VNIVERSIDAD NACIONAL
AVTÓNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"DESARROLLO Y USO DE HERRAMIENTAS PARA LA REFACTORIZACIÓN DE BASES DE DATOS"

T E S I S

QUE PARA OPTAR POR EL GRADO DE:

MAESTRO EN INGENIERÍA (COMPUTACIÓN)

P R E S E N T A:

ROMÁN ARANGO DÍAZ

DRA. AMPARO LÓPEZ GAONA
FACULTAD DE CIENCIAS
PROGRAMA DE POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

MÉXICO, DF. AGOSTO 2013

Agradecimientos

A mi Madre Victoria Díaz Calvo.

Por todo su cariño, apoyo y comprensión. Por todo el sacrificio que hizo para que pudiera salir adelante y por ser mi ejemplo de fortaleza, valentía y perseverancia. Quiero agradecer a Dios por haberme asignado, a ti, como tu hijo, porque fuiste la mejor mamá de todo el Universo. Gracias por todo. Te amo.

A mi familia.
Gracias por todo su apoyo y cariño; por estar conmigo en los momentos felices pero más aún por estar conmigo también en los momentos difíciles.

A mis compañeros de generación y a mis maestros del posgrado.

A mis amigos, ustedes saben quienes son, por alentarme en todos los momentos y por compartir pasajes importantes de mi vida.

A la UNAM y al CONACyT por darme la oportunidad de seguir preparándome en el ámbito profesional.

ÍNDICE GENERAL

INDICE DE FIGURAS

INDICE DE TABLAS

INTRODUCCION

Hoy en día, las bases de datos constituyen una parte muy importante dentro del proceso de desarrollo de software, a la vez que son un recurso esencial para las empresas u organizaciones. Debido a su relevancia, las bases de datos deben estar diseñadas de tal manera que su desempeño sea óptimo, tengan redundancia de información mínima, no presenten inconsistencias y que su mantenimiento y acceso a la información sea eficiente.

En muchas ocasiones vamos a encontrar bases de datos mal diseñadas, principalmente porque son bases de datos heredadas, las cuales será necesario mejorarlas. Debido a que la mayoría de las veces las bases de datos son un recurso compartido, es necesario que los cambios o mejoras que se realicen sean llevados a cabo de manera segura y sin que afecten a la información misma o a las aplicaciones que accesan a ella.

Una técnica de desarrollo de bases de datos que puede ayudar a realizar esos cambios o mejoras es la refactorización de bases de datos.

Una refactorización de bases de datos, es un cambio simple a un esquema de bases de datos el cual mejora su diseño mientras mantiene su semántica de comportamiento y su semántica informativa. En otras palabras, no se puede agregar nueva funcionalidad o romper la funcionalidad existente. Tampoco se pueden agregar nuevos datos o cambiar el significado de los datos existentes.

Un esquema de bases de datos se compone tanto de aspectos estructurales como por ejemplo tablas y definición de vistas, así como de aspectos funcionales, como por ejemplo procedimientos almacenados y disparadores.

El proceso de refactorización de bases de datos, es el acto de realizar estos cambios simples al esquema.

La refactorizaciones de bases de datos son conceptualmente más difíciles que las de código debido a que las refactorizaciones de código solamente necesitan mantener la semántica del comportamiento, mientras que las refactorizaciones de bases de datos también necesitan mantener la semántica informativa. Más aún, las refactorizaciones de bases de datos pueden llegar a ser más complicadas debido al grado de acoplamiento resultante de la arquitectura de la base de datos.

Cuando se refactoriza un esquema de base de datos, se debe mantener tanto la semántica informativa como la semántica del comportamiento, no se debería agregar nada ni tampoco quitar nada. La semántica informativa se refiere al significado de información en la base de datos desde el punto de vista de los usuarios de esa información.

Mantener la semántica informativa, implica que, si se cambian los valores de los datos almacenados en una columna, los clientes de esa información no deberían ser afectados por el cambio.

Similarmente, con respecto a la semántica del comportamiento, el objetivo es mantener la funcionalidad de caja negra, cualquier código que funcione con las modificaciones realizadas

al esquema de la base de datos, debe ser retrabajado para que cumpla con la misma funcionalidad de antes.

