permita agregar condiciones extra a las consultas para realizar filtrados según distintos criterios. Al mismo tiempo, las reglas permiten la inyección de secciones de código a las consultas en forma global o puntual, antes y/o después de la ejecución de las mismas.

- Verificación de correctitud de consultas

  Verificación semántica y sintáctica de consultas, previo a la generación de código, logrando así que no se genere el código de consultas, que posteriormente puedan dar errores al ejecutarse.

- Análisis de impacto

  Poder consultar los usos de un determinado objeto de la base de datos, de modo que se desplieguen todas los usos del mismo. Esto permite que al hacer cualquier cambio que lo involucre, se

sepa de antemano las posibles consecuencias de dicho cambio. Esto es posible a partir de la metadata de código generada, que se podrá consultar a través de un manejador de base de datos.

- Analizador sistemático de consultas

  Lograr que se pueda hacer una verificación de todas las consultas de la DAL contra una base de datos (cuyo esquema potencialmente difiera del vigente dentro del código) a los efectos de saber fácil y rápidamente qué consultas son incorrectas contra el nuevo esquema.

1.3. Estructura del Documento

Este informe se organiza en 4 capítulos, en los cuales se presenta la información más relevante del proyecto. Los siguientes capítulos se estructuran como sigue:

- El capítulo 1 presenta la introducción del proyecto.
- El capítulo 2 describe y analiza la problemática a atacar.
- El capítulo 3 profundiza en la herramienta desarrollada, brindando una visión general y describiendo los componentes *Generador*, *Analizador de Impacto* y *Verificador sistemático de consultas*.
- El capítulo 4 presenta un caso de estudio.
- El capítulo 5 presenta las conclusiones y trabajo a futuro.

Por último se adjunta varios apéndices, los cuales describirán la arquitectura de las herramientas desarrolladas, la estructura de la metadata utilizada, una ampliación de la verificación sistemática de consultas, las sentencias DML soportadas, descripción del plan de verificación y testing, el proceso de desarrollo del producto y glosario. Finalmente se presenta la bibliografía utilizada.

Capítulo 2

Análisis de la problemática

2.1. Definición de Data Access Layer

La DAL es la capa de software que provee los servicios necesarios para persistir y acceder a datos almacenados en algún tipo de medio, tal como bases de datos relacionales o archivos XML. Uno de los principales objetivos de esta capa es ocultar las complejidades intrínsecas del acceso a datos a las capas superiores del software. Por ejemplo, la DAL puede retornar un objeto conteniendo las filas y atributos de una tabla de la base de datos, abstrayendo de esta manera la forma en la cual se almacena la información y los mecanismos necesarios para acceder a la misma. Es importante que los objetos propios de la capa de negocio no estén acoplados de ninguna forma al medio de almacenamiento.

Otro objetivo de la DAL es encapsular la lógica de acceso a la base de datos en un único lugar del sistema. Tal como se mencionó líneas más arriba, la DAL puede interactuar con diversos medios de persistencia de datos, pero dada la naturaleza de este proyecto, vamos a enfocarnos únicamente en motores de base de datos relacionales.

2.2. Implementación de la DAL

Existen diversas alternativas al momento de implementar la DAL y la decisión de cuál seleccionar no solo afecta la tarea de desarrollo

de software sino también su mantenimiento. Al momento de optar por una forma de desarrollo se presenta un diverso abanico de posibilidades, tanto a nivel de paradigma como a nivel de herramientas disponibles en el mercado.

2.2.1. Paradigmas de desarrollo de la DAL

El desarrollo de la DAL puede encararse de varias formas. Esta multiplicidad de opciones puede agruparse en unas pocas clases de equivalencia o paradigmas de desarrollo que presentaremos en este capítulo [14].

Acceso directo a base de datos

La forma más elemental de acceso a la base de datos es utilizando directamente SQL en el código de la DAL. Las sentencias SQL son tratadas simplemente como strings y enviadas al proveedor de conexión con la base de datos. No existe ningún tipo de *middleware* entre la aplicación y el *DBMS* utilizado.
A continuación presentamos las características más importantes de este paradigma.

- Performance de las consultas
  El uso directo de SQL le brinda al desarrollador la ventaja de poder controlar el formato de sus consultas. Esto tiene como principal beneficio el poder dar a la consulta el formato más óptimo desde el punto de vista de la performance.

- Las consultas no son validadas en tiempo de compilación
  No existe ninguna validación al momento de compilar, las consultas no son más que simples strings. Pueden presentar errores sintácticos o acceder a objetos erróneos de la base, estos problemas se detectarán recién al momento de que la consulta sea ejecutada desde la DAL. Pueden aplicarse algunas buenas prácticas de programación para minimizar los aspectos negativos recién descriptos, tal como el uso de enumerados para

representar ciertos objetos de la base (tablas, columnas y secuenciadores) concatenándolos con el cuerpo de la consulta, de esta forma se minimizan los posibles errores de referencia a objetos erróneos. En contraposición, se disminuye de gran manera la legibilidad de la consulta, dado que el string ya no es simple, sino que debe ser armado concatenando los valores correspondientes.

- Puede generarse un fuerte acoplamiento entre la DAL y el DBMS
  A pesar de ser un estándar ANSI, los fabricantes de DBMS suelen hacer modificaciones sobre el SQL que exponen sus bases de datos, además de agregar extensiones que en muchos casos es conveniente utilizar para simplificar las consultas o mejorar la performance (ej. hints y decode de Oracle [2], top N de Microsoft SQL Server [3]). Este fenómeno, suele generar dependencia entre la DAL y el DBMS, cosa que dificulta la migración del sistema a otro DBMS.

- El mantenimiento del código resulta engorroso
  Cambios en el esquema de datos suelen resultar costosos de reflejar en el código de la DAL, tanto en el esquema como en las consultas. Por ejemplo, si se ha optado por manejar enumerados, en el caso del esquema, hay que actualizarlos para reflejar los cambios. Luego actualizar todas las consultas que utilicen los campos modificados. En caso de que se desee modificar una consulta, de no seguir estándares a la hora de generarlas, resulta muy complejo realizar modificaciones si pertenecen a otros usuarios, así como también modificar el string de la consulta sin poder apreciarlo claramente. Para ello sería necesario hacer una traducción para poder ver la consulta en SQL puro, de modo de poder testear la modificación, y luego de validada, se actualizan las porciones del script que correspondan, lo cual da lugar a errores por parte del usuario.

Mapeo objeto-relacional

El mapeo objeto-relacional (Object-Relational mapping en inglés, ORM) es una técnica de programación que permite convertir datos

entre el sistema de tipo utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional. Desde el punto de vista conceptual, esta técnica de programación genera una base de datos orientada a objetos virtual sobre la base de datos relacional, esto permite el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). El objetivo es mitigar el problema conocido como *impedance mismatch*, el cual refiere a la desconexión entre los modelos relacional y orientado a objetos [18]. Existen diversos ORMs en el mercado: Hibernate [8], iPersist [9], LLBL GenPro [7], Microsoft Entity Framework [17] entre otros.

Se han desarrollado otro tipo de productos y patrones de diseño que pueden catalogarse dentro de este paradigma, pero gozan de ciertas particularidades que alientan a analizarlos en forma separada. Un patrón de diseño que ha ganado popularidad en los últimos tiempos es *Active Record* [16]. El enfoque de este patrón es envolver una fila de la base de datos en una clase, de manera que se asocian filas únicas con objetos del lenguaje de programación usado. Cuando se crea uno de estos objetos se inserta una nueva fila en la base de datos. Cuando se modifican los atributos del objeto, se actualiza la fila de la base de datos. Castle ActiveRecord es un proyecto *open source* que implementa el patrón de diseño Active Record para .NET.

Más allá de los detalles de implementación, todos los productos analizados comparten un conjunto de características que presentamos a continuación.

- Independencia con el *DBMS*
  Una de las principales ventajas de este paradigma es que permite lograr independencia entre la DAL y el *DBMS* . Teóricamente no es obligatorio tocar una sola línea de código de la DAL en caso de ser necesario cambiar de *DBMS* , dependiendo del tipo de ORM basta con regenerar código en tiempo de compilación y/o cambiar determinada configuración. Estas herramientas poseen su propio lenguaje de consultas, cuya sintaxis, en algunos casos, suele tener alguna similitud a SQL. Al momento de ejecutar las consultas, el ORM genera dinámicamente la sentencia SQL. Esta funcionalidad permite desarrollar

una DAL desacoplada del tipo de base de datos, debido a que la sentencia SQL será generada en función del *DBMS* subyacente.

- Degradación de la performance
  La independencia con el *DBMS* puede provocar una degradación de la performance en comparación con la consulta escrita directamente en SQL. Un efecto generado por la independencia con el *DBMS* es no poder utilizar features particulares, tales como hints de Oracle o Top N de SQL Server, esto dificulta la optimización de las consultas. Al mismo tiempo se hace complicado o imposible en algunos casos analizar un plan de ejecución de consultas.

- Pérdida de expresividad al escribir las consultas
  Los lenguajes propios de consulta incluidos en los ORM no suelen tener el mismo poder de expresión que SQL, desventaja que comienza a notarse cuando es necesario escribir consultas con cierto grado de complejidad. En estos casos, es muy común que se decida generar una vista de los datos, escrita en el SQL nativo del *DBMS* , para que la consulta en el lenguaje propio sea más simple, esto rompe la independencia con el *DBMS* ya que en caso de ser necesario cambiarlo se deberá reescribir estas vistas.

Data Mappers

Este tipo de herramientas permiten encapsular en archivos XML todas las consultas SQL del sistema y mapear el resultado de la ejecución de las mismas en objetos, tales como DTO, que deben ser previamente definidos por el programador. La diferencia fundamental respecto a las herramientas de Mapeo-Relacional es el lenguaje de consultas, en los Data Mappers es directamente SQL mientras que en las herramientas Mapeo-Relacional es un lenguaje propietario. El producto *open source* IBatis [20] es un claro ejemplo de Data Mapper.
A continuación presentamos las características más importantes de

este paradigma.

- Dependencia con el *DBMS*
  Las consultas son escritas directamente en el SQL del *DBMS* , esto genera una dependencia entre la DAL y el *DBMS* , mismo fenómeno que se produce en el paradigma de consultas embebidas en el código.

- Modificación de consultas en tiempo de ejecución
  Dado que las consultas están almacenadas en archivos XML que son leídos al momento de ejecutarlas, es posible modificarlas en tiempo de ejecución, siempre y cuando no sean alterados los campos que la consulta retorna.

- No validación de consultas en tiempo de compilación
  Estas herramientas no realizan ninguna validación sintáctica ni semántica de las consultas SQL en tiempo de compilación, simplemente las transferirán al proveedor de conexión de la base de datos para luego mapear el resultado obtenido. Consultas erróneas o cambios en el modelo de datos generarán errores en tiempo de ejecución.

- Control de versionado
  Dado que las consultas están contenidas en archivos XML que son leídos en tiempo de ejecución, es muy importante mantener la versión correcta de estos archivos. Errores de versionado y/o instalación pueden generar problemas en tiempo de ejecución, por lo que es importante prestar especial atención a estos detalles.

Programación orientada a aspectos

La programación orientada a aspectos (AOP por sus siglas en inglés) es un paradigma de programación presentado formalmente en el año 1997 [15], el cual se basa fundamentalmente en el concepto de

aspecto. Un *aspecto* se define como un concepto que es imposible de encapsular claramente pues resulta diseminado por todo el código. Ejemplos clásicos de este concepto son tracing, logging y seguridad, funcionalidades que dentro del mundo de la programación orientada a objetos habitualmente están implementadas en diferentes clases. El aspecto de persistencia resulta de especial interés para el presente trabajo pues permitiría contemplar varios de los requerimientos establecidos por el cliente, en particular el de centralizar la lógica de la DAL.

A continuación presentamos las características más importantes de este paradigma [14].

- Separación de la persistencia
  La lógica de la persistencia se mantiene concentrada en un único lugar. Esto mejora el mantenimiento de la DAL e incrementa la modularidad del sistema.

- Nuevo paradigma de programación
  El hecho de tratarse de un nuevo paradigma de programación requiere que los desarrolladores sean capacitado en su teoría, reglas y construcciones.

- Tecnología emergente
  Es aún una tecnología emergente, esto la hace inaplicable fuera del ambiente académico.

Lenguaje de consulta integrado

Los lenguajes de programación con lenguaje de consulta integrado, integran en forma nativa construcciones que permiten definir y ejecutar consultas sobre una fuente de datos. Permitiendo de esta forma uniformidad en el código, la realización de chequeos sintácticos y de tipo en tiempo de compilación como también la asistencia por parte de las herramientas de desarrollo. La mayoría de estos lenguajes se basan en la integración de un lenguaje de programación orientado a objetos (lenguaje host) y un lenguaje de consultas (lenguaje guest)

[14]. Un ejemplo claro y popular de este paradigma es Linq [10] de Microsoft. A continuación presentamos las características más importantes de este paradigma.

- Chequeos en tiempo de compilación
  Es posible efectuar controles sintácticos y semánticos en tiempo de compilación tomado como referencia la fuente de datos a ser utilizada.

- Independencia con el *DBMS*
  Generalmente el lenguaje de consulta integrado es independiente del lenguaje de consultas subyacente, esto genera un independencia con el *DBMS* , incrementando la portabilidad de la aplicación.

- Pérdida de expresividad al escribir las consultas
  Los lenguajes de consulta integrados no suelen tener el mismo poder de expresión que SQL, desventaja que comienza a notarse cuando es necesario escribir consultas con cierto grado de complejidad. En estos casos, es muy común que se decida generar una vista de los datos, escrita en el SQL nativo del *DBMS* , para que la consulta en el lenguaje propio sea más simple, esto rompe la independencia con el *DBMS* ya que en caso de ser necesario cambiarlo se deberá reescribir estas vistas.

- Degradación de la performance
  Un efecto generado por la independencia con el *DBMS* es no poder utilizar features particulares, tales como hints de Oracle o Top N de SQL Server, esto dificulta la optimización de las consultas. Al mismo tiempo se hace complicado o imposible en algunos casos analizar un plan de ejecución de consultas.

2.2.2. Selección del paradigma adecuado

La selección del paradigma adecuado para el desarrollo de la DAL debe considerar ciertos factores de diseño de la aplicación, a contin-

uación listamos los más importantes:

- Escalabilidad de la aplicación
- Independencia con el *DBMS*
- Performance
- Mantenimiento
- Facilidad de desarrollo
- Calidad de código

La preferencia sobre algunos factores puede desmerecer otros. Un ejemplo claro de esto es la contraposición entre la independencia con el motor de base de datos y la performance. Lograr una buena performance implica, en la mayoría de los casos, la utilización de recursos específicos del motor de base de datos, hecho que genera una dependencia directa con el *DBMS* .

En función del peso que tenga cada uno de los factores de diseño debe optarse por el paradigma de desarrollo más adecuado. Este peso depende del tipo de sistema ha ser desarrollado. Analicemos a continuación algunos ejemplos. En un pequeño sistema web, donde no existe un complejo procesamiento de datos y en donde la mayoría de las pantallas son de mantenimiento y reportes, la facilidad de desarrollo y la independencia con el *DBMS* pueden ser los factores en los que se debe prestar mayor atención. Por otro lado, podemos considerar un sistema con un complejo procesamiento de datos y con fuertes requerimientos no funcionales de performance, en este caso, la escalabilidad y la performance deben ser los factores con mayor peso al momento de seleccionar el paradigma de desarrollo de la DAL.

2.3. Problemática del cliente

Tal como se mencionó en la introducción de este trabajo, el cliente se ha enfrentado a diversos problemas vinculados a la capa de acceso

a datos, principalmente porque sus sistemas son fuertemente dependientes de un modelo de datos. Estos problemas lo han llevado a cambiar el paradigma de desarrollo de la DAL en varias oportunidades, pero en todas ellas se ha tropezado con diversos inconvenientes técnicos, motivo que ha inspirado la creación del presente trabajo.

Con el objetivo de ofrecer al lector una visión clara de la problemática del cliente, se presenta el cuadro 2.3 que ilustra los requerimientos planteados por el cliente contra los diferentes paradigmas de desarrollo de la DAL analizados previamente. Algunas características positivas de los paradigmas no han sido incluidas en esta tabla debido a que el cliente no las considera importantes, tal como la independencia con el *DBMS* (*PayTrue Solutions* [1] utiliza exclusivamente Oracle [2]).

La conclusión más importante que puede extraerse de esta tabla es que ninguno de los paradigmas disponibles cubre el conjunto completo de requerimientos, hecho que justifica plenamente el desarrollo de este proyecto.

Cabe aclarar en este punto que el proyecto no sólo se ha restringido al desarrollo de software que cubra las necesidades del cliente, sino también al análisis previo de varias herramientas disponibles en el mercado. El resultado de este análisis, cuyo resultado se plasmó en este capítulo, permitió determinar que ninguna de las herramientas cumplía con los requisitos del cliente. Por otro lado, el análisis de las herramientas *open source* disponibles indicaron que el esfuerzo necesario para modificarlas y de esta forma satisfacer los requisitos planteados era extremadamente alto. Este análisis fue el que determinó que era necesario el desarrollo de las herramientas entregadas en este trabajo.

	Acceso directo	ORM	Data Mapper	AOP	Lenguaje integrado
Control de consultas SQL en tiempo de compilación	No	Si	No	No	Si
Generación de código de la DAL	No	Si	Si	No	No
Centralización de las consultas	No	Si	Si	Si	No
Performante	Si	No	Si	No	No
Análisis de impacto ante cambios en la base de datos	No	Si	No	No	No
Uso directo de SQL	Si	No	Si	No	No
Control sistemático de consultas SQL contra una base Oracle [2]	No	No	No	No	No
Fácil mantenimiento de la DAL	No	Si	Si	Si	No
Uniformidad del código de la DAL	No	Si	Si	Si	No

Cuadro 2.1: Requisitos vs. Paradigmas de desarrollo de la DAL

Capítulo 3

Herramientas desarrolladas

3.1. Introducción

En este capítulo analizaremos en detalle las herramientas desarrolladas para solventar la problemática descripta en el punto anterior, cumpliendo con los objetivos detallados en el punto 1.2.

Los distintos objetivos determinan claramente las siguientes áreas:

- Generación de capa de acceso a datos
  En la cual se encapsulan los objetivos:
  - Generación de código
  - Generación de metadata del modelo de datos y de consultas
  - Aplicación de reglas
  - Verificación de consultas
- Verificación sistemática
  Corresponde con el objetivo:
  - Analizador sistemático de consultas
- Análisis de impacto
  Corresponde con el objetivo:
  - Análisis de impacto

Para cada área, se ha tomado un enfoque en particular, buscando siempre una interacción simple entre las mismas, y una coexistencia en conjunto con las herramientas de desarrollo ya utilizadas en *PayTrue Solutions* , generando de esta manera un nuevo set de herramientas que se acople de manera natural al entorno existente.

Las herramientas desarrolladas para cada una de las áreas comparten información, en particular del modelo de datos y de las consultas desarrolladas, para ello se utiliza una base de datos en común.

Esta base de datos es portable y se encuentra embebida en el conjunto de herramientas, para facilitar la instalación de las mismas, evitando configuraciones y necesidades particulares de entorno.

En cuanto a la **generación de la capa de acceso a datos**, se aplica un enfoque diferente a los clásicamente planteados, el cual se centra en facilitar la escritura de las consultas en lenguaje SQL, sin preocuparse por los detalles correspondientes a la implementación de las operaciones que ejecutarán estas consultas.

Para lograrlo se utiliza una estrategia de generación de código, a partir de las consultas creadas por el usuario e información sobre el modelo de datos.

El código generado posee las siguientes características:

- Manejo de datos de forma tipada
- Generación de métodos con firmas en dos formatos
  - Lista de parámetros
  - Tipo de datos
- Los métodos generados retornan la información de dos maneras:
  - Dataset
  - Lista del tipo de datos correspondiente

- Se permiten añadir a grupos de consultas determinados por el usuario:
  - Secciones de código antes o después de la ejecución de la consulta.
  - Filtrado de datos según parámetros determinados por el usuario.

En cuanto al **análisis de impacto**, corresponde a verificar los usos de elementos de la base de datos en las consultas disponibles en la capa de acceso a datos. La estrategia utilizada se basa en el mantenimiento de la metadata de las consultas en la base de datos embebida, referenciando a la metadata del modelo de datos que utiliza.

El área de **validación sistemática**, tiene como objetivo verificar que en un momento dado la versión de la base de datos en uso corresponde con la versión de los datos del modelo utilizados por el sistema.

Esta validación consiste en verificar contra el catálogo de Oracle [2] de la base de datos indicada, que la estructura de la misma corresponde exactamente con la estructura indicada en la metadata.

Por cada una de las áreas se ha desarrollado una herramienta, las mismas comparten la información almacenada en la base de datos embebida, siendo la de generación de la capa de acceso a datos la proveedora de información, la cual es consumida por las tres herramientas.

A continuación analizaremos en detalle cada una de éstas herramientas.

3.2. Generator

Esta herramienta proporciona los medios para poder generar y mantener fácilmente la DAL, tanto para la creación y modificación de

consultas, así como también para las tareas sistemáticas realizadas sobre éstas.

Para poder homogeneizar el código de la DAL, se aplica un enfoque de generación de código, el cual permite mantener en un lugar centralizado las principales características del mismo.

Esto facilita la implementación de buenas prácticas de programación, corrección de errores, y aplicación de cambios. A su vez evita la generación innecesaria de código repetido, que en muchos casos se copia y reutiliza, multiplicando la posibilidad de incorporación de errores y dificultando las tareas de mantenimiento.

A continuación, en esta sección se brinda una visión general, para dar un acercamiento al funcionamiento de la herramienta, analizando en detalle posteriormente los elementos principales que conforman la misma.

Por último se proporciona una descripción del flujo de ejecución, en donde se detallan los pasos que sigue la herramienta al ser ejecutada, se muestra un ejemplo de uso de la misma, y los problemas conocidos.

3.2.1. Visión general

Para poder generar el código de las consultas, la herramienta maneja la información de los elementos del modelo de datos en forma tipada, de manera de poder utilizarlos y referenciarlos dentro del ambiente de programación sin ambigüedades, proporcionando al usuario firmas claras en los métodos generados.

Para obtener la información tanto del modelo de datos como de las consultas, se analizan los scripts DDL y DML respectivamente, extrayendo la información correspondiente. Esto se logra mediante la generación de una gramática, la cual contiene un subconjunto de

las sentencias soportadas en Oracle [2], esta gramática denominada Gramática TQL es analizada en detalle en el punto 3.2.2.

Esta información es denominada **metadata** (detallada más adelante en el punto 3.2.2) y es almacenada en una base de datos embebida. Este tipo de base de datos permite manejar la información de forma flexible e independiente, sin realizar ningún tipo de configuración, siendo transparente para el usuario.

La herramienta permite generar consultas recibiendo la información de las mismas en un archivo con extensión ".tql", la estructura de este tipo archivo se detalla más adelante en el punto 3.2.3.

Las consultas son escritas por el usuario basándose en la sintaxis Oracle [2] de uso común. La herramienta valida la correctitud sintáctica y semántica de cada consulta en tiempo de compilación y genera el código de los métodos que permiten ejecutar las mismas.

Los métodos para ejecución de consultas se generan en dos modalidades, invocación con lista de parámetros o tipo de datos, siendo ambas tipadas, obteniendo los tipos de datos a partir de la metadata del modelo.

Estas consultas pueden tener parámetros, los cuales también son tipados, mapeando el tipo de datos del parámetro a partir del contexto en el cual se encuentra, analizando las columnas con las cuales interactúa y el tipo de datos de las mismas.

Existen tareas sistemáticas a realizar sobre conjuntos de consultas que corresponden con ciertos criterios. Para poder realizar estas tareas, la herramienta ofrece una funcionalidad denominada Reglas.

Las Reglas también son definidas en archivos TQL siendo detalladas más adelante en el punto 3.2.2.

A continuación se analizan en detalle cada uno de los elementos principales, los cuales conforman la base de la estructura de la herramienta, y merecen una sección independiente para ser analizados.

3.2.2. Elementos principales

Gramática TQL

Se ha desarrollado una gramática basada en un subconjunto de las sentencias correspondientes a la referencia de SQL Oracle [2] 10g, pretendiendo que este sub-conjunto abarque las sentencias más frecuentemente utilizadas.

Esta gramática, denominada Gramática TQL, es utilizada por la herramienta Generator para interpretar scripts DDL que definen el modelo de la base de datos sobre la cual se desea trabajar, y las consultas DML especificadas por los usuarios. Para el desarrollo de esta gramática se ha utilizado *ANTLR* [19].

ANTLR [19] es un generador de analizadores sintácticos, definiendo en primera instancia una gramática en un lenguaje específico, determinando los tokens y las reglas correspondientes a la misma, para posteriormente compilarla y generar el código de los analizadores sintáctico y lexicográfico correspondientes.

Para generar la gramática TQL, se tomó como base una ya existente de Oracle [2] para *ANTLR* [19], adaptándola a las necesidades del proyecto. Debido al gran tamaño de la gramática, y dado que los analizadores sintácticos de DDL y DML se utilizan por separado, se han generado dos gramáticas, con las reglas necesarias para interpretar scripts DDL y DML respectivamente.

Dado que *ANTLR* [19] utiliza análisis sintáctico LL, se han aprovechado tanto la recursión como el hecho de que permite agregar código dentro de las reglas. Es decir, en la gramática, se puede agregar código *C#* en las reglas, que luego será colocado en el código generado correspondiente al analizador sintáctico.

De esta manera se han capturado los elementos utilizados en las sentencias, para replicarlos en una estructura. Esta estructura es utilizada posteriormente para validar semánticamente las sentencias y generar la metadata correspondiente. En los siguientes puntos se detallan las sentencias soportadas para cada tipo de consulta.

A continuación se analizan las sentencias más importantes de las cuales se almacena información en la metadata.Dado que la gramática

soporta muchas más sentencias, como por ejemplo HINTS entre otras, se puede analizar la lista completa de sentencias soportadas en el apéndice D.

- DDL

Sentencia	**CREATE, ALTER, DROP**
TABLE	Sí
COLUMN	Sí
SEQUENCE	Sí
SYNONYM	Sí
INDEX	Sí

Cuadro 3.1: Sentencias DDL soportadas

Sentencia	**Soportado**
PRIMARY KEY	Sí
FOREIGN KEY	Sí
CHECK	Sí

Cuadro 3.2: Constraints DDL soportados

- DML

Sentencia	**Soportado**
SELECT	Sí
UPDATE	Sí
INSERT	Sí
DELETE	Sí
JOIN	([LEFT,RIGHT] [INNER,OUTER] [CROSS])
Conjuntos	UNION, INTERSECT, MINUS, EXCEPT

Cuadro 3.3: Sentencias DML soportadas

Metadata

La metadata se puede dividir claramente en dos, la metadata del modelo, cargada en un inicio a partir de los scripts DDL correspondientes a la creación de la base de datos, y la metadata de las

consultas, generados a partir de la interpretación de los archivos TQL de los usuarios.

Como se mencionó en la visión general de Generator3.2.1, esta metadata es almacenada en una base de datos embebida, en particular se utiliza SQLite [21], el cual es un sistema de gestión de base de datos embebida, gratuito y muy difundido, siendo el gestor más apropiado para el proyecto.

Este tipo de base de datos es transparente para el usuario, dado que no requiere ningún tipo de instalación ni configuración, facilitando así el uso de la herramienta.

Por más que la metadata se divide conceptualmente en dos, ambas partes se encuentran fuertemente relacionadas, dado que las consultas deben referenciar a la metadata del modelo que utilizan.

Dada esta necesidad, los dos tipos de metadata se almacenan en la misma base de datos embebida.

Esta base de datos es fundamental para el análisis de impacto 3.3, dado que proporciona una forma simple de acceso a la toda la información que esa herramienta necesita.

También es fundamental para la generación de código, dado que proporciona toda la información necesaria sobre las consultas, elementos de la base de datos utilizados, tipos de datos de las columnas, entre otras cosas. En los siguientes puntos se detalla la información que la metadata registra para cada área.

- Modelo
  En la tabla 3.4 se puede apreciar la metadata del modelo.

- Consulta
  En este punto se detallan la metadata de las consultas. Se registran los datos básicos de las mismas, así como también datos sobre la generación de código y métodos generados para éstas. Los datos básicos de las consultas se pueden apreciar en la tabla 3.5.

Metadata de generación de código para la consulta:

Objeto	Información
Tablas	Nombre y Módulo correspondiente Primary key (columnas correspondientes) Foreign keys (columnas propias, tablas y columnas referenciadas) Checks (columnas correspondientes) y condición relacionada
Columnas	Nombre Módulo Tabla a la cual pertenece Tipo de datos o namespace y nombre del enumerado relacionado Valor de los atributos length, is null, y default value
Secuenciadores	Nombre Módulo Columna relacionada Valores de los atributos increment y startwith
Sinónimos	Nombre Módulo Objeto referenciado
Índices	Nombre Módulo Tipo

Cuadro 3.4: Metadata del modelo

Nombre	Descripción
Nombre	Nombre de la consulta
Tipo	(SELECT,UPDATE,INSERT, DELETE)
Objetos Referenciados	Objetos referenciados de la metadata del modelo, en el caso de las columnas,se distinguen con un booleano a las que pertenecen a la sentencia principal, de manera de poder diferenciarlas del resto.

Cuadro 3.5: Metadata básica de la consulta

- Consulta correspondiente
- Namespace donde se generan los métodos correspondientes
- Nombre del archivo en el cual se generaron los métodos

Metadata por cada método generado:

- Consulta a la que corresponde

- Nombre del método
- Número de línea en el archivo de código

Reglas

Las reglas son utilizadas para aplicar comportamientos comunes a varias consultas de forma centralizada y simple.

En muchos casos se desea que todas las consultas que cumplan con ciertas características tengan un tratamiento especial, o aplicar cierto comportamiento globalmente a todas las consultas.

Para lograrlo de una manera flexible se ha diseñado la opción de crear reglas en archivos TQL.

Estas reglas poseen dos características:

- Permiten determinar a que consultas se aplica la regla fácilmente.
- Permiten filtrar datos o inyectar código a las consultas.

Para que el usuario pueda determinar de manera simple las consultas que serán afectadas por la regla, se utiliza la base de datos de la metadata, sobre la cual se ejecutará la consulta indicada por el usuario obteniendo el identificador de todas las consultas afectadas.

De esta manera se explota la gran flexibilidad que ofrece el almacenamiento de la metadata en una base de datos relacional.

Existen dos tipos de reglas soportadas por la herramienta:

1. Filtro
   Permite filtrar consultas, agregando condiciones a todas las cláusulas WHERE de las consultas que cumplan alguna condición. En este tipo de reglas, se indica que código se desea agregar, y los parámetros que se utilizan en el mismo.

   Dado que estas reglas están enfocadas a tareas administrativas, las cuales generalmente pretenden ser transparentes para

el usuario común, y para no afectar la firma de los métodos de las consultas, los valores de los parámetros de la regla son obtenidos mediante llamadas a métodos previamente creados.

El código es agregado como una condición WHERE de un SELECT que abarca la consulta original, es decir, la herramienta toma la consulta original y realiza lo siguiente:

```
SELECT * FROM consultaOriginal WHERE filtroInyectado;
```

2. Inyección de código
   Permite realizar llamadas a métodos, previamente y/o posteriormente a la ejecución de las consultas que cumplan alguna condición. Los métodos inyectados recibirán un hashtable con todos los datos disponibles, estos datos son detallados en la tabla 3.6.

Clave del hash	**Contenido**
AFFECTED_ROWS	Información de las filas afectadas por la ejecución.
DATA_ADAPTER	Permite acceder al data adapter utilizado en la ejecución de la consulta
QUERY	Código de la consulta ejecutada o a ejecutar
QUERY_DATASET_RESULT	Dataset con el resultado de la consulta
QUERY_DTO _RESULT	DTO con el resultado de la consulta
QUERY_PARAMETERS	Valores de los parámetros de la consulta
QUERY_TYPE	Tipo de consulta
REFERENCED_TABLES	Tablas referenciadas en la consulta
RETURNED_ROWS	Información de las filas retornadas por el comando

Cuadro 3.6: Datos almacenados en el hashtable disponible en Reglas de Inyección de Código

Dependiendo del tipo de regla, el cual determina el momento de ejecución del método inyectado (PRE o POST), se cuentan con más o menos datos.

Dado que, luego de ejecutar la consulta se cuenta con la información correspondiente al resultado de la misma, estas inyecciones de código permiten realizar tareas como por ejemplo:

- Auditorías:
  Registrar los cambios realizados sobre tablas o columnas

que cumplan alguna condición junto con datos adicionales típicos de una bitácora (fecha, hora, usuario).

- Manejo de permisos:
  Impedir cambios sobre ciertas tablas o columnas a menos que el usuario tenga permiso específico para realizarlos.
- Log:
  Registrar las consultas ejecutadas y detalles sobre la ejecución en archivos de log.

3.2.3. Estructura de Archivo TQL

Un archivo TQL es un documento XML basado en un esquema claramente definido, el cual permite especificar los datos de entrada, de forma simple y estructurada.

Observando el diagrama correspondiente al XSD, analizaremos cada elemento del mismo para detallar su contenido.

Todo el contenido del documento se encuentra dentro de un tag denominado **Data**.

Dado que un archivo TQL puede contener definición tanto de reglas como de consultas, existen dos elementos, uno para cada grupo de definiciones, permitiendo definir la cantidad que se desee de cada una de estas.

El diagrama correspondiente se puede apreciar en la Figura 3.1.

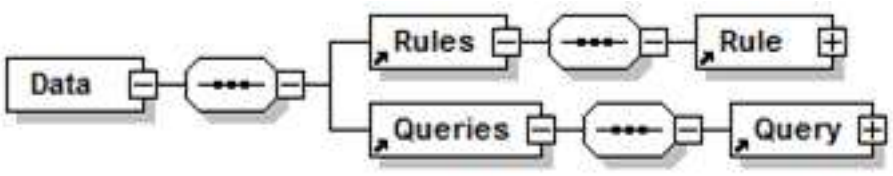

Figura 3.1: Estructura base de archivo TQL

- Consultas

En el caso de las Consultas, se pueden crear tantos elementos **Query**, dentro del elemento **Queries**, como se desee, indicando su nombre en el atributo "name", y el código de la consulta en el atributo "source", especificando los parámetros de la forma: ":nombreParametro" dentro de la consulta.

Se pueden indicar los parámetros que son opcionales, agregándolos a la colección opcional de ConditionalParameters, indicando en el atributo "name", el nombre del parámetro correspondiente. Esto permite que el código de consulta generado tenga en cuenta que esos parámetros pueden ser especificados o no. El diagrama correspondiente a la estructura de una consulta se puede apreciar en la Figura 3.2.

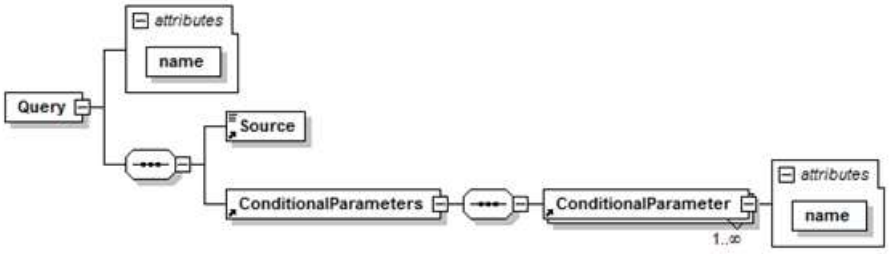

Figura 3.2: Estructura XML de una consulta

- Reglas

En el caso de las Reglas, se pueden crear tantos elementos **Rule**, dentro del elemento **Rules**, como se desee.

Atributos del elemento **RULE**:

Atributo	Descripción
Name	Nombre de la regla

Cuadro 3.7: Atributos del elemento rule

Se debe especificar también a que consultas se debe aplicar la regla, para ello, en un elemento **Appliance**, se agrega dentro de un tag **CDATA** el código de una consulta sobre la metadata. Esta consulta

debe retornar los identificadores de todas las consultas a las cuales se les desea aplicar la regla.

En caso de desear agregar una regla de tipo Filtro, se puede agregar un elemento **Filter**, indicando dentro del elemento **Code**, en un tag **CDATA**, el código a agregar. Agregando en caso de utilizar parámetros en dicho código, los datos de cada uno de éstos en un elemento **Parameter**, dentro de **Parameters**.

Atributos del elemento **PARAMETER**:

Atributo	**Descripción**
Name	Nombre del parámetro
Namespace	Espacio de nombres donde se encuentra la clase que posee el método que obtiene el valor del parámetro.
Class	Nombre de la clase que posee el método que obtiene el valor del parámetro.
Method	Nombre del método que obtiene el valor del parámetro.

Cuadro 3.8: Atributos del elemento parameter

En caso de desear agregar inyecciones de código, se pueden agregar las mismas dentro del elemento **Inyections**, en un elemento **Inyection**.

Atributos del elemento **INYECTION**:

Atributo	**Descripción**
Type	Tipo de inyección deseada (PRE, POST, BOTH)
Namespace	Espacio de nombres donde se encuentra la clase que posee el método a inyectar
Class	Nombre de la clase que posee el método a inyectar
Method	Nombre del método a inyectar

Cuadro 3.9: Atributos del elemento inyection

El diagrama correspondiente a la estructura de una regla se puede apreciar en la Figura 3.3

3.2.4. Flujo de ejecución

En esta sección se detalla cual es el flujo de trabajo que desencadena internamente el uso de la herramienta, detallando para cada paso,

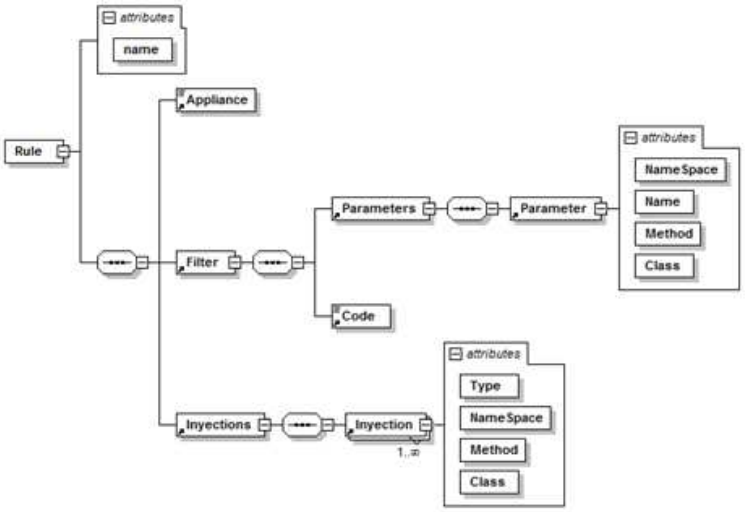

Figura 3.3: Estructura XML de una regla

que archivos se utilizan, que datos se actualizan, y a que elementos de la herramienta se referencia.

El diagrama de flujo se puede observar en la figura 3.4.

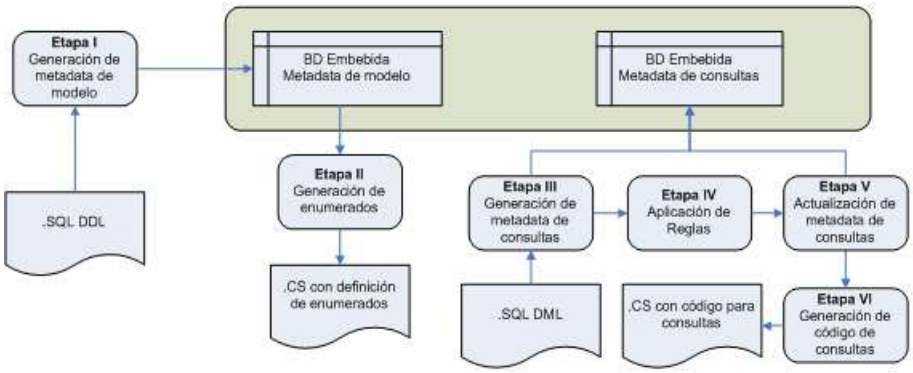

Figura 3.4: Generator - Flujo de ejecución

La herramienta es utilizada desde la línea de comandos. La misma recibe los siguientes parámetros:

- Ruta donde se encuentran los archivos SQL que contienen la definición de la base de datos (DDL).

- Ruta donde se encuentra el proyecto correspondiente a las interfaces del sistema, al cual se desean añadir las interfaces, enumerados y datatypes generados por la herramienta.
- Ruta donde se encuentra el proyecto correspondiente a la capa de acceso a datos, al cual se desean añadir los métodos generados.