Es importante reconocer que las refactorizaciones de bases de datos son un subconjunto de las transformaciones de bases de datos. Una transformación de bases de datos podría cambiar la semántica; una refactorización de bases de datos no la cambia.

Entre los problemas que existen actualmente con esta técnica ágil de desarrollo evolutivo de bases de datos, están:

1.- Que a pesar de que esta técnica representa un gran avance en cuanto a tratar de emparejar la forma de trabajar entre el área de desarrollo de software y el área de base de datos, no se ha adoptado como tal debido a que muchos de los profesionistas del área de datos prefieren utilizar un enfoque de desarrollo tradicional.

2.- La curva de aprendizaje para aprender nuevas técnicas puede tomar tiempo, más cuando se trata de cambiar de una mentalidad serial o tradicional a una mentalidad ágil.

3.- Las herramientas de software, en las cuales se podría apoyar el proceso de refactorización, aún están en desarrollo o evolución.

4.- Las herramientas de software existentes requieren de conocimientos previos en temas específicos, lo cual dificulta la adopción de esa técnica. Por ejemplo, conocimientos de XML, XSD y SQL.

OBJETIVOS:

- Desarrollar e implementar una herramienta de software que automatice y facilite la creación de archivos XML, utilizados para refactorizar bases de datos.
- Aplicar el proceso de refactorización a una base de datos en operación utilizando esta herramienta.
- Mejorar el diseño de un esquema de base datos haciendo uso de la refactorización de bases de datos.
- Impulsar y difundir el uso de la técnica de refactorización de base de datos entre la comunidad de los administradores de bases de datos.

ALCANCE

El proceso de refactorización que se aplicará al esquema de base de datos, se llevará a cabo bajo el escenario de una sola aplicación. Esta es la situación más simple que se puede encontrar, porque se tiene el control total sobre el esquema de la base de datos y el código de la aplicación que accesa a ella. Esto implica que se puede refactorizar a ambos en paralelo, sin tener que utilizar periodos de transición demasiado largos para cada refactorización que se aplique al esquema.

CONTENIDO DE LA TESIS

Capítulo 1.-Marco teórico.
En este capítulo se centran las bases para situar el problema en estudio dentro de un conjunto de conocimientos sólidos y confiables, que permitan orientar la búsqueda y ofrezcan una conceptualización adecuada de los términos que se van a utilizar. Este capítulo está enfocado a conceptos básicos de bases de datos, conceptos de refactorización de bases de datos, al proceso de refactorización de una base de datos y a la biblioteca Java conocida como Liquibase.

Capítulo 2.- Tecnologías en Java relacionadas con XML.
En este capítulo se definen las bibliotecas Java empleadas en la construcción de la aplicación, la cual será utilizada para generar documentos en formato XML a partir de un esquema XSD. Dentro de estas bibliotecas Java se encuentran XSOM(XML Schema Object Model) y JDOM.

Capítulo 3.- Construcción de la aplicación para generar refactorizaciones en formato XML.
En este capítulo se describe el desarrollo de la aplicación utilizando como metodología de desarrollo de software el proceso unificado y el patrón de arquitectura conocido como MVC(Modelo-Vista-Controlador).

Capítulo 4.- Caso práctico: Refactorizando una base de datos en un ambiente de una sola aplicación.
En este capítulo se describe como se debe llevar a cabo el proceso de refactorización a una base de datos en operación, haciendo uso de las herramientas de software disponibles para ello. Dentro de estas herramientas se encuentra la aplicación desarrollada en el capítulo 3 y la biblioteca Java Liquibase. También se muestran los beneficios obtenidos después de aplicar el proceso.

Capítulo 5.- Conclusiones.

Capítulo 6.- Referencias bibliográficas.

CAPITULO 1: Marco Teórico.

1.1. BASES DE DATOS RELACIONALES

En este capítulo, veremos los conceptos teóricos y prácticos vinculados con bases de datos relacionales, desde el diseño hasta la derivación del modelo físico, pasando por el estudio de las entidades, las relaciones, los procesos de normalización y las claves de evaluación de un diseño.

¿QUÉ ES UNA BASE DE DATOS?

Una base de datos representa una colección de información, organizada en función de una estructura que permite recuperar y actualizar sus datos. Esa colección de información representa una o varias funciones del proceso de negocios al que asiste y entrega información que concierne sólo a este proceso para la toma de decisiones, el registro de sus operaciones y el análisis de sus procesos (Rosa, 2005).
Si bien ésta definición puede parecer algo abstracta, de su interpretación se puede inferir que una base de datos:

- Contiene información de valor para una organización;
- Permite registrar las operaciones de su interés, almacenando únicamente la información relevante, y
- Ayuda a la toma de decisiones.

Las principales ventajas que ofrece la utilización de las bases de datos relacionales son:

- Evitar la redundancia de información
- Mantener la consistencia de los datos almacenados
- Mantener y asegurar la integridad de los datos almacenados;
- Permitir el acceso concurrente a los datos, garantizando la integridad ante concurrencia de actualizaciones, y
- Brindar la posibilidad de mantener las estructuras y de desarrollar aplicaciones con mayor facilidad.

MODELO ENTIDAD/RELACIÓN

Las entidades son objetos, reales o abstractos, que existen (o pueden llegar a existir) en un contexto determinado y de las cuales deseamos guardar información. Pueden clasificarse del siguiente modo:

- Entidades fuertes: aquellas que existen por si solas y no dependen de la existencia de instancias (ejemplificación) de otras entidades. Por ejemplo, Clientes.
- Entidades débiles: aquellas en las que se hace necesaria la existencia de instancias de otras entidades distintas para que puedan existir ejemplares en esta entidad. Por ejemplo, LineasDePedido no podría existir si no existe un registro en Pedidos.

Durante el transcurso de la fase de diseño de una base de datos, es importante recoger información sobre las entidades que forman parte del modelo con el propósito de crearlas en función de su especificación y uso.

Las entidades se componen de atributos (propiedades) que aseguran que cada instancia de la tabla (fila) sea única.
Los atributos pueden ser:

- Obligatorios: aquellos que deben tener un valor asignado, para los cuales no está permitido asignarles nulos, u

- Opcionales: aquellos que pueden o no tener valor asignado.

Además, los atributos pueden ser simples o compuestos, monovaluados o multivaluados. Existen atributos denominados derivados, cuyo valor se obtiene a partir de los valores de otros atributos. Supongamos un ejemplo para ilustrarlo: TotalPedido es un atributo que surge de sumar todos los campos Valor LineaPedido de las filas de LineasdePedido para un pedido determinado.
Los atributos derivados suelen obtenerse en tiempo de ejecución de una consulta.

Una relación es una asociación entre diferentes entidades.

Con el fin de garantizar la unicidad de las tuplas, se implementa la restricción de clave o llave primaria (Primary Key) sobre uno o más de los atributos de la tabla, que permiten identificar un –y sólo un- elemento de la relación.
Debido a que en una relación puede existir más de un atributo candidato a ser designado como clave primaria, podemos identificar también las claves candidatas durante el análisis y diseño de entidades. Es usual que se utilicen códigos ficticios, autogenerados, para ser asignados al campo que contendrá la clave primaria, ya que suelen ser más fáciles de usar y ocupan menos espacio de almacenamiento.
Un atributo o un conjunto de atributos pueden ser designados como clave primaria de una relación o tabla. A esta relación entre entidad y clave primaria se le conoce como integridad de dominio. El dominio indica y restringe los valores que puede tomar un campo de la relación.

La **correspondencia de cardinalidades**, o razón de cardinalidad, expresa el número de entidades a las que otra entidad puede estar asociada vía un conjunto de relaciones.
La correspondencia de cardinalidades es la más útil describiendo conjuntos de relaciones binarias, aunque ocasionalmente contribuye a la descripción de conjuntos de relaciones que implican más de dos conjuntos de entidades.
Para un conjunto de relaciones binarias *R* entre los conjuntos de entidades *A* y *B,* la correspondencia de cardinalidades debe ser una de las siguientes:

- **Uno a uno**. Una entidad en *A* se asocia con *a lo sumo* una entidad en *B*, y una entidad en *B* se asocia con *a lo sumo* una entidad en *A*

- **Uno a varios**. Una entidad en *A* se asocia con cualquier número de entidades en *B* (ninguna o varias). Una entidad en *B*, sin embargo, se puede asociar con *a lo sumo* una entidad en *A.*

- **Varios a uno**. Una entidad en *A* se asocia con *a lo sumo* una entidad en *B*. Una entidad en *B*, sin embargo, se puede asociar con cualquier número de entidades (ninguna o varias) en *A.*

- **Varios a varios**. Una entidad en *A* se asocia con cualquier número de entidades (ninguna o varias) en *B*, y una entidad en *B* se asocia con cualquier número de entidades (ninguna o varias) en *A.*

EL MODELO RELACIONAL.