Recibiendo estos tres parámetros, la herramienta procede de la siguiente manera:

Etapa I - Generación de metadata de modelo

Busca todos los archivos con extensión ".sql" ubicados en la ruta indicada en el primer parámetro. Luego, por cada archivo realiza el análisis sintáctico del mismo, aplicando las reglas de la gramática. Una vez culminado el procesamiento de los archivos, si la sintaxis del script era correcta, se procede a almacenar los datos obtenidos en la base de metadata del modelo.

Etapa II - Generación de enumerados

A partir de los datos obtenidos en la Etapa I, se procede a la generación de los enumerados necesarios. Los mismos son agregados al proyecto Interface, correspondiente a la ruta del segundo parámetro.

Etapa III - Generación de metadata de consultas

Se realiza una búsqueda de los archivos TQL que se encuentren dentro del proyecto DAL, indicado como tercer parámetro. Por cada archivo encontrado, se procede a analizar sintácticamente la consulta del mismo, extraer la información de parámetros condicionales y de reglas. Si la sintaxis del script era correcta, se realiza la validación semántica. De pasar esta última validación, se procede a almacenar los datos de la consulta y continuar con la siguiente.

Etapa IV - Aplicación de reglas

Una vez finalizado el análisis y guardado de datos de todos los TQL, se procede a ejecutar cada regla encontrada. Se realiza la búsqueda de las consultas que aplican a la regla, ejecutando la consulta indicada en la regla para éste uso. Se procede a la aplicación de las reglas por cada consulta a la que corresponda. Luego de aplicar las reglas, la nueva consulta generada se valida sintáctica y semánticamente.

Etapa V - Actualización de metadata de consultas

Para las consultas a las cuales se les han aplicado reglas, y han pasado las validaciones, la metadata es almacenada, sobrescribiendo la guardada anteriormente.

Etapa VI - Generación de código de consultas

Una vez obtenida la información completa de las consultas, se procede a generar el código de los métodos correspondientes a cada una. Éste código es agregado en el proyecto DAL, indicado como tercer parámetro, El código correspondiente a los datatypes a utilizar, se agregan al proyecto interface correspondiente al segundo parámetro.

3.2.5. Ejemplo de uso - experiencia de usuario

En este caso de ejemplo, realizaremos los pasos necesarios para generar los métodos de acceso a datos correspondientes con las siguientes consultas:

- `UPDATE CT_CONTRACT SET CONTRACT_NUMBER=100 WHERE ID_CONTRACT=:contrato;`
- `SELECT C.ID_CONTRACT, CO.ID_COMPANY FROM CT_CONTRACT C, CT_COMPANY CO WHERE C.CONTRACT_NUMBER=:contract_number AND C.ID_COMPANY=CO.ID_COMPANY;`
- `SELECT SRD_SERVER.NEXTVAL FROM DUAL;`

Creación de archivo TQL

El primer paso consiste en generar el archivo TQL correspondiente, para ello, respetando la estructura indicada en el punto 3.2.3, generamos el archivo que contendrá a las tres consultas.

El mismo se puede ver en la Figura 3.5.

Generación de código

Una vez generado el archivo TQL, el siguiente paso es la generación de código. Para ello se debe ejecutar la herramienta Generator, pasando como parámetros:

- Ruta de archivos DDL, los cuales contienen la definición del modelo de datos a utilizar, la herramienta los analizará y generará la metadata correspondiente.
- Ruta del archivo de proyecto interface donde se agregarán los enumerados, datatypes, y demás elementos generados por la herramienta.
- Ruta del archivo de proyecto de capa de acceso a datos, donde se agregarán las clases con los métodos generados, dentro de la ruta correspondiente a este archivo, se realizará la búsqueda de archivos TQL, los cuales serán procesados para generar el código correspondiente.

Para ejecutarlo, desde una consola MS-DOS, a partir de un archivo ".bat", o una tarea Nant, u otros mecanismos definidos por el usuario, ejecutar la herramienta de la siguiente manera:

Generator.exe <Ruta de archivos DDL ><Ruta completa al archivo de proyecto de interface Interface.csproj><Ruta completa al archivo de proyecto de la DAL DataAccessLayer.csproj>

El resultado será similar al de la Figura 3.6, donde se muestran los archivos DDL procesados y el resultado de cada procesamiento,

así como también los resultados de generación de metadata y código correspondientes a DDL y DML.

Una vez finalizada la ejecución del script, quedan disponibles los métodos dentro del proyecto indicado.

En el caso del ejemplo, en la Figura 3.7 se pueden apreciar los archivos generados.

Dentro del proyecto Data Access Layer, se ha generado al mismo nivel del archivo TQL, la clase ContractQueries.cs, la cual contiene el código de las consultas ubicadas en el mismo.

En el proyecto Interface, entre otras cosas, se han generado los DTO, y los enumerados utilizados por los métodos de la clase Contract-Queries.cs.

El código generado lo veremos más adelante, dado que es de interés agregar reglas a éstas consultas.

Creación de reglas

En este ejemplo, crearemos dos reglas, una de cada tipo (Filtro e inyección). La sección agregada al archivo TQL se puede apreciar en la Figura 3.8.

- Filtro

Para la primer regla, se indica en "appliance" la consulta a ejecutar contra la base de datos de metadata para obtener los identificadores de las consultas correspondientes. En este caso se quiere seleccionar la consulta por su nombre, `SELECT IDQUERY FROM QM_QUERY WHERE QUERYNAME = 'GetContractIdByContractNumber';`.

A esta consulta se le desea aplicar un filtro, el cual concatenará una cláusula "where" con el código indicado en la sección "Code" (ROWNUM <= :maxRows), de esta forma se consigue limitar la cantidad de filas retornadas por la consulta.

El valor del parámetro "maxRows"se indica en la colección de parámetros de la regla, <Parameter Name="maxRows" NameSpace="Utilities" Method="GetMaxRows" Class="Util" Module="BC"/>, esto indica que el parametro se obtendrá ejecutando el método Utilities. Util. GetMaxRows.

- Inyección

La segunda regla corresponde a un ejemplo de inyección de código, de la misma manera que la regla anterior, la consulta de "appliance" puede ser tan compleja como el usuario necesite siempre y cuando retorne "IDQUERY".

En esta regla se declaran tres "validators", uno por cada tipo de inyección (PRE, POST, BOTH).

El tipo de inyección determina el momento en el cual se ejecuta el método solicitado, puede ser antes de ejecutar la consulta, después o ambos.

De la misma manera que los parámetros de las reglas de Filtro, se indican los métodos a ejecutar para las inyecciones de código. Por ejemplo para la validación de tipo pre, se ejecuta el método Paytrue.PayStudio.Checks.ValidateAmountPre.

Generación de código

El código generado varía según las características de las consultas, para las del caso de ejemplo se generan los siguientes métodos:

1. GetContractIdByContractNumberDs

   a) `public DataSet GetContractIdByContractNumberDs (long maxRows,long contract_number)`

   b) `public DataSet GetContractIdByContractNumberDs (GetContractIdByContractNumberInputDTO dto)`

c) `public List <GetContractIdByContractNumberOutputDTO> GetContractIdByContractNumberLs (long maxRows,long contract_number)`

d) `public List <GetContractIdByContractNumberOutputDTO> GetContractIdByContractNumberLs (GetContractIdByContractNumberInputDTO dto)`

2. GetNextSequenceLs

a) `public List<GetNextSequenceOutputDTO> GetNextSequenceLs ()`

b) `public DataSet GetNextSequenceDs()`

3. GetContractNumberById

a) `public int GetContractNumberById (long contrato)`

b) `public int GetContractNumberById (GetContractNumberByIdInputDTO dto)`

Se generan métodos que retornan datasets o listas de tipos de datos para las consultas que retornan una colección de elementos. El tipo de datos es generado por la herramienta conteniendo los datos exactos que retornará cada elemento de la consulta de forma tipada.

Para las que retornan un valor determinado, se generan los métodos retornando el tipo de datos adecuado.

Para las consultas que reciben más de un parámetro se generan dos versiones de los métodos, una con la lista de parámetros y otra con un tipo de datos que contiene la lista de parámetros, siempre tipados.

El texto de las consultas es el escrito por el usuario, validado semántica y sintácticamente por la herramienta.

En el caso de la consulta a la cual se le aplicaba la regla de tipo filtro, se ha encapsulado el código SQL, para dejarlo de la siguiente manera:

```
SELECT * FROM (SELECT C.ID_CONTRACT, CO.ID_COMPANY FROM
CT_CONTRACT C, CT_COMPANY CO WHERE
C.CONTRACT_NUMBER = :CONTRACT_NUMBER AND
C.ID_COMPANY = CO.ID_COMPANY) AS TQL_AUX
WHERE ROWNUM <= :MAXROWS
```

y el parámetro "MAXROWS" es agregado a la colección de parámetros del comando: command.Parameters.Add("MAXROWS", OracleDbType.Long, maxRows, ParameterDirection.Input);

En la sección donde se colocan las llamadas a los métodos inyectados de tipo PRE, se añade la llamada a la operación que obtiene el parámetro:

```
Utilities.Util.GetMaxRows(queryDataHash);
```

Se puede apreciar que el método recibe como parámetro el queryDataHash, el mismo posee toda la información disponible hasta el momento acerca de la ejecución de la consulta:

```
 queryDataHash.Add(EnumQueryDataHashKeys.QUERY,
command.CommandText);
queryDataHash.Add(EnumQueryDataHashKeys.QUERY_TYPE,
EnumQueryType.SELECT);
queryDataHash.Add(EnumQueryDataHashKeys.DATA_ADAPTER,
oDataAdapter);


ArrayList referencedTables = new ArrayList();
referencedTables.Add(new CT_COMPANY());
referencedTables.Add(new CT_CONTRACT());


queryDataHash.Add(EnumQueryDataHashKeys.REFERENCED_TABLES,
referencedTables);
```

En el caso de la consulta a la cual aplicaba la regla de inyección, se han añadido las llamadas a los métodos correspondientes en las secciones correspondientes:

- Métodos inyectados de preejecución

```
Paytrue.PayStudio.ValidateAmountPre.Checks(queryDataHash);
Paytrue.PayStudio.ValidateAmountBoth.Checks(queryDataHash);
```

- Ejecución de la consulta

```
affectedRows = command.ExecuteNonQuery();
queryDataHash.Add(EnumQueryDataHashKeys.AFFECTED_ROWS,
affectedRows);
```

- Métodos inyectados de postejecución

```
Paytrue.PayStudio.ValidateAmountPost.Checks(queryDataHash);
Paytrue.PayStudio.ValidateAmountBoth.Checks(queryDataHash);
```

Uso de código

Los métodos quedan disponibles en la clase ContractQueries, y pueden ser utilizados en la solución según sea necesario.

Como se ha podido apreciar, la tarea del usuario se limita a la creación de las consultas, armado del archivo TQL, diseño de las reglas de ser necesario y ejecución de Generator. Luego de estos pasos tiene disponible en su proyecto los métodos correspondientes.

3.2.6. Problemas conocidos

Existen una serie de limitaciones, principalmente en el ámbito de la gramática e interpretación de la misma. Dado que la misma es un subconjunto de reglas de la gramática de Oracle [2], existen varias

sentencias que no son soportadas por el momento, pero se pueden agregar en un futuro.

También existen una serie de restricciones sobre la gramática TQL actual, la cual no soporta ciertas sentencias en casos particulares, como ser:

- Palabras clave: no se aceptan ciertas palabras clave como nombres de elementos de la base de datos (value, body, table, column, position, year, translation, entre otros). Estas palabras pueden ser aceptadas si son referenciadas entre comillas.
- Nombres de objetos: no se están soportando nombres de objetos con caracteres especiales.

3.3. Análisis de impacto

El objetivo de esta herramienta es poder buscar información respecto al uso de los objetos de la base de datos en las consultas TQL existentes en el desarrollo.

Como se ha mencionado previamente, Generator crea metadata tanto del modelo como de las consultas en una base de datos relacional.

En la misma se registra la información del modelo de datos y de las consultas, en particular, que objetos del modelo son utilizados.

Para cumplir con los objetivos de esta herramienta no ha sido necesario desarrollar una aplicación, dado que por medio de cualquier herramienta de administración de bases de datos que soporte conexiones con bases SQLite [21], y permita ejecutar consultas SQL contra la misma, éstos son abarcados completamente.

De esta manera, desde el gestor de bases de datos se pueden ejecutar cualquier tipo de consultas soportadas por SQL, para obtener los datos deseados, siendo esto muy flexible y fácil de usar.

Se han desarrollado una serie de vistas, disponibles en la base de datos de la metadata, para facilitar las consultas más frecuentes a los usuarios.

Se ha considerado que las consultas más frecuentes serían por tipo de objeto, por lo cual las vistas desarrolladas corresponden con la obtención de los datos de los distintos tipos de objetos de la base de datos.

3.4. Verificación sistemática de consultas

Esta herramienta permite validar la estructura del modelo de datos almacenado en la metadata, contra el catálogo de una base de datos real de Oracle [2]. De encontrarse diferencias las mismas son reportadas en un informe final de ejecución.

De esta manera se puede comprobar que se está trabajando con la versión adecuada del modelo de datos.

Esta comprobación es fundamental dado que las herramientas desarrolladas se encuentran basadas en la metadata que la misma registra, los cuales como ya mencionamos se encuentran divididos conceptualmente en dos, metadata del modelo y de las consultas.

Tanto la metadata de consultas como el código generado, dependen totalmente del modelo de datos sobre el cual se trabaja, por lo cual un cambio en el modelo de datos puede provocar que toda la capa de acceso a datos quede obsoleta.

Para evitar este tipo de problemas esta herramienta ofrece la posibilidad de comprobar la consistencia entre la metadata del modelo de datos contra el modelo de datos que utiliza la aplicación realmente.

3.4.1. Funcionamiento

A continuación se brinda una breve descripción del funcionamiento de la herramienta.

En primer lugar se ejecutan varios test para cada consulta detectada, asignándole una categoría a cada una de estas siendo las categorías posibles Error y Warning, según si la consulta deja de funcionar o no respectivamente.

Una vez finalizado el test, se retorna un reporte en donde se especifican las fallas detectadas, dando el detalle del objeto de la base de datos que provoca el error, la consulta y el archivo en la que se encuentra, ordenadas por categoría.

La herramienta desarrollada cuenta con una DAL desarrollada utilizando la herramienta Generator. Se brindan mas detalles acerca de esta herramienta en el punto C.1.

3.5. Integración con el entorno de desarrollo

Se ha tenido especial cuidado a la hora de diseñar las herramientas, dado que una característica fundamental de las mismas debía ser la integración simple al entorno de desarrollo.

En particular, para la herramienta de generación de la DAL, se analizó en que etapa del proceso de desarrollo se genera, pudiendo la misma generarse en primera instancia, o a medida que se genera la capa de dominio, se crean las operaciones según se necesita.

Dado que pueden darse ambos casos, según las necesidades particulares del desarrollador, se ofrece una herramienta flexible, la cual permite generar las consultas previamente al desarrollo de la capa de dominio, o a medida que se crea la misma.

Esto es posible dado que para crear las operaciones de acceso a datos, el usuario simplemente genera los archivos TQL en un directorio o estructura de directorios determinada, pasando como parámetro a la herramienta la raíz de la estructura que los contiene.

La herramienta analiza esta estructura en busca de archivos TQL, procesando los mismos y generando las clases con los métodos correspondientes, agregándolas al proyecto indicado también como parámetro, dejando disponibles las nuevas operaciones al usuario sin necesidad de ejecutar ningún paso más.

Para ejecutar la herramienta se dispone de un script parametrizable, el cual es fácilmente configurable en caso de trabajar sobre un mismo proyecto por un tiempo prolongado, permitiendo así al usuario

generar progresivamente su capa de acceso a datos simplemente creando archivos TQL y ejecutando el script.

En cuanto a la herramienta de Análisis de Impacto, como hemos mencionado previamente su objetivo es realizar consultas para conocer los usos de determinados elementos de la base de datos por parte de las consultas disponibles en la DAL.

Los datos necesarios para el uso de esta herramienta se encuentran en su totalidad en una base de datos relacional, lo cual la integra perfectamente al entorno de desarrollo, dado que mediante el uso de un gestor de base de datos, se pueden realizar todas las tareas necesarias.

Respecto a la herramienta de Validación Sistemática, se ofrece una interface simple, que permite al usuario indicando solamente la ubicación de la base de datos de la metadata y la base de datos contra la cual se desea ejecutar la validación, obtener los resultados de la validación de forma gráfica, con la información necesaria para detectar las diferencias entre ambos modelos.

3.6. Conclusiones

Teniendo en claro las características y el funcionamiento de las herramientas desarrolladas, nos encontramos en condiciones de analizar de que forma cumplen con las necesidades planteadas en el capitulo anterior.

Estas herramientas poseen características de distintos paradigmas, logrando de ésta manera un híbrido que consigue abarcar las necesidades plantadas.

A continuación listamos las principales características brindando una breve descripción de las mismas.

- Performance de las consultas
  Logrado mediante el uso directo de SQL, ofreciendo de esta manera un manejo óptimo desde el punto de vista de la performance.

- Validación de consultas en tiempo de compilación
  El usuario a la hora de compilar puede ejecutar el script de generación, el cual regenera la capa de acceso a datos de acuerdo al contenido de la base de datos de metadata y los archivos TQL correspondientes, de esta manera se validan las consultas previamente, para evitar errores en tiempo de ejecución.

  En caso de tener dudas de la correspondencia de la versión del modelo en la metadata con la versión de la base de datos a utilizar, el usuario puede ejecutar la validación sistemática.

- Expresividad al escribir las consultas
  Logrado también mediante el uso directo de SQL.

- Facilidad en el mantenimiento y uniformidad del código de la capa de acceso a datos
  Dado que el código correspondiente a la capa de acceso a datos es generado en base a plantillas, y almacenado en un proyecto determinado de la solución sobre la cual se trabaja, se logran estos dos objetivos claramente.

  Esto da lugar también a la centralización de las consultas.

- Generación de código de la capa de acceso a datos
  Principal característica de la herramienta Generator.

- Análisis de impacto ante cambios en la base de datos
  Logrado mediante la disponibilidad de una base de datos relacional , con todos los datos necesarios para realizar los análisis de impacto, pudiendo ser consultada mediante el uso de SQL.

- Control sistemático de consultas SQL contra una base Oracle [2]

  Logrado mediante una herramienta específica, que fácilmente realiza la comparación de los elementos del modelo de datos alojado en la base de datos utilizada por la aplicación y la metadata utilizada por las herramientas desarrolladas.

```
<Data>
    <Queries>
        <Query name="UpdateContractNumberById">
            <Source>
                <![CDATA[UPDATE
                            CT_CONTRACT
                        SET
                            CONTRACT_NUMBER=100
                        WHERE
                            ID_CONTRACT=:contrato]]>
            </Source>
        </Query>
        <Query name="GetContractIdByContractNumber">
            <Source>
                <![CDATA[SELECT
                            C.ID_CONTRACT, CO.ID_COMPANY
                        FROM
                            CT_CONTRACT C, CT_COMPANY CO
                        WHERE
                            C.CONTRACT_NUMBER:=contract_number
                            AND C.ID_COMPANY=CO.ID_COMPANY]]>
            </Source>
        </Query>
        <Query name="GetNextSequence">
            <Source>
                <![CDATA[SELECT
                            SRD_SERVER.NEXTVAL
                        FROM
                            DUAL]]>
            </Source>
        </Query>
    </Queries>
</Data>
```

Figura 3.5: Ejemplo de sección Queries de archivo TQL

Figura 3.6: Resultado de ejecución de Generator

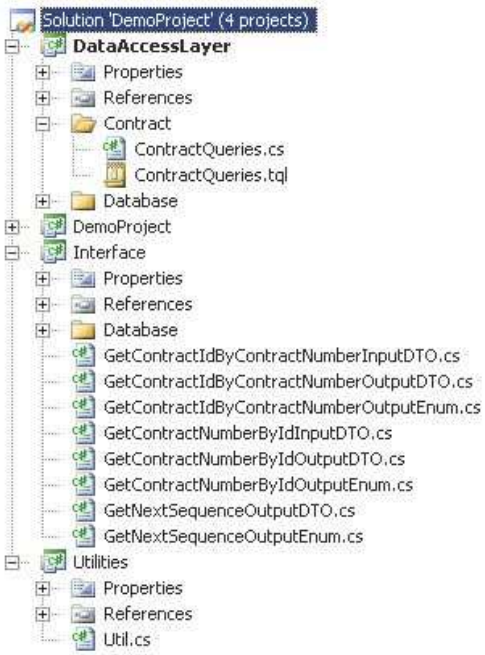

Figura 3.7: Archivos generados por Generator

```
<Rules>
    <Rule name="TestRule">
        <Appliance>
            <![CDATA[SELECT IDQUERY FROM QM_QUERY WHERE
                    QUERYNAME = 'GetContractIdByContractNumber';]]>
        </Appliance>
        <Filter>
            <Parameters>
                <Parameter Name="maxRows" NameSpace="Utilities"
                    Method="GetMaxRows" Class="Util" Module="BC"/>
            </Parameters>
            <Code>
                <![CDATA[ROWNUM <= :maxRows]]>
            </Code>
        </Filter>
    </Rule>
    <Rule name="TestRule2">
        <Appliance>
            <![CDATA[SELECT IDQUERY FROM QM_QUERY WHERE
                    QUERYNAME = 'GetContractNumberById';]]>
        </Appliance>
        <Validations>
            <Validation Type="PRE"  NameSpace="Paytrue.PayStudio"
            Method="ValidateAmountPre" Class="Checks" Module="CS" />
            <Validation Type="POST" NameSpace="Paytrue.PayStudio"
            Method="ValidateAmountPost" Class="Checks" Module="CS" />
            <Validation Type="BOTH" NameSpace="Paytrue.PayStudio"
            Method="ValidateAmountBoth" Class="Checks" Module="CS" />
        </Validations>
    </Rule>
</Rules>
```

Figura 3.8: Ejemplo de sección Rules de archivo TQL

Capítulo 4

Caso de estudio

El objetivo de este capítulo es desarrollar un breve caso de estudio, donde mostraremos la aplicación de cada una de las herramientas desarrolladas y la forma en que estas resuelven los requisitos planteados por el cliente.