Para implementar el modelo relacional, es necesario hablar entonces de entidades denominadas tablas, que almacenan los datos, tanto del proceso al que asisten como de la propia estructura de la base de datos (metadatos). Podemos definir una entidad como cualquier objeto, real o abstracto, que existe en un contexto determinado –o que puede llegar a existir- y del cual deseamos guardar información (Rosa, 2005).
Este modelo no sólo está basado en un sólido concepto matemático (el álgebra de tuplas) sino que provee excelentes soluciones a la necesidad de procesamiento de datos basado en entidades del mundo real.

Las tablas, por lo general, representan entidades del mundo real como Empleados, Clientes o Países. Sobre estas tablas, el modelo relacional define un conjunto de operaciones matemáticas y restricciones que se utilizan para asegurar la lógica de negocios. En este esquema, un motor de bases de datos relacional cumple un rol fundamental al proveer mecanismos que aseguren el mantenimiento de las operaciones matemáticas relacionadas con el modelo.

NORMALIZACIÓN

El proceso de normalización consiste en una serie de pasos por aplicar sobre el diseño de una tabla, de manera que se pueda evitar:

- La redundancia de información;
- Las inconsistencias por eliminación de registros;
- La pérdida de dependencias funcionales por descomposición;
- La creación de relaciones o dependencias funcionales inexistentes, y
- La ineficiencia en el mantenimiento y el acceso a las relaciones.

La normalización es un término que hace referencia al conjunto de reglas metodológicas que se aplican sobre entidades con el propósito de adaptarlas al modelo relacional. Este proceso apunta a generar modelos de datos donde la redundancia de éstos no existe o se mantiene en su mínima expresión.
Fue Codd quien enunció las reglas o formas de normalización. Esta metodología permite a una entidad en estado inicial (o denormalizada) pasar a la primera forma normal (abreviada como 1FN), segunda forma normal (2FN), y así sucesivamente hasta alcanzar el grado de normalización necesario para cumplir con el objetivo de diseño.

Los objetivos de un proceso de normalización son los siguientes:

- Eliminar anomalías de actualización.
- Conservar la información (descomposición sin pérdida de información)
- Conservar las dependencias funcionales (descomposición sin pérdida de dependencia funcional).
- Proteger las propiedades inherentes al contenido semántico de los datos que deben cumplirse para cualquier extensión de la relación.
- Identificar las restricciones de integridad que verifican las interrelaciones de los atributos en el mundo real.
- Lograr invariabilidad en el tiempo.
- No crear dependencia nuevas o interrelaciones inexistentes.
- Ser eficientes.

Primera forma normal
Una tabla se encuentra en primera forma normal (1FN) si – y sólo si- cada uno de los campos contiene un único valor para un registro determinado y la tabla tiene definida una clave primaria.

Segunda forma normal
Una entidad se encuentra en segunda forma normal (2FN) si comparando todos y cada uno de los campos de la tabla con la clave definida todos dependen directamente de ésta. Por otra parte para aplicar la 2FN, la entidad ya debe estar en 1FN.

Los problemas que presentan las entidades que no están en 2FN, son:
- Presentan redundancia de datos;
- Necesitan actualizaciones en cadena;
- Se encuentran expuestas a la posibilidad de fallas en actualizaciones en cadena: inconsistencia, y
- Enfrentan la imposibilidad de almacenar algunos datos.

Tercera forma normal
Se dice que una tabla está en una tercera forma normal (3FN) si- y sólo si- los campos de la tabla dependen únicamente de la clave, es decir que los campos de las tablas no dependen unos de otros.