4.1. Presentación del caso de estudio

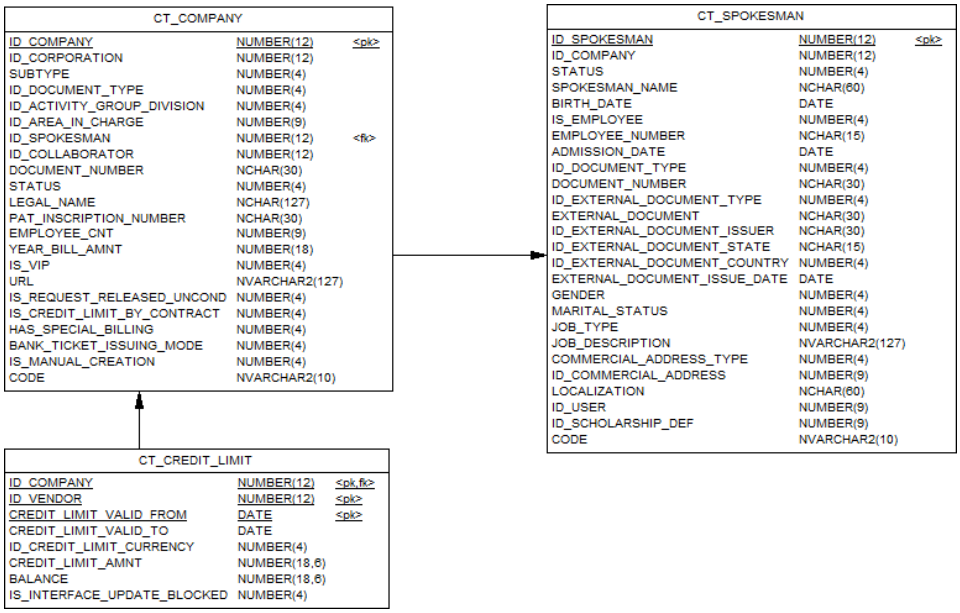

Figura 4.1: Modelo de datos del caso de estudio

Utilizaremos un extracto de un modelo de datos perteneciente a uno de los sistemas desarrollados por el cliente y tomaremos consultas SQL incluídas en la DAL de este mismo sistema, con el fin de contrastar el desarrollo actual con el propuesto por el presente trabajo. La figura 4.1 muestra un extracto del modelo de datos utilizado.

4.1.1. Consultas

Se trabajará sobre las siguientes consultas SQL:

c1
```
SELECT CT_COMPANY.ID_CORPORATION
FROM
CT_SPOKESMAN INNER JOIN CT_COMPANY ON
CT_SPOKESMAN.ID_COMPANY = CT_COMPANY.ID_COMPANY
WHERE
```

```
CT_SPOKESMAN.ID_SPOKESMAN = :paramSpokesmanId
```

c2 INSERT
INTO
CT_COMPANY
(ID_COMPANY, ID_CORPORATION, SUBTYPE, ID_DOCUMENT_TYPE,
DOCUMENT_NUMBER, STATUS,LEGAL_NAME,
PAT_INSCRIPTION_NUMBER,
EMPLOYEE_CNT, YEAR_BILL_AMNT,
IS_VIP, URL, IS_REQUEST_
RELEASED_UNCOND, IS_CREDIT_LIMIT_BY_CONTRACT,
ID_ACTIVITY_GROUP_DIVISION) VALUES
(:id_company, :id_corporation, :subtype,
:id_documenty_type, :document_number,
:status, :legal_name, :pat_inscription_number,
:employee_cnt, :year_bill_amnt,
:is_vip, :url, :id_activity_group_division)

4.2. Generación de código

4.2.1. Metodología actual

El paradigma seleccionado por *PayTrue Solutions* para el sistema analizado en este caso de estudio es Acceso directo a base de datos [2.2.1]. A continuación se presentan los pasos que cada desarrollador debe seguir para agregar una consulta a la DAL.

a. Escribir las consultas en un editor SQL
   Este paso no es obligatorio, pero le permite al desarrollador construir y testear la consulta en forma directa.

b. Transcribir la consulta a *C#*

c. Testear la consulta incorporada a la DAL en el paso b

El paso b es el que más tiempo de desarrollo consume y es también un punto de inyección de errores, se estima que un 70 % del tiempo

de desarrollo de una consulta se consume en este paso. A pesar de que la consulta escrita en el paso a esté sintáctica y semánticamente correcta, la transcripción a *C#* suele introducir sistemáticamente diferentes tipos de errores, basta observar la figura 4.2 para entender este punto.

```
public long GetCorporationIdOfSpokesman(long spokesmanId)
{
    StringBuilder s = new StringBuilder();
    IDbConnection connection = Connection;

    s.Append(" SELECT ");
    s.Append(string.Format("{0}.{1}", Tables.CT_COMPANY.ToString(), CT_COMPANY.ID_CORPORATION.ToString()));
    s.Append(" FROM ");
    s.Append(Tables.CT_SPOKESMAN.ToString());
    s.Append(" INNER JOIN ");
    s.Append(Tables.CT_COMPANY.ToString());
    s.Append(" ON ");
    s.Append(string.Format("{0}.{1}", Tables.CT_SPOKESMAN.ToString(), CT_SPOKESMAN.ID_COMPANY.ToString()));
    s.Append(" = ");
    s.Append(string.Format("{0}.{1}", Tables.CT_COMPANY.ToString(), CT_COMPANY.ID_COMPANY.ToString()));
    s.Append(" WHERE ");
    s.Append(string.Format("{0}.{1}", Tables.CT_SPOKESMAN.ToString(), CT_SPOKESMAN.ID_SPOKESMAN.ToString()));
    s.Append(" = :paramSpokesmanId");

    // Adjuntar Parametros de la consulta
    OracleParameter p1 = new OracleParameter();
    p1.OracleDbType = OracleDbType.Decimal;
    p1.Value = spokesmanId;
    p1.ParameterName = "paramSpokesmanId";

    OracleCommand oracleCommand = new OracleCommand(s.ToString(), (OracleConnection) connection);
    oracleCommand.Parameters.Add(p1);
    object result = oracleCommand.ExecuteScalar();
    return Convert.ToInt64(result);
}
```

Figura 4.2: Código *C#* de una consulta SQL

4.2.2. Generator

Utilizando el Generator, la metodología de trabajo se verá sensiblemente alterada. Los pasos que el desarrollador deberá seguir son los siguientes:

a. Escribir las consultas en un editor SQL

b. Transcribir las consultas al archivo tql correspondiente

c. Invocar el Generator

Es importante destacar la diferencia sustancial de esfuerzo que radica el paso b respecto al mismo paso de la metodología actual de trabajo de *PayTrue Solutions* . Utilizando el Generator, el paso b se reduce a un simple copy/paste de la consulta, mientras que en el paso b anterior implica el desarrollo de un método *C#* . El paso c de la metodología actual de trabajo dejaría de tener tanto sentido al utilizar el Generator, ya que la consulta será validada semántica y sintácticamente en tiempo de compilación. La figura 4.3 muestra un archivo tql conteniendo las consultas del caso de estudio. La implementación de algunas de los futuros trabajos analizados en 5.2 permitiría modificar esta metodología de trabajo y reducirla a únicamente 2 pasos. Si se logra embeber el Generator dentro del entorno de desarrollo de Visual Studio a modo de AddIn, la generación de código sería transparente para el desarrollador.

```
<Data>
    <Queries>
        <Query name="GetCorporationIdOfSpokesman">
            <Source>
                <![CDATA[SELECT CT_COMPANY.ID_CORPORATION
                        FROM
                            CT_SPOKESMAN  INNER JOIN CT_COMPANY ON CT_SPOKESMAN.ID_COMPANY = CT_COMPANY.ID_COMPANY
                        WHERE
                            CT_SPOKESMAN.ID_SPOKESMAN = :paramSpokesmanId]]>
            </Source>
        </Query>
        <Query name="AddCompany">
            <Source><![CDATA[INSERT INTO
                                CT_COMPANY
                                    (ID_COMPANY, ID_CORPORATION, SUBTYPE, ID_DOCUMENT_TYPE, DOCUMENT_NUMBER, STATUS,LEGAL_NAME, PAT_INSCRIPTION_NUMBER,
                                    EMPLOYEE_CNT, YEAR_BILL_AMNT, IS_VIP, URL, IS_REQUEST_RELEASED_UNCOND, IS_CREDIT_LIMIT_BY_CONTRACT,
                                    ID_ACTIVITY_GROUP_DIVISION)
                                VALUES
                                    (:id_company, :id_corporation, :subtype, :id_documenty_type, :document_number, :status, :legal_name,
                                    :pat_inscription_number, :employee_cnt, :year_bill_amnt, :is_vip, :url, :id_activity_group_division)]]>
            </Source>
        </Query>
    </Queries>
</Data>
```

Figura 4.3: Archivo TQL del caso de estudio

Cabe destacar la notable disminución en el tiempo de desarrollo de las consultas que se logró al utilizar el Generator. El desarrollador debe concentrarse únicamente en la implementación de la consulta SQL (paso a de ambas metodologías), cosa que es en definitiva la carga útil del trabajo que requiere la incorporación de una consulta a la DAL del sistema.

4.3. Análisis de impacto

El análisis de impacto respecto a cambios en el modelo de datos, es una actividad que consume una cantidad considerable de tiempo al equipo de desarrollo de *PayTrue Solutions* , fundamentalmente cuando el desarrollo de la aplicación se encuentra en etapas avanzadas. Supongamos que debido a cambios en las reglas de negocio, el campo `CT_COMPANY. ID_AREA_IN_CHARGE` cambia de NULL a NOT NULL y no es definido ningún valor por default. Este tipo de alteraciones de la base de datos, vinculado fuertemente a la calidad de datos, cuando por razones de negocio es necesario almacenar datos que originalmente no se creían necesarios, es bastante habitual es etapas maduras del proyecto. Una alteración estructural de estas características afectará directamente a las sentencias del tipo UPDATE e INSERT, en particular la consulta c1 generará un error en tiempo de ejecución debido a que este campo no es alimentado por ningún parámetro.

4.3.1. Metodología actual

Ante un cambio en el modelo de datos, es necesario efectuar una revisión de cada una de las consultas de la DAL con el fin de detectar cuales están utilizando el objeto afectado. Dada la naturaleza del código de la DAL, la tarea de análisis de convierte en engorrosa y poco precisa.
El uso de enumerados para representar las tablas, vistas, columnas y sequences (ver imagen 4.2) son una ayuda importante para cumplir con el objetivo , ya que bastaría determinar los usos de estos enumerados en el código, pero aquí surgen dos problemas: la arquitectura modular de las aplicaciones de *PayTrue Solutions* y

la ausencia de la metodología de enumerados en muchas consultas del sistema. El primer inconveniente se puede explicar brevemente, los enumerados son definidos en un único módulo el cual puede ser referenciado por otros si es que necesitan acceder a los objetos de la base de datos del primero, esto dificulta en sobremanera buscar referencias al enumerado en los diferentes módulos, recordemos que cada módulo se expresa en una solución de Microsoft Visual Studio. Por estos motivos, la metodología de análisis de impacto utilizada es costosa en término de horas/hombre, este tiempo se divide en dos, el utilizado para efectuar el análisis y el utilizado para efecutar los test de regresión, esto último se debe a que el análisis no es 100 % preciso a causa de los factores descriptos.

4.3.2. Consultar la metadata de consultas

La metadata de consultas, producida por el Generator, contiene la información necesaria para efectuar el análisis de impacto. Se presenta a continuación la consulta SQL que es necesario efectuar sobre la base de datos de la metadata, para determinar que consultas del sistema (del tipo UPATE e INSERT) se vieron afectadas por el cambio en la columna `ID_AREA_IN_CHARGE`.

```
SELECT Q.QUERYNAME, QG.FILENAME
FROM QM_QUERY Q, MM_OBJECT O, QM_QUERYOBJECT QO, QM_GENERATION
QG
WHERE Q.IDQUERY=QO.IDQUERY AND QO.IDOBJECT=O.IDOBJECT
AND Q.IDGENERATION=QG.IDGENERATION
AND OBJECTNAME='ID_AREA_IN_CHARGE'
AND QUERYTYPE IN (2,3);
```

El retorno de esta consulta son el nombre de las consultas SQL y los nombres de los archivos TQL que contienen las mismas, información suficiente para cubrir el análisis de impacto.

4.4. Verificación sistemática de consultas

En etapas de un proyecto de software donde los cambios en el modelo de datos son relativamente habituales, pueden existir diferencias de versión entre el código de la DAL y la base de datos sobre la cual se ejecutará el sistema. Específicamente concentrémonos en el cambio presentado en el punto anterior, la alteración de la columna ID_AREA_IN_CHARGE.

4.4.1. Metodología actual

Ante cualquier duda ante posibles diferencias de versión entre la DAL y la base de datos subyacente, no existe un mecanismo formal definido por parte del cliente para atacar este problema. Existen diversas alternativas, tales como chequear documentos de cambios del modelo de datos (en caso de que existan), contactar a las personas que tienen acceso para modificar el modelo de datos con el fin de recabar información sobre posibles cambios en la base de datos, chequear los últimos cambios en la DAL y verificar que hayan sido impactos en la base de datos o directamente un test de regresión. Cualquiera de las alternativas tratadas pueden llegar a ser poco precisas y costosas en término de tiempo, por lo que muchas veces este tipo de problemas se detectan recién en tiempo de ejecución. Particularmente en el caso de la columna ID_AREA_IN_CHARGE, se generará un error al momento de ejecutar sentencias del tipo UPDATE o INSERT sobre la tabla CT_COMPANY.

4.4.2. Herramienta de verificación sistemática de consultas

El Verificador Sistemático de Consultas permite detectar todas aquellas consultas que se verán afectadas ante cambios en la base de datos, este test se basa fundamentalmente en la comparación de la metadata del modelo contra el catálogo Oracle de la base de datos subyacente a la DAL. El resultado de la ejecución de esta herramienta será un reporte indicando que la consulta c2 dejará de funcionar y una sintética explicación del problema.

Capítulo 5

Conclusiones y Trabajo a Futuro

5.1. Conclusiones

A partir de un exhaustivo análisis de las necesidades planteadas por el cliente y de las herramientas disponibles en el mercado vinculadas al desarrollo de la DAL, se concluyó que era necesario el desarrollo de nuevas herramientas para satisfacer los requisitos presentados.

Las dos herramientas desarrolladas, descriptas en detalle en el capítulo 3, se adhieren perfectamente a los requisitos planteados por el cliente. Estas herramientas se adaptaron a las necesidades específicas del cliente pero sin perder de vista una posible futura generalización de las capacidades de las mismas. En particular, la herramienta Generator trabaja sobre bases de datos Oracle [2] y genera código *C#* , pero la arquitectura de la misma permite que la gramática interpretada pueda ser modificada para soportar nuevas extensiones de este DBMS así como también soportar otras gramáticas, tales como SQL Server, DB2, PostgreSQL, etc. Al mismo tiempo, el generador de código puede ser fácilmente modificado para que produzca código de otros lenguajes de programación, gracias al uso de templates.

Cabe hacer en este punto una breve reseña de los problemas que el cliente enfrenta, así como también que solución fue ofrecida para cada uno de ellos.

5.1.1. Complejo mantenimiento de la DAL

El mantenimiento de la DAL se planteó como uno de los principales problemas del cliente. Incluir una nueva consulta en los sistemas de cliente implica un costo en horas/hombre relativamente alto, así como también la modificación de consultas existentes. Esto se debe fundamentalmente al paradigma de desarrollo utilizado por el cliente, lo cual acarrea que el código de la DAL sea extremadamente engorroso. La herramienta Generator ofrece una solución a este problema, permitiendo que el programador se enfoque únicamente en el desarrollo de la consulta SQL, automatizando la generación de código y efectuando una validación sintáctica y semántica de la consulta en tiempo de compilación.

5.1.2. Código de la DAL heterogéneo

La forma de desarrollo de la DAL que utiliza el cliente puede provocar que el código de la DAL termine siendo heterogéneo. Esto sucede especialmente en proyectos de gran porte donde participan decenas de desarrolladores, estas condiciones propician la gradual degradación estructural del código de la DAL. La herramienta Generator es una solución a esta problemática. La forma de desarrollo que implica el uso de esta herramienta, obliga al desarrollador a escribir las consultas en archivos TQL sin necesidad de tocar el código generado, de esta forma el código de la DAL será homogéneo y además estará centralizado.

5.1.3. Costoso análisis de impacto

El cliente ha planteado la problemática de que en etapas maduras de los proyectos que encara le es altamente costoso efectuar un análisis de impacto de cambios en el modelo de datos. La metadata generada por la herramienta Generator permite efecutar análisis de impacto por medio de simples consultas SQL, esto reduce sensiblemente el costo de medir que tanto afecta al sistema un cambio en el modelo de datos.

5.1.4. Disociación entre la DAL y el modelo de datos

El modelo de datos, mantenido por el cliente con la herramienta QDesigner, no tiene ningún punto de contacto tangible con la DAL, esto implica que puedan ser escritas consultas que referencien a objetos erróneos de la base de datos. La herramienta Generator valida en tiempo de compilación que los objetos a los que hacen referencia cada una de las consultas de la DAL pertenezcan al modelo de datos. Para esto, parsea en primer lugar cada unos de los scripts SQL DDL del modelo de datos, sobre el resultado de este proceso validará cada una de las consultas.

5.2. Trabajo a Futuro

Existen diversos aspectos a ser mejorados y algunas nuevas funcionalidades que han quedado por fuera del alcance de este proyecto. El objetivo de esta sección es enumerar y describir estos puntos para cada una de las herramientas desarrolladas.

5.2.1. Generator

Procesamiento de PL/SQL

El procesamiento de código PL/SQL enriquecería el análisis de impacto y la verificación sistemática de consultas. Por ejemplo, sería posible detectar problemas en la invocación de stored procedures cuando existieran inconsistencias entre los objetos referenciados por el mismo y el catálogo de Oracle [2].

Optimización de consultas

Consideramos una funcionalidad interesante la optimización de consultas en tiempo de ejecución. Por medio del uso de reglas de inyección de código sería posible la invocación de un método de optimización antes de la ejecución de las consultas. Utilizando la API runtime del Generator, este método podría tener acceso tanto a la

metadata del modelo como a las utilidades de parseo, las cuales permiten generar un AST [13] a partir del string de la consulta SQL.

Esta funcionalidad permitiría efectuar ciertas optimizaciones sobre la consulta, tal como la predicción del resultado de la misma sin necesidad de que sea enviada al DBMS. Para aclarar este punto analicemos el siguiente ejemplo, supongamos que se cuenta con una tabla Customer con una columna Age la cual contiene un check que restringe su dominio a los valores del 0 al 100. Cualquier consulta del tipo SELECT * FROM Customer WHERE age>100 retornaría cero tuplas. El Generator posee la información y las herramientas como para detectar este tipo de situaciones antes de que la consulta sea ejecutada.

Interpretación de sentencias SQL de múltiples DBMS

El Generator se adapta específicamente a las necesidades de *PayTrue Solutions* [1], la gramática SQL está basada en el lenguaje provisto por la base de datos Oracle [2], por lo que solo es posible procesar sentencias DML y DDL escritas para este motor. A pesar de que un alto porcentaje de esta gramática pertenece al estándar y por lo tanto también está incluida en otros DBMS, es claro que cada versión tiene sus propias características y es deseable que sean interpretadas por el Generator. Creemos interesante considerar las bases de datos Microsoft SQL Server [3], MySQL [4] y PostgreSQL [6], esto expandiría sensiblemente el campo de aplicación de la herramienta.

Procesamiento parcial de scripts de esquemas y consultas

Habitualmente, cuando se invoca la generación de código, únicamente se han modificado algunos pocos archivos TQL y/o archivos SQL, a pesar de esto el Generator genera código y metadata para todos los archivos de la solución.

Una posible mejora consistiría en almacenar una firma hash MD5 de cada archivo en la base de datos de la metadata. Solo serían procesados aquellos archivos cuya firma hash difiera de la almacenada en la metadata, esto aceleraría notablemente el proceso de generación.

Add-in para Visual Studio

Los add-in son una buena forma de integrar funcionalidades dentro del entorno de desarrollo. Embeber el Generator dentro de Visual Studio redunda en un aumento significativo de la calidad de usabilidad de la herramienta.

A groso modo, la idea es incluir una nueva opción en el menú pop-up de la ventana de edición de código. Si el archivo que el desarrollador se encuentra editando es TQL, está opción quedará habilitada y por medio de dos clics podrá disparar la generación de código, sin necesidad de abandonar el entorno de desarrollo para ejecutar un script.

Apéndice A

Arquitectura de la Herramienta Generator

El diseño se alinea a la arquitectura actual de software del cliente, en particular aplicando el enfoque de componentes. Los mismos quedan claramente definidos por sus funcionalidades. En la figura A.1 se puede apreciar el diagrama de componentes.

En este anexo se proporciona una descripción de cada componente y su ubicación dentro del flujo de ejecución de la herramienta.

A.1. Main Module

Componente principal, encargado del flujo del proceso de generación de código. Recibe los parámetros correspondientes, y realiza las llamadas a los módulos encargados de cada etapa para cumplir con los objetivos de la herramienta. En primer lugar es el encargado de obtener los parámetros de entrada, una vez validados los mismos, procede a iniciar la Etapa I – Generación de metadata de modelo.

Para iniciar la generación de la metadata, realiza la llamada al módulo DDL Parser, enviándole la ruta en la cual se encuentran los scripts dll para que extraiga la metadata de los mismos.

El resultado de la generación de la metadata es almacenado en un dataset, el cual será utilizado en las distintas etapas, por los distintos

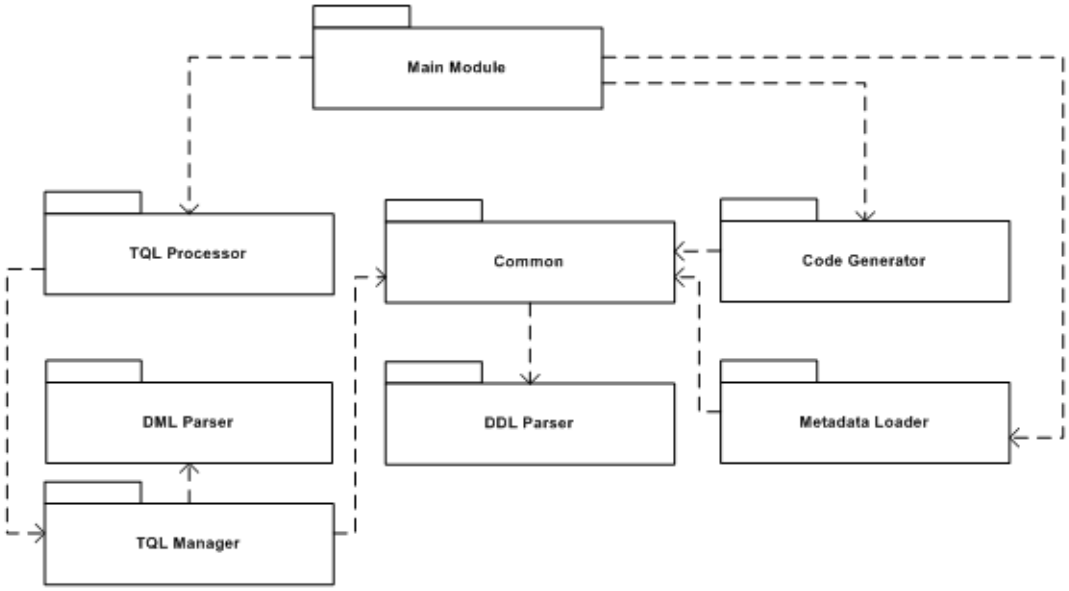

Figura A.1: Arquitectura: Diagrama de componentes

componentes, para agregar y actualizar los datos que sean necesarios. Teniendo la metadata del modelo cargada en el dataset, pasa a la Etapa II – Generación de Código DDL, llamando al Componente Code Generator, pasando el dataset con la metadata cargada y la ruta donde debe crear los archivos. Una vez creados los archivos agrega los mismos al archivo de proyecto y se pasa a la siguiente etapa.

Teniendo la metadata del modelo y su código ya generado se pasa a la Etapa III – Generación de metadata de consultas, llamando al Componente DML ParserA.7. La metadata de las consultas es agregada al mismo dataset que contenía la metadata del modelo, haciendo referencia a los objetos que utiliza cada consulta del modelo de datos.

Las etapas IV, V y VI se llevan a cabo dentro de otros componentes, siendo referenciadas en el detalle de los mismos. Finalmente se procede a persistir la metadata en la base de datos embebida, dejándola así disponible para su uso.

A.2. Metadata Loader

Este componente no forma parte del flujo de ejecución de Generator, ha sido creado para proporcionar una interfaz visual para poder ejecutar la carga de metadata apreciando fácilmente el progreso de la carga y facilitando la indicación de la ruta donde se encuentran los scripts DDL.

Encargado de la carga inicial de la metadata, a partir de los scripts DDL de la base de datos correspondiente, utiliza el componente DDL Parser[A.4] para interpretar dichos scripts.

Esta carga se realiza en primera instancia sobre un dataset y posteriormente es persistida en la base de datos embebida. En resumen, este componente facilita la generación de una base de datos embebida, con la metadata del modelo de datos.

A.3. Common

En este componente se centralizan las definiciones de enumerados, datatypes e interfaces globales utilizados por los demás componentes, así como también el acceso a la base de datos de la metadata. Las operaciones comunes de acceso a la base de datos embebida, se encuentran en este componente, el cual oficia como componente de definiciones y de acceso a datos.

A.4. DDL Parser

Este componente es el encargado de interpretar el texto correspondiente a un script DDL, validándolo sintácticamente. Se basa en la gramática TQL DDL para realizar su tarea. A medida que detecta los distintos elementos que conforman la base de datos, utiliza el componente Common para insertar los mismos en la base de datos de la metadata.

Esto es posible dado que *ANTLR* permite agregar código *C#* dentro de las definiciones de las reglas, consiguiendo de esta manera,

almacenar en el dataset los datos de los statements a medida que se van encontrando. Dado que los scripts DDL son generados por una herramienta, no es necesario validarlos semánticamente.

A.5. TQL Processor

De la misma manera que el componente Metadata Loader facilita la carga de metadata del modelo en una base de datos embebida, este componente facilita el procesamiento de archivos TQL.

Este componente no forma parte del flujo de ejecución de Generator, ha sido creado para proporcionar una interfaz visual para poder ejecutar el procesamiento de archivos TQL, apreciando fácilmente la evolución del mismo y facilitando la indicación de la ruta donde se encuentran los scripts TQL. Procesa un determinado directorio en busca de archivos TQL, de los cuales extrae tanto las consultas como las reglas.

En primer lugar utiliza el componente DMLParser para interpretar las consultas encontradas, utilizando el componente TQL Manager para almacenar la información extraída, luego procede a aplicar las reglas a la información que mantiene el TQLManager.

Posteriormente obtiene las consultas con las reglas ya aplicadas y las ejecuta nuevamente contra el DMLParser para validar su estructura, posteriormente validando la semántica mediante el TQLManager. Como paso final persiste los datos de las consultas en la base de datos embebida utilizando el componente Common.

A.6. TQL Manager

Componente utilizado para mantener toda la información de las consultas y reglas aplicadas a las consultas de varios archivos TQL. Ofrece también la funcionalidad de validar cada consulta semánticamente. En este componente se definen las estructuras que son utilizadas para almacenar temporalmente los datos de las consultas para poder realizar fácilmente su validación.

Se define una estructura para cada tipo de statement, pudiendo anidarse entre ellos ilimitadamente, tal cual se pueden anidar statements en un script SQL. Estas estructuras además de almacenar los datos básicos del statement correspondiente, almacenan los datos específicos de las columnas y valores utilizados en el mismo, así como también parámetros y parámetros opcionales. Con toda esta información es posible realizar una validación semántica avanzada.

A.7. DML Parser

Este componente es el encargado de interpretar el texto correspondiente a un script DML, validándolo sintácticamente. Se basa en la gramática TQL DML para realizar su tarea. A medida que detecta los distintos elementos que conforman la consulta, utiliza el componente TQLManager para almacenar los datos correspondientes.

Aprovechando la recursión implícita en el parseo de los scripts y el hecho de que *ANTLR* permite agregar código *C#* dentro de las reglas que se ejecuta cada vez que se detecta la misma, se genera la estructura mencionada en el componente TQL Manager.

Una vez generada la metadata de las consultas, persiste la información de las mismas en la base de datos embebida, para posteriormente aplicar las reglas que ha encontrado en los arhivos TQL, esto corresponde con la Etapa IV – Aplicación de reglas.

Ejecutando los consultas indicadas en las reglas, contra la base de datos de la metadata, obtiene las consultas afectadas por cada regla, luego de aplicar las mismas sobre dichas consutlas, procede a actualizar su metadata, lo cual corresponde con la Etapa V – Actualización de metadata de consultas. Finalmente llama al Componente Code Generator para generar el código de las consultas.

A.8. Code Generator

Este componente es el encargado de generar el código de las consultas, las clases y los tipos de datos que las mismas utilizan, así como

también de las clases que representan los objetos del modelo de datos. Esto implica que sea utilizado en dos instancias, la primera para la generación de código correspondiente a los scripts ddl, y la segunda para la genración de código correspondiente a los archivos TQL.

En el caso de la generación DDL, Etapa II – Generación de Código DDL, recibe el dataset con los datos de la metadata del modelo y procede a generar el código que representa los objetos de dicho modelo en archivos ubicados en la ruta indicada.

En el caso de la generación DML (scripts ubicados dentro de archivos TQL), Etapa VI – Genración de código de consultas, recibe los datos de las consultas, generando las clases en los archivos ubicados en el directorio relativo respecto a la ruta del proyecto Interface indicada, modificando el archivo de proyecto DataAccessLayer indicado, para agregar los archivos generados.

Apéndice B

Estructura de la Metadata

B.1. Introducción

En este anexo analizaremos en detalle la estructura de la base de datos de la metadata del sistema.

La metadata se puede dividir claramente en dos, la del modelo, cargada en un inicio a partir de los scripts DDL correspondientes a la creación de la base de datos, y la de las consultas, generada a partir de la interpretación de los archivos TQL de los usuarios.

B.2. Modelo

En lo que respecta al modelo, se almacenan los datos de los elementos más relevantes. Estos elementos son considerados como objetos, por lo cual en el modelo se ha generado una tabla base, denominada MM_Object, la cual posee los datos en común que se manejan para todos los objetos.

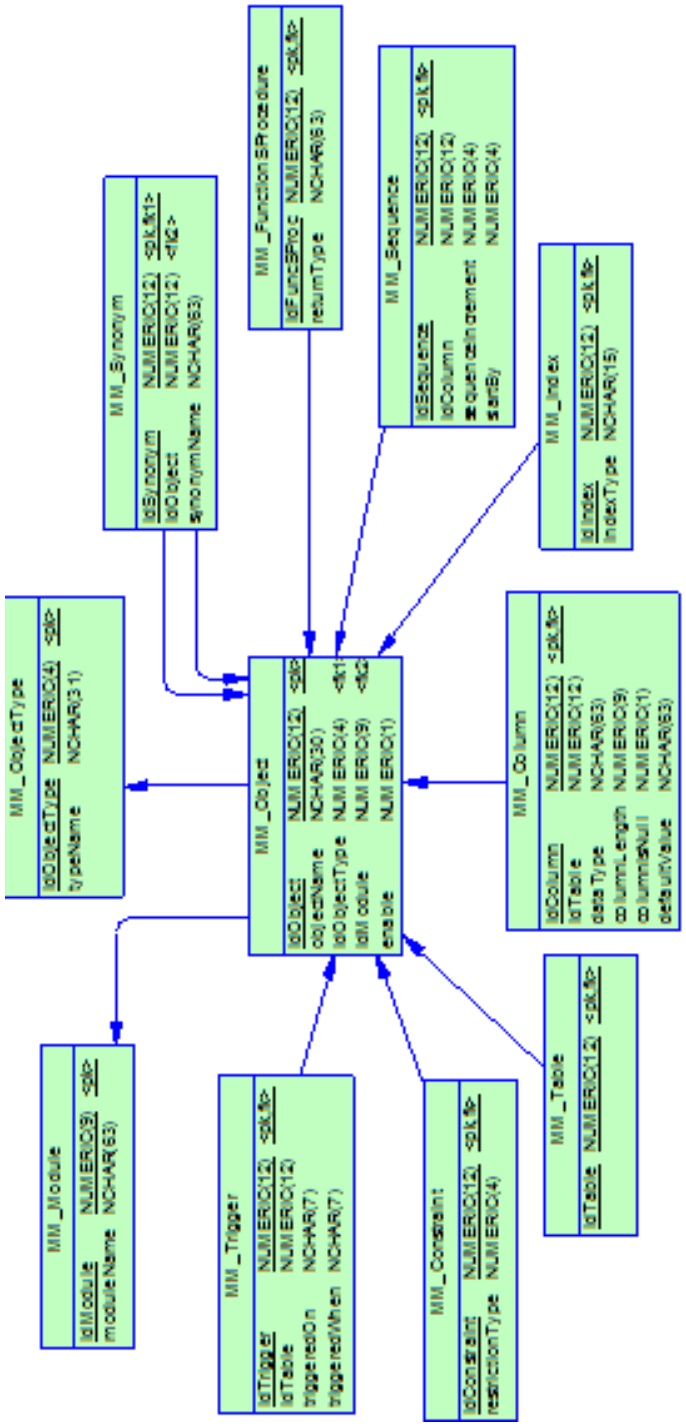

Figura B.1: Metadata del Modelo

Los datos que se almacenan en MM_Object son:

- objectName
  Corresponde con el nombre del objeto.
- idObjectType
  Indica el tipo del objeto, los distintos tipos son almacenados en la tabla MM_ObjectType.
- idModule
  Indica el módulo al cual corresponde el objeto, los módulos se almacenan en la tabla MM_Module.

En los siguientes puntos se analizan los datos manejados de los distintos tipos de objetos soportados.

B.2.1. Tablas

Los datos básicos de las tablas son los mismos que se almacenan para los objetos, como puede verse, se ha generado una estructura de herencia por lo cual solamente se almacena en la tabla MM_Table el idTable que es una foreign key a la columna IdObject de la tabla MM_Object.

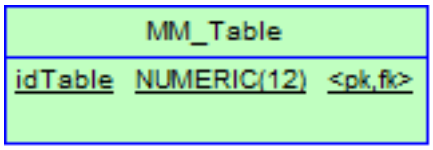

Figura B.2: Tabla

Columnas

Para las columnas, además de los datos básicos almacenados en la tabla MM_ Object, se almacenan en la tabla MM_Column:

- idTable
  Foreign key para determinar a que tabla pertenece la columna.

- dataType
  Tipo de datos de la columna, en caso de corresponderse con un enumerado, se indica el namespace y nombre para identificar el mismo.
- columnLength
  Largo de la columna.
- columnIsNull
  Indica si se permiten valores null en la columna.
- defaultValue
  Valor por defecto de la columna.

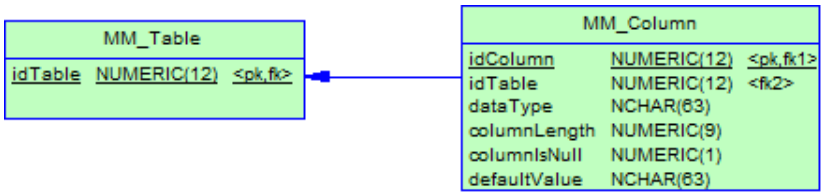

Figura B.3: Columna

B.2.2. Constraints

El modelo soporta tres tipos de constraints:

1. Primary Keys
   Para las primary keys se almacena en MM_PrimaryKey la tabla a la cual corresponde, y luego por cada columna que la conforma, se almacenan en MM_PrimaryKeyColumn, la referencia a la columna correspondiente en la tabla MM_Column.
2. Foreign Keys
   Para las foreign keys se almacena en MM_ForeignKey la tabla a la cual corresponde, y luego por cada columna que la conforma, se almacenan en MM_ForeignKeyColumn, las referencias a la columnas correspondiente en la tabla MM_Column, siendo idColumn la correspondiente a la columna de la tabla base y idForeignColumn la correspondiente a la columna de la tabla referenciada.

3. Checks
   Para los checks se almacena en MM_Check la columna a la cual corresponde y la condición que valida.

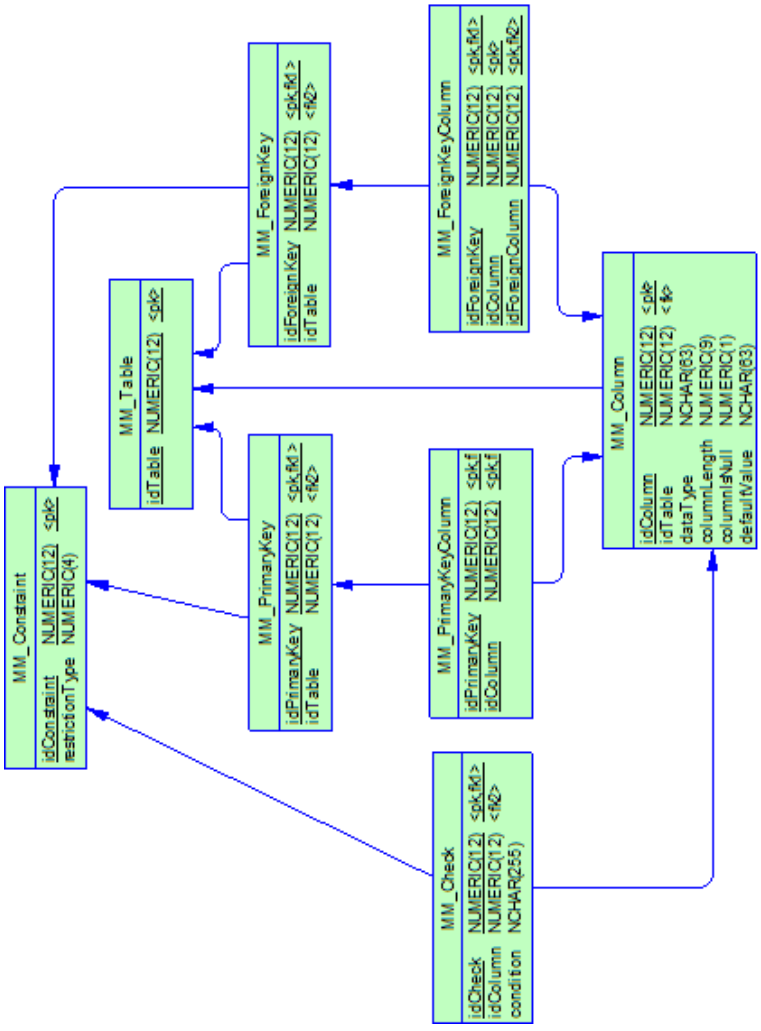

Figura B.4: Constraint

B.2.3. Triggers

Para los triggers, se almacena en MM_Trigger, la tabla a la cual corresponde y los valores de triggeredOn y triggeredWhen.