CONSIDERACIONES DE DISEÑO
A continuación, se detallan algunos conceptos referidos al diseño de una base de datos relacional, que nos permitirán realizar más eficientemente su implementación.

Ciclo de vida de un diseño relacional.
El ciclo de vida de desarrollo de bases de datos relacionales incluye una serie de pasos iterativos y ordenados que, partiendo de la definición de requisitos concluye en las tareas de implementación y mantenimiento.
El ciclo de vida, tal como se muestra en la Figura 1.1-1, permite reiniciarlo para tareas de mantenimiento del modelo, dado que los pasos son siempre los mismos.

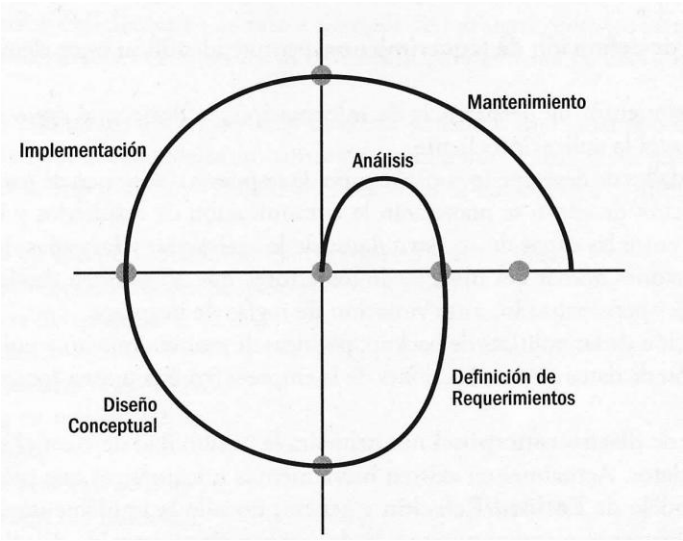


Fig.1.1-1.Diagrama del ciclo de vida de un proyecto de desarrollo de bases de datos relacionales.
Fuente: K. E. Rosa, SQL SERVER Bases de Datos Robustas y Confiables, MP Ediciones, 2005.

La etapa de análisis permite identificar los siguientes elementos.

- **Usuarios claves:** que representan al usuario consumidor de datos. En el caso de desarrollo de bases de datos, serían los programadores que son los que conocen el objetivo que persigue el usuario final.
- **Usuarios**: del sistema y niveles de seguridad.
- **Necesidades tecnológicas:** de los puestos de la aplicación cliente.
- **Viabilidad técnica**: recursos de hardware disponibles para soportar la estructura de datos.
- **Viabilidad operativa**: recursos humanos disponibles para el desarrollo.
- **Viabilidad económica**: evaluación de costos del desarrollo de la base de datos sobre el costo total del proyecto.

La etapa de **definición de requerimientos** permite identificar estos elementos.

- Requerimientos de persistencia de información, es decir, qué datos consumirá o actualizará la aplicación Cliente.
- Necesidades de desempeño como tiempo de respuesta o volumen de paquetes de red.
- Definición de cómo se manejarán la comunicación de resultados y las actualizaciones entre las capas de acceso a datos de la aplicación y las capas de negocio.
- Definiciones acerca del manejo de los errores que se generen desde el motor o mensajes personalizados ante violación de reglas del negocio.
- Definición de las políticas de respaldo, políticas de mantenimiento y políticas de propagación de datos a otras locaciones de la empresa.

La etapa de **diseño conceptual** nos brindará la posibilidad de crear el modelo de la base de datos. Actualmente, existen herramientas informáticas que permiten dibujar el modelo de Entidad/Relación y generar no sólo la implementación física, sino administrar el mantenimiento y la documentación sostenida del diseño.

La **etapa de implementación** permite derivar el modelo conceptual en el modelo físico al crear las tablas, índices, relaciones, vistas, procedimientos, condiciones de validación y planes de recuperación, replicación y mantenimiento de la base de datos.

Por último, la **etapa de mantenimiento** es la que reinicia el ciclo, puesto que para implementar cambios sobre el diseño inicial, ya implementado, es necesario efectuar primero el análisis mencionado de requisitos de usuario y luego analizar los elementos que será necesario modificar, para asegurarse de que el resto del diseño no resulte afectado.