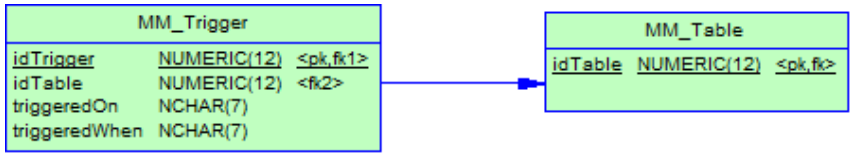

Figura B.5: Trigger

B.2.4. Secuenciadores

En el caso de los secuenciadores, se almacena en la tabla MM_Sequence, la columna a la cual corresponde, el incremento y el valor de comienzo.

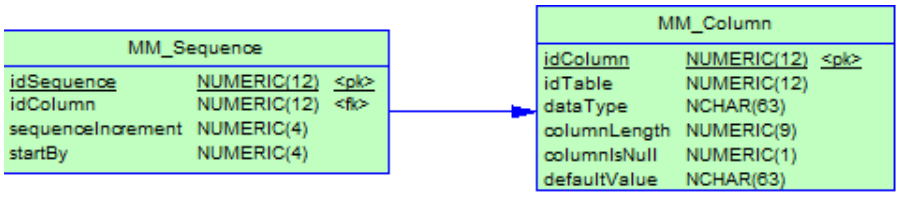

Figura B.6: Secuenciador

B.2.5. Sinónimos

En el caso de los sinónimos, se almacena en la tabla MM_Synonym el objeto al cual corresponde el sinónimo, y el nombre que se le da.

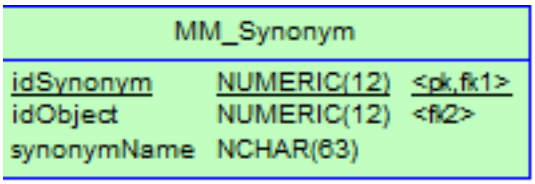

Figura B.7: Sinónimo

B.2.6. Procedimientos Almacenados y Funciones

En cuanto a los procedimientos almacenados y funciones, se almacena en la tabla MM_FunctionSProcedure el tipo de retorno si corresponde, y luego los distintos parámetros en la tabla MM_FuncSProcParameter (nombre, tipo de datos y dirección).

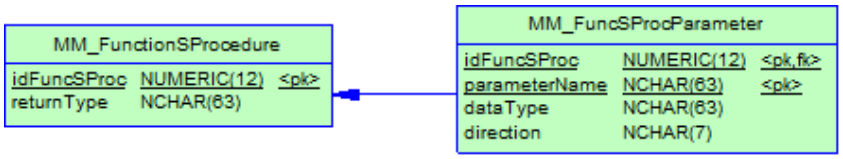

Figura B.8: Procedimiento Almacenado y Función

B.2.7. Índices

En cuanto a los índices, se almacena en la tabla MM_Index su tipo, y luego para cada columna del mismo, en la tabla MM_ColumnIndex la columna correspondiente y el tipo de orden.

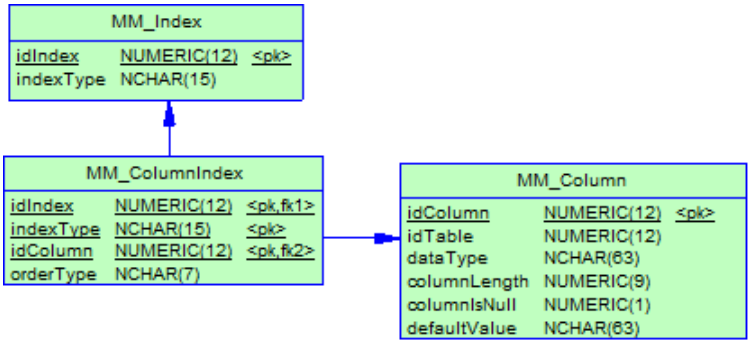

Figura B.9: Índice

B.3. Consultas

En esta sección del modelo almacena la información de las consultas, la cual se puede dividir en las siguientes secciones:

1. Objetos utilizados
   Los objetos utilizados por la consulta son almacenados en la tabla QM_QueryObject. Se almacena la referencia a la consulta correspondiente, la referencia al objeto del modelo utilizado de la tabla MM_Object, y si es de retorno o no, esto corresponde a las consultas de tipo SELECT, en las cuales interesa diferenciar entre las columnas que son retornadas por la consulta y las que son utilizadas internamente.

2. Generación de código
   En la tabla QM_Query se almacenan los datos básicos de las consultas, siendo estos el nombre y tipo de la consulta, también se almacena la referencia a la tabla QM_Generation. También se almacenan datos sobre la generación de código, como ser el módulo, namespace y nombre del archivo en el cual se ha generado el código de la consulta.

   - Métodos
     Dado que una consulta generalmente tiene varios métodos generados, para ofrecer distintas alternativas al usuario, se registran en la tabla QM_Method los datos de los mismos. Se almacena en dicha tabla la referencia a la consulta, el nombre del método, y el número de línea donde se encuentra en el archivo.

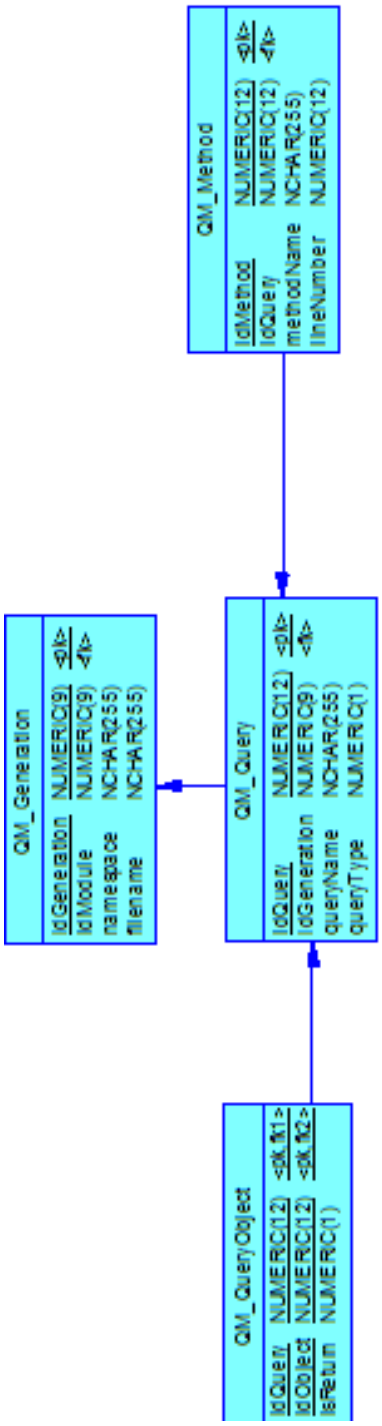

Figura B.10: Metadata de Consultas

Apéndice C

Verificación Sistemática de Consultas

C.1. Realidad Planteada

Como se introdujo en la sección 3.4, la *Verificación Sistemática de Consultas* es una herramienta que permite validar de forma fácil y rápida qué consultas son incorrectas contra una base de datos (dado que su esquema pudiera resultar diferente al referenciado por el código).

Básicamente, la aplicación consiste en retornar un reporte con los problemas detectados, a partir de la ubicación, del nombre de usuario y contraseña de la base de datos y la ubicación del archivo de la metadata del Generator, como puede verse en la Figura C.1.

La ubicación de la base de datos debe estar registrada en el archivo de configuración de Oracle [2], tnsnames.ora, en donde se especifica el host y el puerto para poder realizar la conexión.

Para el desarrollo de la herramienta, se implementaron varios tests, los cuales permiten generar el reporte resultado. Dichos tests consisten en comparaciones de los diferentes objetos, referenciados en las consultas guardadas en la metadata del Generator, contra el catálogo de Oracle, de forma de poder detectar los posibles cambios no impactados en la misma.

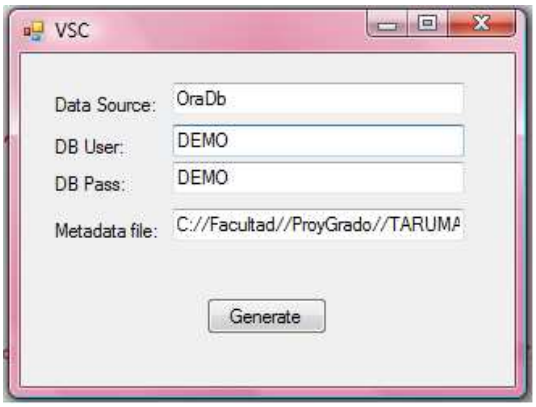

Figura C.1: Pantalla inicial de la Verificación

Los test se invocan dependiendo del tipo de consulta, es decir, si es un *INSERT*, *DELETE*, *UPDATE* o *SELECT*. A medida que se ejecutan, se les asigna una categoría, pudiendo ser Error o Warning, según si la consulta deja de funcionar o no respectivamente.

Según el tipo de consulta, se puede detectar diversos problemas. Las tablas C.1 y C.2 contienen el detalle de los mismos.

Las consultas que conforman los test planteados, se realizaron en dos modalidades:

- Consultas a la metadata del Generator: Se implementaron funciones para acceder directamente a los datos de la metadata.
- Consultas al catálogo de Oracle: Se creo un proyecto DAL incluido en la Solución de la herramienta, cuyo código es generado por la aplicación Generator.

En la figura C.2 muestra un ejemplo de archivo TQL utilizado para realizar las consultas sobre el catálogo de Oracle.

En la imagen se puede ver claramente la estructura del archivo TQL, en donde se ve la consulta de nombre *GetSequenceBySequenceName* encapsulada en el tag *Query*, y la misma se encuentra definida dentro del tag *Data*.

Luego de identificar los problemas en las consultas, se retorna un reporte en donde se especifican las fallas detectadas, dando el detalle

Problema	Categoría
No existe alguna de las tablas referenciadas.	ERROR
No existe alguna de las columnas referenciadas.	ERROR
No existe algún secuenciador referenciado.	ERROR
No existe algún sinónimo referenciado.	ERROR
Se agregó una columna nueva a alguna de las tablas referenciadas.	WARNING si la columna puede ser nula, ERROR en caso contrario.
Se agregó una columna nueva a la constraint de alguna de las tablas referenciadas.	ERROR
No existe una columna en la constraint de alguna de las tablas referenciadas.	WARNING
Se agregó un trigger en alguna de las tablas referenciadas.	WARNING
No existe algun trigger en alguna de las tablas referenciadas.	WARNING
Alguna de las columnas referenciadas tiene distinto tipo de datos.	ERROR
Alguna de las columnas referenciadas tiene distinto largo.	ERROR
Alguna de las columnas referenciadas tiene distinto el atributo nuleable.	ERROR
Alguna de las columnas referenciadas tiene distinto valor por defecto.	WARNING
Alguna de las secuencias referenciadas tiene distinto incrementador.	WARNING
Alguna de las secuencias referenciadas tiene distinto valor inicial.	WARNING

Cuadro C.1: Problemas en INSERT/UPDATE

del objeto del error, la consulta y el archivo en la que se encuentra, ordenadas por categoría. La figura C.3 muestra un ejemplo de este reporte.

Problema	Categoría
No existe alguna de las tablas referenciadas.	ERROR
No existe alguna de las columnas referenciadas.	ERROR
No existe algún secuenciador referenciado.	ERROR
No existe algún sinónimo referenciado.	ERROR
Se agregó una columna nueva a la constraint de alguna de las tablas referenciadas.	ERROR
No existe una columna en la constraint de alguna de las tablas referenciadas.	WARNING
No existe algun trigger en alguna de las tablas referenciadas.	WARNING
Alguna de las columnas referenciadas tiene distinto tipo de datos.	ERROR
Alguna de las columnas referenciadas tiene distinto largo.	ERROR
Alguna de las columnas referenciadas tiene distinto el atributo nuleable.	ERROR
Alguna de las columnas referenciadas tiene distinto valor por defecto.	WARNING
Alguna de las secuencias referenciadas tiene distinto incrementador.	WARNING
Alguna de las secuencias referenciadas tiene distinto valor inicial.	WARNING

Cuadro C.2: Problemas en SELECT/DELETE

C.1.1. Modelo de Datos

El modelo de datos es el mismo catálogo de Oracle. Las tablas utilizadas permiten obtener la información de los objetos referenciados en las consultas, tales como, las tablas, columnas, secuenciadotes, triggers, sinónimos, y constraints.

C.2. Construcción de la DAL asociada

La herramienta desarrollada cuenta con una DAL de pequeña escala, integrada por ocho consultas de tipo SELECT, las cuales obtienen información sobre las tablas, columnas, sequence, triggers y sinónimos referenciados en las consultas de la metadata del Generator.

Dichas consultas fueron divididas es diferentes archivos TQL según

```
<Data>
    <Queries>
        <Query name="GetSequenceBySequenceName">
            <Source>
            <![CDATA[SELECT
                       seq.obj,
                       obj."NAME",
                       seq."increment",
                       seq."MINVALUE"
                     FROM
                       seq, obj
                     WHERE
                       seq.obj = obj.obj
                       AND "NAME" = :seqname]]>
            </Source>
        </Query>
    </Queries>
</Data>
```

Figura C.2: Ejemplo de archivo TQL

los objetos referenciados. Con el esquema de las tablas del catálogo de Oracle, junto con los archivos .tql y pasando como referencia al proyecto interfaz y el de la DAL, se invoca el generador de código. En la figura C.4 se puede ver la estructura inicial de archivos. La misma esta organizada en tres proyectos:

- DAL: Este proyecto contiene los archivos ".tql", y es el lugar de destino del código generado de las consultas.
- TarumanInterface: El proyecto TarumanInterface representa una interfaz de la metadata del Generator, utilizado para acceder a los objetos que representan a las entidades de la base de datos.
- VSC: Este es el proyecto principal de la solución, correspondiente a la lógica de la aplicación y reportes.

Luego del proceso, obtenemos como resultado una carpeta Database dentro del proyecto interfaz (TarumanInterface) que contiene las clases generadas que hacen referencia a los objetos del esquema y la

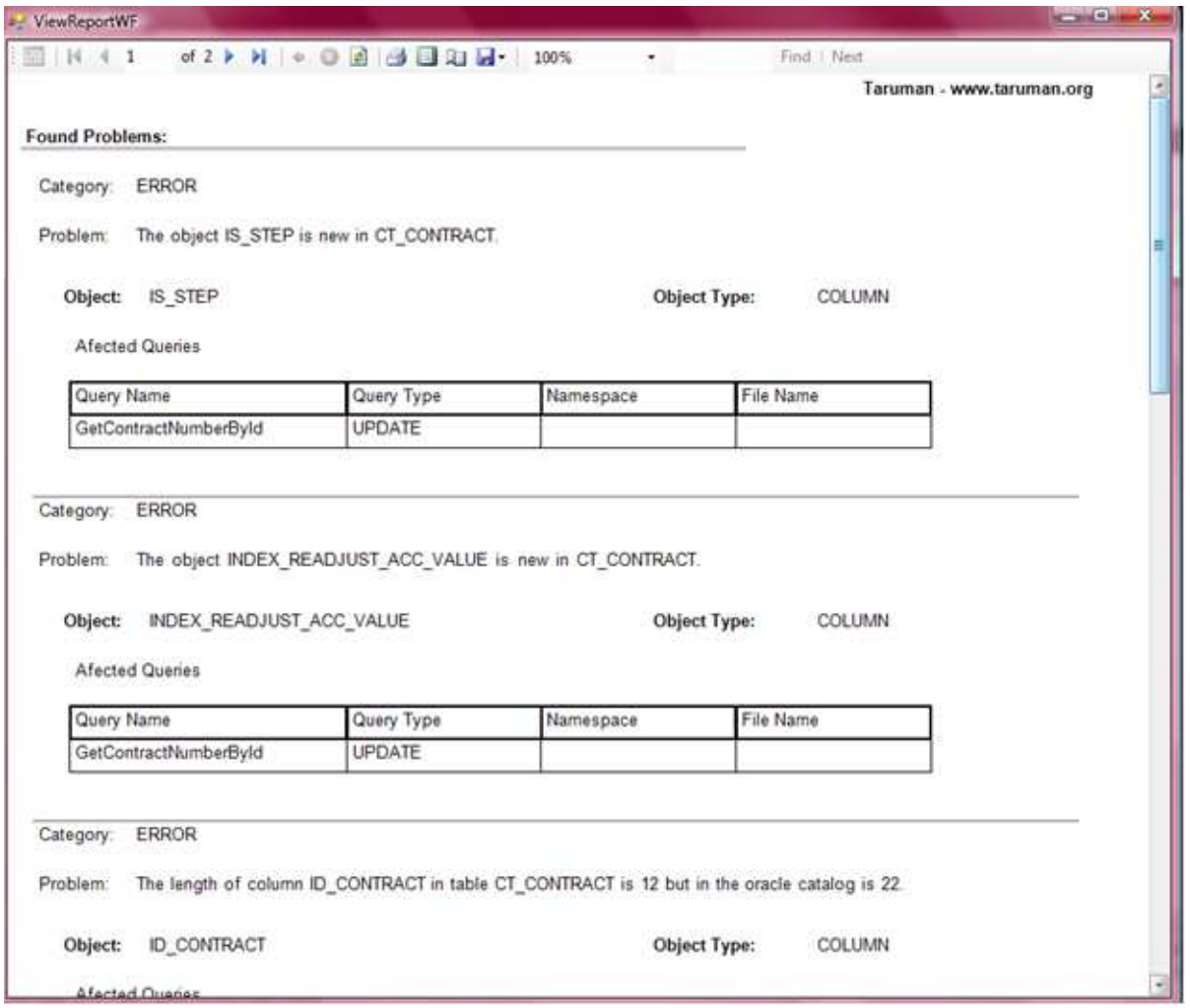

Figura C.3: Ejemplo de reporte de fallas

base de datos embebida del Generator (TarumanMetadata.sqlite), correspondiente a la metadata del modelo de datos y de las consultas, como se ve en la figura C.5. En la figura, también se puede ver los enumerados y datatypes generados correspondientes a los objetos de entrada y salida de las consultas procesadas.

Dentro del proyecto de la DAL, se generaron los archivos correspondientes al código de las consultas, ubicado junto con su archivo ".tql.ªsociado. El código generado consiste en cuatro métodos que

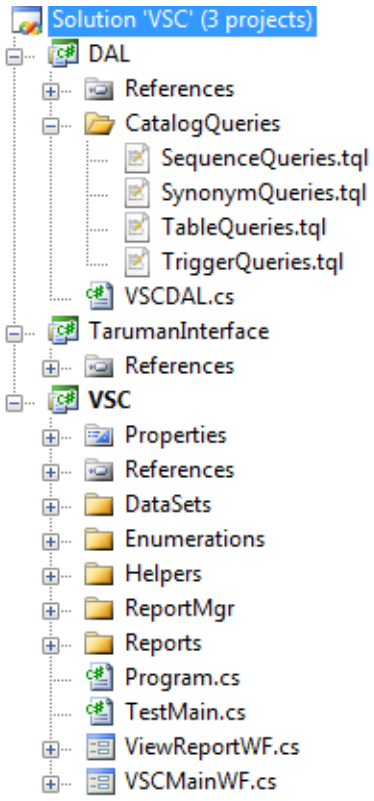

Figura C.4: Estructura de la solución

implementan la consulta indicada, pero sobrecargada para poder ser invocada con diferentes tipos de parámetros y obtener distintos tipos de datos de salidas. En la figura C.6 pueden verse los archivos con el código de las consultas.

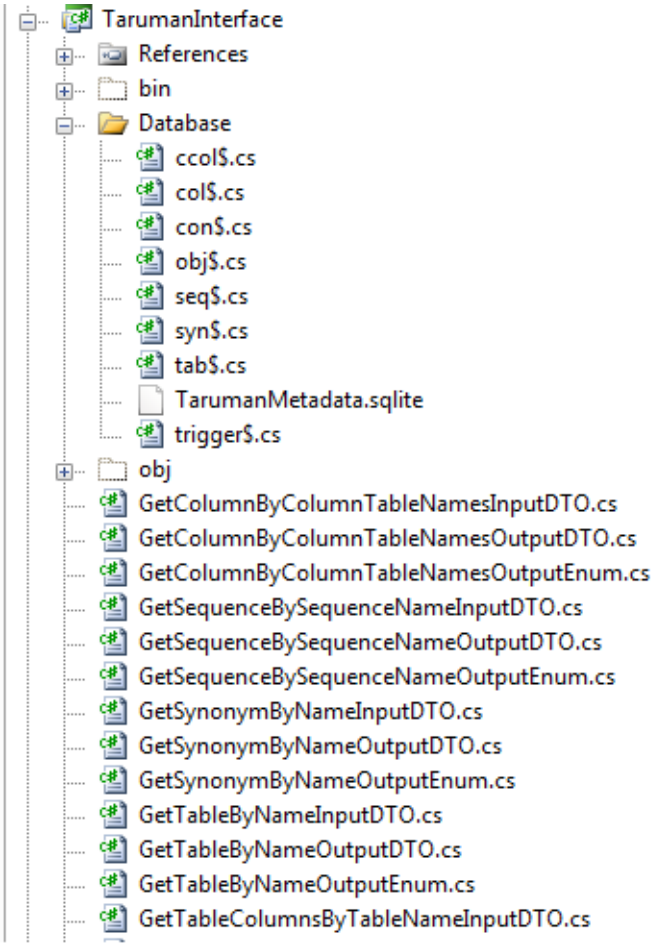

Figura C.5: Estructura de la solución

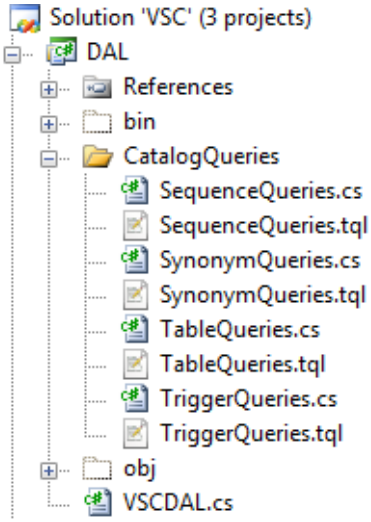

Figura C.6: Estructura de la solución

Apéndice D

Sentencias DML soportadas

D.1. Introducción

En este apéndice analizamos en detalle el soporte que ofrece la herramienta Generator de sentencias DML.

Se muestra la tabla D.1 en la sección D.2, en donde por cada sentencia se listan los distintos casos posibles, mostrando para los que corresponde un ejemplo de código, si es soportado o no. Finalmente se listan las distintas funciones soportadas en la sección D.3, agrupadas por tipo.

D.2. Sentencias soportadas

A continuación, se presenta la tabla de sentencias D.1, en donde se indica cuales son soportadas y cuales no.

Cuadro D.1: Sentencias DML soportadas

Sentencia	Ejemplo	Soportada
SELECT ... FROM ...	`select id_contract from ct_contract`	SI
SELECT hint ... FROM ...	`SELECT /*+INDEX_COMBINE(emp sal_bmi hiredate_bmi)*/ * FROM emp WHERE sal <50000 AND hiredate <'01-JAN-1990'`	SI
SELECT DISTINCT ... FROM ...	`select distinct id_contract from ct_contract`	SI
SELECT UNIQUE ... FROM ...	`select unique id_contract from ct_contract`	SI
SELECT ALL ... FROM ...	`select all id_contract from ct_contract`	SI
FROM ... HAVING ...	`select id_contract from ct_contract having`	SI
SELECT ... UNION ...	`ID_COMPANY >4 select id_contract from ct_contract union (select id_contractee from ct_contract)`	SI
SELECT ... UNION ALL ...	`select id_contract from ct_contract union all (select id_contractee from ct_contract)`	SI
SELECT ... INTERSECT ...	`select id_contract from ct_contract intersect (select id_contractee from ct_contract)`	SI
SELECT ... MINUS ...	`select id_contract from ct_contract minus (select id_contractee from ct_contract)`	SI
WITH ... AS	`with hola as (select id_contract from`	NO

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada
	`ct_contract)`	
AS	`select id_contract as b from ct_contract as b`	SI
ONLY	`select id_contract as b from only(ct_contract)`	NO
SELECT ANIDADOS	`select ID_CORPORATION, id_contract from ct_contract where department_id IN (select department_id from employees where last_name = a)`	SI
	`select * from ct_contract`	SI
table.* / view.* / materializer view.*	`select ct_contract.* from ct_contract`	SI
comparadores (=,>, >, etc)	`select ID_CORPORATION from ct_contract where ID_PRODUCT >= 3 and ID_COMPANY >= 4`	SI
VERSION BETWEEN SCN ... AND ... AS OF SCN ...		NO
VERSION BETWEEN TIMESTAMP ... AND ... AS OF SCN ...		NO
VERSION BETWEEN SCN ... AND ... AS OF TIMESTAMP ...		NO
VERSION BETWEEN TIMESTAMP ... AND ... AS OF TIMESTAMP ...		NO

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada
MINVALUE		NO
MAXVALUE		NO
AND	`select ID_CONTRACT from ct_contract where ID_COMPANY = 1 and ID_PRODUCT = 2`	SI
OR	`select ID_CONTRACT from ct_contract where ID_COMPANY = 1 or ID_PRODUCT = 2`	SI
squema.table PARTITION / idem view y materialized view	`select clasname, studentname, exam1_point = dense_ranck() over (partition by clasname order by exam1 desc) * (1)`	NO
squema.table SUBPARTITION / idem view y materialized view	`select clasname, studentname, exam1_point = dense_ranck over (subpartition by clasname order by exam1 desc) * (1)`	NO
squema.table @DBLINK / idem view y materialized view	`select ID_COMPANY from ct_contract@dblink`	NO
SAMPLE ...		NO
SAMPLE BLOCK ...		NO
SAMPLE BLOCK ... SEED ...		NO
WITH READ ONLY ...		NO
WITH CHECK OPTION ...		NO
WITH CHECK OPTION CONSTRAINT ...		NO
TABLE	`select ID_COMPANY from table ct_contract`	NO

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada
JOIN ...	`select ID_COMPANY from ct_contract join CT_COMPANY`	SI
JOIN ON ...	`select ID_CONTRACT from ct_contract join CT_COMPANY on ct_contract.ID_COMPANY=CT_COMPANY.ID_COMPANY`	SI
JOIN USING ...	`select ID_CONTRACT from ct_contract join CT_COMPANY using (ID_COMPANY)`	SI
INNER JOIN ...	`select ID_CONTRACT from ct_contract inner join CT_COMPANY`	SI
INNER JOIN ON ...	`select ID_CONTRACT from ct_contract inner join CT_COMPANY on ct_contract.ID_COMPANY=CT_COMPANY.ID_COMPANY`	SI
INNER JOIN USING ...	`select ID_CONTRACT from ct_contract inner join CT_COMPANY using(ID_COMPANY)`	SI
CROSS JOIN ...	`select ID_CONTRACT from ct_contract cross join CT_COMPANY`	SI
NATURAL JOIN ...	`select ID_CONTRACT from ct_contract natural join CT_COMPANY`	SI
NATURAL INNER JOIN ...	`select ID_CONTRACT from ct_contract natural inner join CT_COMPANY`	SI
PARTITION BY ...		NO

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada
FULL ...	`select ID_CONTRACT from ct_contract full join CT_COMPANY`	NO
LEFT ...	`select ID_CONTRACT from ct_contract left join CT_COMPANY`	SI
RIGHT ...	`select ID_CONTRACT from ct_contract right join CT_COMPANY`	SI
FULL OUTER ...	`select ID_CONTRACT from ct_contract full outer join CT_COMPANY`	SI
LEFT OUTER ...	`select ID_CONTRACT from ct_contract left outer join CT_COMPANY`	SI
RIGHT OUTER ...	`select ID_CONTRACT from ct_contract right outer join CT_COMPANY`	SI
WHERE ...	`select ID_CONTRACT from ct_contract where ID_COMPANY=1 and ID_CONTRACT= 2`	SI
START WITH CONNECT BY NOCYCLE ...		NO
SATRT WITH CONNECT BY ...		NO
CONNECT BY NOCYCLE...		NO
CONNECT BY ...		NO
GROUP BY ...	`select ID_CONTRACT from ct_contract group by ID_CONTRACT`	SI
GROUP BY ... HAVING...	`select ID_CONTRACT from ct_contract group`	SI

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada
	`by ID_CONTRACT having ID_CONTRACT>4`	
ROLLUP ...		NO
CUBE ...		NO
GROUPING SETS ...		NO
MODEL ...		NO
IGNORE NAV ...		NO
KEEP NAV ...		NO
UNIQUE DIMENSION...		NO
UNIQUE SINGLE REFERENCE...		NO
RETURN UPDATED ROWS ...		NO
RETURN ALL ROWS ...		NO
REFERENCE ... ON ...		NO
MAIN ...		NO
DIMENSION BY ... MESURES ...		NO
RULES ...		NO
RULES UPDATE ...		NO
RULES UPSERT ...		NO
RULES UPSERT ALL ...		NO
RULES AUTOMATIC ORDER ...		NO
RULES SEQUENTIAL ORDER ...		NO

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada
RULES ... ITERATE ...		NO
RULES ... ITERATE ... UNTIL ...		NO
FOR ... IN ...		NO
FOR ... FROM ... TO ... INCREMENT ...		NO
FOR ... FROM ... TO ... DECREMENT ...		NO
ORDER BY ...	`select ID_CONTRACT from ct_contract order by ID_CONTRACT`	SI
ORDER SIBLINGS BY ...		NO
ORDER BY ... ASC ...	`select ID_CONTRACT from ct_contract order by ID_CONTRACT asc`	SI
ORDER SIBLINGS BY ... ASC ...		NO
ORDER BY ... DESC ...	`select ID_CONTRACT from ct_contract order by ID_CONTRACT desc`	SI
ORDER SIBLINGS BY ... DESC ...		NO
ORDER BY ... NULLS FIRST ...		NO
ORDER SIBLINGS BY ...NULLS FIRST ...		NO
ORDER BY ... ASC ...NULLS FIRST ...		NO
ORDER BY ... DESC ...NULLS FIRST ...		NO
ORDER BY ... NULLS LAST ...		NO
ORDER SIBLINGS BY ...NULLS LAST ...		NO
ORDER BY ... ASC ...NULLS LAST ...		NO

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada
ORDER BY ... DESC ... NULLS LAST ...		NO
FOR UPDATE ...	`select ID_CONTRACT from ct_contract` `where ID_CONTRACT=1 for update`	SI
FOR UPDATE OF ...	`select ID_CONTRACT from ct_contract` `where ID_CONTRACT=1 for update of ID_CONTRACT`	SI
FOR UPDATE NOWAIT ...		NO
FOR UPDATE WAIT ...		NO
FOR UPDATE OF ... NOWAIT ...		NO
FOR UPDATE OF ... WAIT ...		NO
SELECTS ANIDADOS EN EL WHERE		

EXISTS, NOT EXISTS	`SELECT ct_contract`
	`FROM ct_contract contact`
	`WHERE NOT EXISTS (`
	`SELECT ct_branch`
	`FROM ct_branch branch`
	`WHERE`
	`contact.id_conctractee = branch.id_branch)`
		SI

GROUP COMPARISON - ALL	`SELECT id_contract, id_company, id_product`
	`FROM ct_contract WHERE`
	`id_product >= ALL (SELECT`
	`id_product FROM ct_contract WHERE`
		SI

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada
	`id_company = 3)`	

GROUP COMPARISON - ANY	`SELECT id_contract, id_company, id_product`
	`FROM ct_contract WHERE`
	`id_product >= ANY (SELECT`
	`id_product FROM ct_contract WHERE`
	`id_company = 3)`
		SI

GROUP COMPARISON - SOME	`SELECT id_contract, id_company, id_product`
	`FROM ct_contract WHERE`
	`id_product >= SOME`
	`(SELECT id_product FROM ct_contract WHERE`
	`id_company = 3)`
		SI

IN (list)	`SELECT id_contract, id_company`
	`FROM ct_contract`
	`WHERE id_product IN (1,2,3)`
		SI

IN (subquery)	`SELECT id_contract, id_company`
	`FROM ct_contract`
	`WHERE id_product IN (`
	`SELECT MIN(id_product)`
	`FROM ct_contract)`
		SI

NOT IN Antijoin	`SELECT id_contract, id_company` `FROM ct_contract`	SI

Continúa en la página siguiente

Cuadro D.1 – Continúa de la página anterior

Sentencia	Ejemplo	Soportada

	`WHERE id_product NOT IN (`
	`SELECT MIN(id_product)`
	`FROM ct_contract)`	

D.3. Funciones soportadas

Se presenta las tablas de funciones, en donde se indica cuales son soportadas y cuales no, agrupadas por tipo de función.

D.3.1. Numéricas

Función	**Soportada**
ABS	SI
ACOS	NO
ASIN	NO
ATAN	NO
ATAN2	NO
BITAND	NO
CEIL	SI
COS	NO
COSH	NO
EXP	NO
FLOOR	SI
LN	NO
LOG	NO
MOD	NO
NANVL	NO
POWER	SI
REMAINDER	NO
ROUND (number)	NO
SIGN	SI
SIN	NO
SINH	NO
SQRT	NO
TAN	NO
TANH	NO
TRUNC (number)	SI
WIDTH_BUCKET	NO

Cuadro D.2: Funciones numéricas

D.3.2. Caracteres

Función	Soportada
CHR	SI
CONCAT	SI
INITCAP	NO
LOWER	SI
LPAD	NO
LTRIM	SI
NLS_INITCAP	NO
NLS_LOWER	NO
NLSSORT	NO
NLS_UPPER	NO
REGEXP_REPLACE	NO
REGEXP_SUBSTR	NO
REPLACE	SI
RPAD	NO
RTRIM	SI
SOUNDEX	NO
SUBSTR	SI
TRANSLATE	NO
TREAT	NO
TRIM	SI
UPPER	SI

Cuadro D.3: Funciones sobre caracteres

D.3.3. Fechas

Función	Soportada
ADD_MONTHS	SI
CURRENT_DATE	SI
CURRENT_TIMESTAM	SI
DBTIMEZONE	NO
EXTRACT (datetime)	SI
FROM_TZ	NO
LAST_DAY	SI
LOCALTIMESTAMP	SI
MONTHS_BETWEEN	SI
NEW_TIME	NO
NEXT_DAY	NO
NUMTODSINTERVAL	NO
NUMTOYMINTERVAL	NO
ROUND (date)	SI
SESSIONTIMEZONE	NO
SYS_EXTRACT_UTC	NO
SYSDATE	SI
SYSTIMESTAMP	NO
TO_CHAR (datetime)	SI
TO_TIMESTAMP	SI
TO_TIMESTAMP_TZ	NO
TO_DSINTERVAL	NO
TO_YMINTERVAL	NO
TRUNC (date)	SI
TZ_OFFSET	NO

Cuadro D.4: Funciones sobre fechas

D.3.4. Conversión

Función	Soportada
ASCIISTR	NO
BIN_TO_NUM	NO
CAST	NO
CHARTOROWID	NO
COMPOSE	NO
CONVERT	NO
DECOMPOSE	NO
HEXTORAW	NO
NUMTODSINTERVAL	NO
NUMTOYMINTERVAL	NO
RAWTOHEX	NO
RAWTONHEX	NO
ROWIDTOCHAR	NO
ROWIDTONCHAR	NO
SCN_TO_TIMESTAMP	NO
TIMESTAMP_TO_SCN	NO
TO_BINARY_DOUBLE	NO
TO_BINARY_FLOAT	NO
TO_CHAR	SI
TO_DATE	SI
TO_NCHAR	NO
TO_LOB, TO_NCLOB	NO
TO_MULTI_BYTE	NO
TO_NUMBER	SI
TO_DSINTERVAL	NO
TO_SINGLE_BYTE	NO
TO_TIMESTAMP	SI
TO_TIMESTAMP_TZ	NO
TO_YMINTERVAL	NO
TO_YMINTERVAL	NO
TRANSLATE ... US	NO
UNISTR	NO

Cuadro D.5: Funciones de conversión

D.3.5. NLS Carácter

Función	Soportada
NLS_CHARSET_DECL_LEN	NO
NLS_CHARSET_ID	NO
NLS_CHARSET_NAME	NO

Cuadro D.6: NLS Carácter

D.3.6. Funciones C que retornan N

Función	Soportada
ASCII	NO
INSTR	NO
LENGTH	SI
REGEXP_INSTR	NO

Cuadro D.7: Funciones C que retornan N

D.3.7. Comparación

Función	Soportada
GREATEST	SI
LEAST	SI

Cuadro D.8: Funciones de Comparación

D.3.8. Objetos grandes

Función	Soportada
BFILENAME	NO
EMPTY_BLOB, EMPTY_CLOB	NO

Cuadro D.9: Funciones sobre Objetos grandes

D.3.9. Conjuntos

Función	Soportada
CARDINALITY	NO
COLLECT	NO
POWERMULTISET	NO
POWERMULTISET_BY_CARDINALITY	NO
SET	SI

Cuadro D.10: Funciones sobre Conjuntos

D.3.10. Jerarquía

Función	Soportada
SYS_CONNECT_BY_PATH	NO

Cuadro D.11: Funciones sobre Jerarquías

D.3.11. Data Mining

Función	Soportada
CLUSTER_ID	NO
CLUSTER_PROBABILITY	NO
CLUSTER_SET	NO
FEATURE_ID	NO
FEATURE_SET	NO
FEATURE_VALUE	NO
PREDICTION	NO
PREDICTION_COST	NO
PREDICTION_DETAILS	NO
PREDICTION_PROBABILITY	NO
PREDICTION_SET	NO

Cuadro D.12: Funciones sobre Data Mining

D.3.12. Codificación y Decodificación

Función	Soportada
DECODE	SI
DUMP	NO
ORA_HASH	NO
VSIZE	NO

Cuadro D.13: Funciones sobre Codificación y Decodificación

D.3.13. Valores Nulos

Función	Soportada
COALESCE	NO
LNNVL	NO
NULLIF	SI
NVL	SI
NVL2	SI

Cuadro D.14: Funciones sobre Valores nulos

D.3.14. Ambiente e Identificación

Función	Soportada
SYS_CONTEXT	NO
SYS_GUID	NO
SYS_TYPEID	NO
UID	NO
USER	NO
USERENV	NO

Cuadro D.15: Funciones sobre Ambiente e Identificación

D.3.15. Agrupación

Función	Soportada
AVG	SI
COLLECT	NO
CORR	NO
CORR_*	NO
COUNT	SI
COVAR_POP	NO
COVAR_SAMP	NO
CUME_DIST	NO
DENSE_RANK	NO
FIRST	SI
GROUP_ID	NO
GROUPING	SI
GROUPING_ID	NO
LAST	SI
MAX	SI
MEDIAN	SI
MIN	SI
PERCENTILE_CONT	NO
PERCENTILE_DISC	NO
PERCENT_RANK	NO
RANK	NO
REGR_ (Linear Regression)	NO
STATS_*	NO
STDDEV	NO
STDDEV_POP	NO
STDDEV_SAMP	NO
SUM	SI
VAR_POP	NO
VAR_SAMP	NO
VARIANCE	NO

Cuadro D.16: Funciones sobre Agrupación

D.3.16. XML

Función	Soportada
APPENDCHILDXML	NO
DELETEXML	NO
DEPTH	NO
EXTRACT (XML)	NO
EXISTSNODE	NO
EXTRACTVALUE	NO
INSERTCHILDXML	NO
INSERTXMLBEFORE	NO
PATH	NO
SYS_DBURIGEN	NO
SYS_XMLAGG	NO
SYS_XMLGEN	NO
UPDATEXML	NO
XMLAGG	NO
XMLCDATA	NO
XMLCOLATTVAL	NO
XMLCOMMENT	NO
XMLCONCAT	NO
XMLFOREST	NO
XMLPARSE	NO
XMLPI	NO
XMLQUERY	NO
XMLROOT	NO
XMLSEQUENCE	NO
XMLSERIALIZE	NO
XMLTABLE	NO
XMLTRANSFORM	NO

Cuadro D.17: Funciones sobre XML

Apéndice E

Verificación y testing

El objetivo principal de la verificación, consiste en asegurar el correcto funcionamiento de cada componente del proyecto, de forma que el resultado final sea el estipulado en el alcance y cubra lo especificado en los requerimientos del cliente.

En primer lugar se mostrará la planificación de pruebas y sus respectivos desempeños, describiendo los métodos de testing, así como las herramientas utilizadas para tal fin. Y por último se dará una lista de los bugs conocidos.