1.2 REFACTORIZACION DE BASES DE DATOS

Refactorización de código.

En *Refactorización*, Martin Fowler (1999) describe una técnica de programación llamada refactorización, la cual es una manera disciplinada de reestructurar código en pasos pequeños. La refactorización permite evolucionar el código lentamente a través del tiempo, para tomar un enfoque evolutivo (iterativo e incremental) respecto a la programación. Un aspecto crítico de una refactorización es que mantiene el comportamiento semántico del código. No se agrega funcionalidad cuando se está refactorizando y tampoco se le quita. Una refactorización simplemente mejora el diseño del código – no más y no menos.

Una verdadera refactorización, se da cuando se cambia el código y después de haber hecho esos cambios, la aplicación funciona como antes.

Refactorización de bases de datos.

Una refactorización de bases de datos (Ambler 2003), es un cambio simple a un esquema de bases de datos el cual mejora su diseño mientras mantiene su semántica de comportamiento y su semántica informativa. En otras palabras, no se puede agregar nueva funcionalidad o romper la funcionalidad existente, tampoco se pueden agregar nuevos datos o cambiar el significado de los datos existentes.

Desde el punto de vista de Scott W. Ambler y Pramod J. Sadalage, un esquema de bases de datos incluye aspectos estructurales como por ejemplo tablas y definición de vistas, así como también incluye aspectos funcionales, como por ejemplo procedimientos almacenados y disparadores.

El proceso de refactorización de bases de datos, es el acto de realizar estos cambios simples a nuestro esquema.

Las refactorizaciones de bases de datos son conceptualmente más difíciles que las de código: las refactorizaciones de código solamente necesitan mantener la semántica del comportamiento, mientras que las refactorizaciones de bases de datos también necesitan mantener la semántica informativa. Más aún, las refactorizaciones de bases de datos pueden llegar a ser más complicadas debido al grado de acoplamiento resultante de la arquitectura de la bases de datos.

El acoplamiento es una medida de la dependencia entre dos objetos; entre más acoplamiento haya, más grande es la probabilidad de que un cambio en uno requiera un cambio en el otro. La arquitectura de una base de datos de una sola aplicación es la situación más simple – en este caso, la aplicación es la única que interactúa con la base de datos, lo cual nos permite refactorizar a ambas en paralelo y desplegarlas simultáneamente. Estas situaciones son frecuentemente referidas como aplicaciones independientes. La segunda arquitectura donde la base de datos interacciona con múltiples aplicaciones externas es mucho más complicada, y algunas de esas situaciones están más allá de nuestro control. En este caso, no se puede asumir que todos los programas externos serán desplegados a la vez, y deben por lo tanto soportar un periodo de transición (también referido como un período de aprobación) durante el cual tanto como el viejo como el nuevo esquema son mantenidos en paralelo.

Ambientes de bases de datos con una sola aplicación.

Empecemos a través del ejemplo de mover una columna de una tabla a otra en este ambiente. Esta es la situación más simple que se puede encontrar, porque se tiene el

control total sobre el esquema de la base de datos y el código de la aplicación que accesa a ella. Esto implica que se puede refactorizar a ambos, no se necesita mantener el esquema original y el nuevo esquema en paralelo porque solamente hay una aplicación que accesa a la base de datos.

En este escenario(Figura 1.2-1), los autores sugieren que dos personas trabajen en pares; una persona debería tener habilidades en la programación de aplicaciones, y la otra, habilidades en el desarrollo de bases de datos, e idealmente ambos tendrían un conjunto de habilidades. Este par empieza por determinar si el esquema de la base de datos necesita ser refactorizado. La refactorización será desarrollada y probada primeramente en un ambiente controlado de desarrollo (sandbox). Cuando es finalizada, los cambios son promovidos en el ambiente de integración de proyectos y el sistema es reconstruido, probado, y arreglado como se necesite.

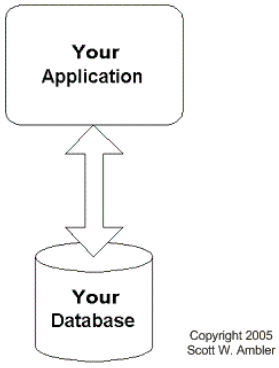


Fig.1.2-1. Ambiente de base de datos de una sola aplicación

Ambientes de bases de datos multiaplicación.

Esta situación (Figura 1.2-2) es más difícil porque las aplicaciones individuales tienen nuevos lanzamientos, los cuales serán desplegados en tiempos diferentes. Para implementar esta refactorización de base de datos se hace la misma clase de trabajo que se hizo para el ambiente de una sola aplicación, con la diferencia de que se requerirá un periodo de transición para que el equipo de desarrollo tenga tiempo para actualizar y redesplegar sus aplicaciones.

Se necesita un periodo de transición largo, porque algunas aplicaciones podrían no estar siendo trabajadas en este momento, mientras que otras podrían estar siguiendo un ciclo de vida de desarrollo tradicional y solamente hacen lanzamientos cada cierto tiempo – el periodo de transición debe de tomar en cuenta los equipos de desarrollo lentos, así como a los rápidos.

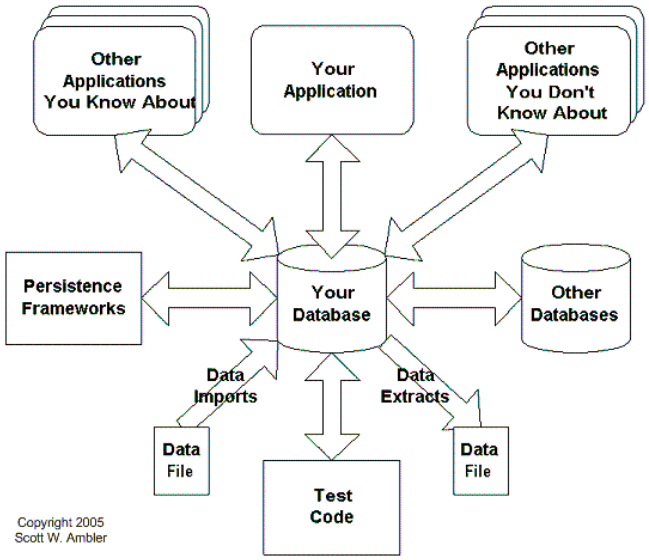


Fig.1.2-2. Ambiente de base de datos multiaplicación.

¿PORQUE SE NECESITA REFACTORIZAR UNA BASE DE DATOS?.

1. Para arreglar de manera segura bases de datos heredadas.
Las bases de datos heredadas no se van a arreglar o adaptar solas, y desde un punto de vista técnico refactorizar una base de datos es una manera segura y simple de mejorar la calidad de los datos y la base de datos.

2. Para soportar desarrollo evolutivo.
Los procesos modernos de desarrollo de software, tales como RUP o XP trabajan de una manera evolutiva. Los profesionales en esta área de datos necesitan adoptar técnicas, incluida esta, la cual los habilite para trabajar de esta manera.

MANTENIENDO LA SEMÁNTICA.

Cuando se refactoriza un esquema de base de datos, se debe mantener tanto la semántica informativa como la semántica del comportamiento, no se debería agregar nada ni tampoco quitar nada. La semántica informativa se refiere al significado de información en la base de datos desde el punto de vista de los usuarios de esa información.

Mantener la semántica informativa, implica que si se cambian los valores de los datos almacenados en una columna, los clientes de esa información no deberían ser afectados por el cambio.

Similarmente, con respecto a la semántica del comportamiento, el objetivo es mantener la funcionalidad de caja negra, cualquier código que funcione con los aspectos cambiados del esquema de la base de datos debe ser re trabajado para que cumpla la misma funcionalidad como antes.

Es importante reconocer que las refactorizaciones de bases de datos son un subconjunto de las transformaciones de bases de datos. Una transformación de bases de datos puede o no puede cambiar la semántica; una refactorización de bases de datos no la cambia.

Categorías de refactorización de bases de datos.

Existen seis diferentes categorías de refactorizaciones de bases de datos (Ambler & Sadalage, 2006), como se muestra en la tabla 1.2-1.