E.1. Planificación de pruebas

Al plantear el plan de verificación se consideraron como puntos a tener en cuenta:

- Que se debía asegurar que la gramática soportara como mínimo los objetos de Oracle que se estipularon en el alcance.
- Que la metada tanto del modelo como de consultas, sea almacenada correctamente, y que los datos que se almacenen cubran las necesidades básicas, para la posterior generación de código, verificación semántica y sintáctica de consultas y análisis de impacto.

- Que las clases *C#* generadas tengan la estructura de código estipulada en el alcance y que el funcionamiento de las mismas sea el apropiado.

En primer lugar se planificó la etapa de pruebas unitarias, de modo que a medida que se fueron desarrollando los diferentes componentes, se verificara que en cada etapa los resultados obtenidos eran los esperados.

Luego de la etapa de desarrollo de cada componente, se procedió con las pruebas integradas. Las mismas se llevaron a cabo mediante la utilización de la herramienta Nunit [22], realizando los test de ejecución automática.

En este punto, las pruebas se dividieron en 3 etapas, la verificación de los objetos de Oracle soportados por la gramática, las pruebas de verificación de la metadata y la de verificación de código generado.

E.2. Ejecución de pruebas integradas

La ejecución de las pruebas integradas, se subdividió en varios pasos para testear cada uno de las etapas de la generación de código, los cuales se describen a continuación:

- Verificación de los objetos soportados

  Para comprobar que tipo de sentencias/objetos de catálogo son soportados por la gramática, se verificó tanto la gramática del modelo de datos (de aquí en más gramática DDL), como la de consultas (de aquí en más gramática DML). Para las pruebas de la gramática DDL, se probó con tres esquemas de base de datos proporcionados por el cliente. Se detectaron varios casos nos soportados por la gramática. Para los casos que no se pudieron resolver se hicieron las modificaciones necesarias a los archivos (".sql") de definición de los modelos para poder llevar a cabo el resto de las pruebas, reportando dichos casos como bugs conocidos. Para las pruebas de la gramática DML, se creó una planilla donde se describían las pruebas básicas que se debían

ejecutar para comprobar, que los diferentes objetos del catalogo de Oracle eran soportados por la gramática. Se hicieron pruebas con sentencias de diversa complejidad, que incluían las clausulas SELECT, UPDATE, INSERT y DELETE.

- Verificación de metadata

  Para la verificación de metadata tanto de la gramática DDL, como DML, se hicieron comparaciones de los datos almecenados en la base de datos, de forma tal que se crearon nuevas tablas idénticas a las originales, con el prefijo TEST, de modo de almacenar en estas los resultados esperados, para ser comparados posteriormente, mediante los test automáticos, con los resultados reales.

- Verificación de código generado

  En la última etapa se hicieron pruebas del código generado, verificando que las clases tuvieran la estructura esperada y comprobando mediante su utilización, que el código generado funcionaba como era esperado. Se probaron consultas de diverso grado de complejidad, tratando de abarcar una amplia variedad de combinaciones de sentencias SELECT, UPDATE, INSERT Y DELETE. A su vez se hicieron diversas pruebas con aplicación de reglas, a las sentencias. Verificando en el caso de las reglas de filtro, que la sentencia se aplicaba correctamente al código de la consulta a aplicar, y que su ejecución era correcta. Y en el caso de las reglas de verificación, se probó con diferentes métodos, con y sin parámetros, validando que se aplicaran según correspondía y que su ejecución era la esperada. Para esto se crearon las clases de equivalencia correspondientes, para ser comparadas, mediante los test automáticos, con las clases reales resultantes, verificando que la estructura de las clases generadas eran las esperadas. A su vez se crearon clases de test, para ejecutar los métodos creados en las clases generadas, probando que el código tenía la funcionalidad esperada.

Por último, se realizaron pruebas con un volumen importante de consultas, para comprobar que los componentes de Generator en su conjunto tienen el resultado esperado. En primer lugar, como mencionamos anteriormente, se utilizaron tres esquemas brindados por

el cliente, los cuales constaban de 28 archivos de especificación del modelo de datos, en total. Y por otro lado, con respecto a las consultas de pruebas, se crearon alrededor de 120 consultas, distribuidas en uno y en varios archivos ".tql".

El criterio de selección de casos de pruebas, en primer lugar, se hizo en base a la planilla descripta anteriormente, para validar las diferentes sentencias soportadas. A continuación se procedió a combinar consultas, incluyendo diferentes anidamientos de las mismas. Luego, se incluyeron diferentes tipos de parámetros, con diferentes tipos de datos. Y por último se incorporaron diferentes tipos de reglas. Por otro lado también se probaron casos de error, es decir, sentencias que efectivamente eran erróneas, para verificar que realmente se interpretaran como tales.

En esta etapa se detectaron errores en el 20 % de las consultas, de los cuales se resolvió un 75 % de los errores encontrados.

Aquí se detectaron varios errores que serán descriptos en el cuadro E.2, agrupándolos por tipo.

Errores encontrados	**Resolución**
Errores al intentar procesar sentencias no soportadas por la gramática.	Resueltos en un 85 %, con algunas excepciones descriptas en los bugs conocidos.
Errores de almacenamiento de datos de la metadata.	Resueltos en un 65 %, con algunas excepciones descriptas en los bugs conocidos.
Errores de sentencias que Oracle no soporta y Generator las permite.	No resueltos, se describirán en los bugs conocidos.
Errores al generar las clases de código *C#* .	Resueltos en en un 90 %, con algunas excepciones descriptas en los bugs conocidos.
Errores de ejecución de las clases de código generadas.	Resueltos en su totalidad.

Cuadro E.1: Errores encotrados vs resultados de su resolución

Los errores fueron reportados y administrados mediante la herramienta Mantis [23], brindada por *PayTrue Solutions* [1], la cual nos permitió tener un buen control del reporte y avance de los diferentes errores encontrados.

Se muestra a continuación en el gráfico E.1 el avance en la resolución de errores, agrupados según el tipo de error, haciendo una analogía con el cuadro anterior.

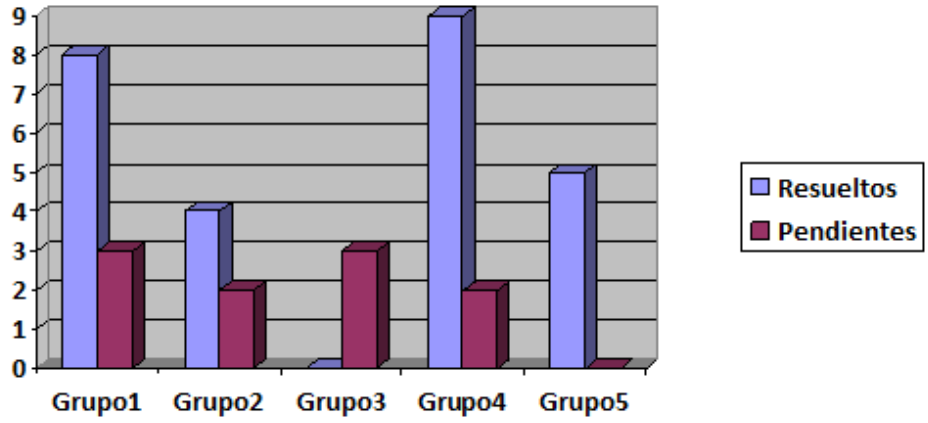

Figura E.1: Avance en la resolución de errores

Como se aprecia en el gráfico, la mayoría de las incidencias encontradas han sido resueltas. Las que no, algunos son errores conocidos que no es posible o no amerita la resolución de los mismos considerando el alcance del proyecto. Y en otros casos, los errores han sido descubiertos a último momento.

E.3. Bugs Conocidos

Casos de prueba	**Errores encontrados**
CREATE TABLE tablename (YEAR number(8))	No se soporta que los nombres de tablas o de columnas contengan palabras reservadas. Por otro lado, estos nombres si son soportados si se los pone entre comillas.
CREATE PROCEDURE	No se soporta la creación de store procedures

Cuadro E.2: Sentencias no soportadas por la gramática

Casos de prueba	Errores encontrados
SELECT c1 FROM t1 (INNER) JOIN t2 USING (c2)	Se registra en la metadata solo un nombre de columna correspondiente a una de las dos tablas del join
select c1 from t1 cross join t2	No se registra en la metadata el nombre de la segunda tabla del join

Cuadro E.3: Error de almacenamiento de datos en la metadata

Casos de prueba	Errores encontrados
SELECT c FROM t AS t1	Oracle no soporta esta sentencia por el alias, mientras el componente Generator sí
SELECT c1 FROM t1 HAVING c2>4	Oracle no soporta esta sentencia porque le falta el group by, mientras el componente Generator sí
SELECT c FROM t1 (INNER/LEFT/RIGHT) (OUTER) JOIN t2	Oracle no soporta esta sentencia, mientras el componente Generator sí

Cuadro E.4: Sentencias que Oracle no soporta y la gramática las permite

Casos de prueba	Errores encontrados
SELECT c1, c2, c3 FROM t1 WHERE c3 IN (SELECT c6 FROM t2 WHERE c6 =:param)	No compila la clase generada, por un error en el casteo de los datos resultantes
CREATE TABLE con	Windows no permite el nombre de archivo "con.*" con lo cual no es posible crear la clase con.cs

Cuadro E.5: Errores en las clases de código de consultas

Apéndice F

Resumen Ejecutivo

En el siguiente apéndice, se presenta en forma general como fue el proceso realizado para el desarrollo del producto, junto con el tiempo aproximado correspondiente a la duración de cada fase y las tareas realizadas en las mismas.

F.1. Desarrollo del Proyecto

F.1.1. Fases

El proceso seguido durante la elaboración de las herramientas se dividió en cuatro fases:

- Fase de Análisis
- Fase de Investigación
- Fase de Implementación Básica
- Fase de Implementación Complementaria
- Fase de Testing y Corrección de errores

Fase de Análisis

Esta fase se dividió en dos partes, grupo y producto.

- **Grupo**
  Se definió la forma de trabajo, los medios de comunicación dentro del grupo, con el cliente y tutor.
- **Producto**
  Se estudió a fondo las tecnologías disponibles para el procesamiento de la gramática SQL, como manipular los datos parseados de la misma. También, se definieron los casos de uso y se analizaron en detalle, de manera de poder definir la arquitectura de la herramienta principal.

Duración de la fase: 12 semanas

Fase de Investigación

Esta fase abarca la investigación de las tecnologías a utilizar, de modo de poder mitigar el riesgo en forma temprana. Las herramientas que se investigaron incluyen:

- Procesadores de gramática SQL, en particular *ANTLR* [19] (versiones 2 y 3)
- Varios ejemplos de gramática SQL
- Distintas opciones de bases de datos embebidas. Finalmente, se decidió utilizar SQLite [21].

Por último se construyeron prototipos desechables, de modo de poder aprender a utilizar la tecnología.

Duración de la fase: 8 semanas

Fase de Implementación Básica

Al llegar a esta fase, ya se tenían definidas la tecnología a utilizar para poder procesar las sentencias SQL, y para manejar el almacenamiento de los datos (lo cual se definió como metadata en la fase anterior).

Lo que se pretendía en esta fase, era implementar una versión reducida de la herramienta Generator. La misma contaba con las funcionalidades básicas del Generator, tales como el parseo de los archivos SQL correspondientes al modelo de datos, y parseo de los archivos

TQL (soportando algunos tipos de consulta), mantenimiento de la metadata y generación de código de enumerados.

Duración de la fase: 8 semanas

Fase de Implementación Complementaria

En esta fase de completó el desarrollo del Generator. Se logró la integración de los módulos del mismo, ya que se estaban utilizando individualmente, se continuó agregando funcionalidades a soportar para la gramática SQL DML y generación de código de las consultas, y se comenzó a implementar el Verificador sistemático de consultas.

Paralelamente, se empezó a definir el mecanismo de testing de la herramienta Generator utilizando NUnit.

Duración de la fase: 8 semanas

Fase de Testing y Corrección de errores

Se realizaron los test en NUnit, además de pruebas unitarias para detectar errores dentro de la herramienta Generator, y se verificó el correcto funcionamiento del código generado por el mismo. Los errores detectados fueron reportados en la aplicación Mantis (brindada por *PayTrue Solutions*), y se asignaron para ser solucionados.

Duración de la fase: 4 semanas

F.1.2. Riesgos ocurridos

Durante el desarrollo del proyecto surgieron varios riesgos, algunos de los cuales fueron identificados en la fase de análisis, y otros que fueron independientes del mismo pero repercutieron en su avance. A continuación presentamos los riesgos identificados junto con las acciones tomadas para mitigarlos:

- Uno de los integrantes del grupo se muda al extranjero: Para ello, se reforzaron los medios de comunicación vía Internet, creando varios canales de información, vía chat, e-mails, Google Groups, de modo que no se vea afectado el avance del proyecto. El único inconveniente que no se pudo controlar fue la diferencia horaria, la cual se vio incrementada por el cambio de horario.

- Rotura de uno o más equipos de desarrollo: Se trató de sustituir el/los equipo/s dañado a la brevedad.
- Utilización de la gramática SQL con *ANTLR* [19]: La utilización de *ANTLR* [19] y la çustomización"de la gramática SQL llevó más tiempo de lo esperado. Para ello, se separó la gramática SQL DDL y la DML, de modo de poder dividir los recursos. Se hizo un chequeo general de la misma para poder determinar que se estaba soportando y que había que agregar.

Dentro de los riesgos no identificados, se destacaron problemas serios de salud por parte de alguno de los integrantes del grupo, y de familiares directos de los mismos. Esto provocó que los tiempos estimados se dilaten más de lo esperado.

F.2. Mediciones de Esfuerzo (semanas)

Presentamos el gráfico F.1 que muestra las semanas por fase durante el desarrollo del proyecto.

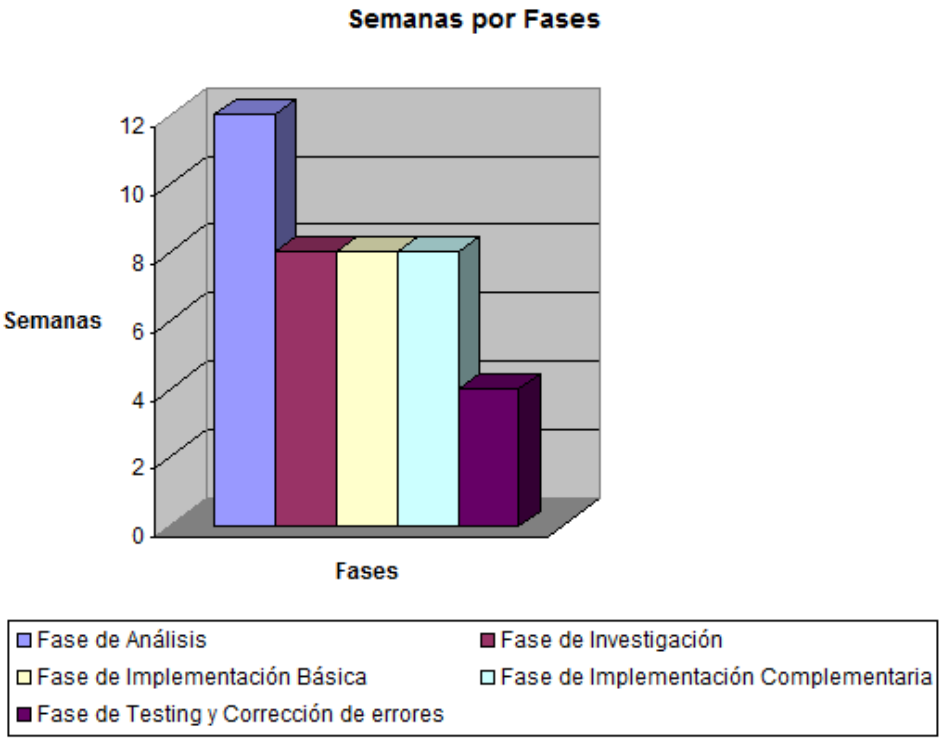

Figura F.1: Semanas por Fases

Apéndice G

Glosario

AST Abstract Syntax Tree. Es un árbol que representa en forma abstracta la estructura sintáctica de código fuente escrito en un lenguaje de programación.

Dataset Representación de datos residente en memoria que proporciona un modelo de programación relacional coherente independientemente del origen de datos que contiene.

DBMS Database Management System. Interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan.

DDL Data Definition Language. Comandos soportados por una base de datos que permite la crear, remover, o modificación de estructuras de la misma.

DML Data Manipulation Language. Comandos soportados por una base de datos que permite el acceso a los datos, incluyendo la creación, retornar, actualizar y borrado de los mismos.

DTO Data Transfer Object. Objeto utilizado para la transferencia de datos entre distintas capas de una aplicación.

Hint Indicación incluída en las consultas SQL para que el optimizador SQL de Oracle elabore el plan de ejecución de acuerdo a los criterios establecidos por el usuario.

ANSI American National Standards Institute

Metadata Datos estructurados y codificados que describen características de instancias conteniendo información para ayudar a identificar, descubrir, valorar y administrar las instancias descriptas.

Middleware Componente de software que permite la conexión entre dos o más elementos de forma tal que los mismos puedan intercambiar información.

Modelo de datos Sistema forma y abstracto que permite describir los datos de acuerdo con reglas definidas.

ORM Object-Relational mapping. Definición de la relación entre los datos de un esquema de objetos y un esquema de una base de datos relacional.

Sistema de información Conjunto de elementos que interactúan entre sí con el fin de apoyar las actividades de una empresa o negocio. Un sistema de información realiza cuatro actividades básicas: entrada, almacenamiento, procesamiento y salida de información.

TQL Taruman Query Language. Lenguaje utilizado para escribir las consultas y reglas. Básicamente, se trata de XML con SQL embebido. Para su manipulación, se utilizan archivos con extensión ".tql".

Bibliografía

[1] PayTrue Solutions. http://www.paytrue.com, último acceso en junio de 2009.

[2] Oracle. http://www.oracle.com, último acceso en junio de 2009.

[3] Microsoft SQL Server. http://www.microsoft.com/latam/sqlserver, último acceso en junio de 2009.

[4] MySQL. http://www.mysql.org, último acceso en junio de 2009.

[5] QDesigner. http://www.quest.com/toad-data-modeler, último acceso en junio de 2009.

[6] PostgreSQL. http://www.postgresql.org, último acceso en junio de 2009.

[7] LLBL GenPro. http://www.llblgen.com, último acceso en junio de 2009.

[8] Hibernate. https://www.hibernate.org, último acceso en junio de 2009.

[9] iPersist. http://ipersist.sourceforge.net, último acceso en junio de 2009.

[10] Linq. http://msdn.microsoft.com/en-us/library/bb308959.aspx, último acceso en junio de 2009.

[11] ADO.NET. http://msdn.microsoft.com/es-es/library/e80y5yhx(VS.80).aspx, último acceso en junio de 2009.

[12] Refactoring Databases. Scott W. Ambler, Pramond J. Sadalage. *Refactoring Databases, Evolutionary Database Design*. Addison-Wesley, 2006.

[13] Abstract Syntax Tree. Joel Jones. *Abstract Syntax Tree Implementation Idioms*. Department of Computer Science University of Alabama, 2003.

[14] Mecanismos de Persistencia en Sistemas Orientados a Objetos. Pablo Miranda, Joaquín Prudenza, Andrés Segurola. *Informe de Proyecto de Grado*. Facultad de Ingeniería, Universidad de la República, 2007.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, y John Irwin. Aspect-Oriented Programming. En Mehmet Ak¸sit y Satoshi Matsuoka, editores, Proceedings European Conference on Object-Oriented Programming, volumen 1241, páginas 220–242. Springer- Verlag, Berlin, Heidelberg, y New York, 1997.

[16] Patterns of Enterprise Application Architecture. Martin Fowler *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[17] Programming the Microsoft ADO.NET Entity Framework. David Sceppa *Programming the Microsoft ADO.NET Entity Framework, Draft Version*. Microsoft Press, 2009.

[18] Fighting impedance mismatch at the database level. Mary A. Finn *Fighting impedance mismatch at the database level Version*. Intersystem Corporation.

[19] ANother Tool for Language Recognition. ANTLR. http://www.antlr.org/, último acceso en junio de 2009.

[20] IBatis. http://ibatis.apache.org//, último acceso en junio de 2009.

[21] SQLite. http://www.sqlite.org/, último acceso en junio de 2009.

[22] NUnit. http://www.nunit.org/index.php/, último acceso en junio de 2009.

[23] Mantis. http://www.mantisbt.org/, último acceso en junio de 2009.