Categoría	Descripción	Ejemplo
Estructural.	Un cambio en la definición de una o más tablas o vistas.	Mover una columna desde una tabla a otra o dividir una columna multipropósito en varias columnas separadas, una para cada propósito.
Calidad de los datos.	Un cambio que mejora la calidad de la información contenida en la base de datos.	Hacer que una columna no contenga nulos para asegurar que siempre contenga un valor o aplicar un formato común a una columna para asegurar consistencia.
Integridad referencial.	Un cambio que asegura que un renglón referenciado existe en otra tabla y/o que asegura que a un renglón que ya no es necesitado sea removido apropiadamente.	Agregar un disparador para habilitar un borrado en cascada entre dos entidades, código que fue formalmente implementado fuera de la base de datos.
Arquitectónico.	Un cambio que mejora de manera general la forma en que los programas externos interactúan con la base de datos.	Remplazar una operación Java existente en una biblioteca de código compartido, por un procedimiento almacenado en la base de datos. Teniéndolo como un procedimiento almacenado lo hace accesible para aplicaciones que no están en Java.
Método.	Un cambio a un método (un procedimiento almacenado, una función almacenada, o un disparador) el cual mejora su calidad. Muchas refactorizaciones de código son aplicables a los métodos de las bases de datos.	Renombrar un procedimiento almacenado para hacerlo más fácil de entender.
Transformación de no refactorización.	Un cambio al esquema de la base de datos el cual cambia su semántica.	Agregar una nueva columna a una tabla existente.

Tabla 1.2-1. Categorías de refactorizaciones de bases de datos.
Fuente: Ambler, S. W., & Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley.

Indicadores para refactorizar una base de datos.

Fowler (1997) introdujo el concepto de "olores de los códigos", una categoría común de problemas en el código que indican la necesidad de refactorizarlo. Estos indicadores comunes incluyen instrucciones *switch*, métodos largos, código duplicado.

Similarmente, hay indicadores comunes en las bases de datos, que indican potencial necesidad para refactorizarlas (Ambler 2003). Entre esos indicadores se encuentran los siguientes:

- **Columnas Multipropósito**. Si una columna está siendo usada para muchos propósitos, es probable que exista código extra para asegurar que la fuente de datos está siendo usada de la manera correcta, frecuentemente revisando los valores de una o más columnas. Un ejemplo es una columna usada para almacenar la fecha de cumpleaños si él o ella es un cliente o la fecha de inicio si esa persona es un empleado. Más aún, probablemente esté restringido en la funcionalidad que ahora soporta, por ejemplo ¿cómo se almacenaría la fecha de nacimiento de un empleado?.
- **Tablas Multipropósito**. De manera similar, cuando una tabla está siendo usada para almacenar varios tipos de entidad, probablemente hay una falla en el diseño. Un ejemplo es una tabla genérica *Cliente* que es usada para almacenar información acerca de personas y empresas. El problema con este enfoque es que las estructuras de datos para personas y empresas son diferentes; por ejemplo, las personas tienen un nombre, apellido paterno y apellido materno, mientras que una empresa solamente tiene una razón social. Una tabla genérica *Cliente* podría tener columnas que son nulas para algunas clases de clientes pero no para otros.
- **Datos Redundantes**. La redundancia de datos es un problema serio en bases de datos operacionales porque cuando los datos son almacenados en varios lugares, la oportunidad para la inconsistencia ocurre. Por ejemplo, es bastante común descubrir que la información de los clientes está almacenada en muchos lugares diferentes en la organización. De hecho, muchas compañías son incapaces de juntar con seguridad una lista de quiénes son sus clientes actualmente. El problema es que en una tabla Juan Pérez vive en Filomeno Mata # 123, y en otra tabla vive en Anaxágoras # 158. En este caso, se refiere a una persona que solía vivir en Filomeno Mata # 123, pero se cambió de domicilio el año pasado; pero desafortunadamente Juan Pérez no envió los dos formatos de cambio de domicilio a la compañía, una para cada aplicación que tenía información acerca de él.
- **Tablas con demasiadas columnas**. Cuando una tabla tiene muchas columnas, esto es indicativo de la falta de cohesión de la tabla, que está tratando de almacenar datos de muchas entidades. Quizá la tabla *Cliente* contiene columnas para almacenar tres diferentes direcciones o muchos números de teléfonos. Probablemente se necesite normalizar esta estructura agregando la tabla *Dirección* y la tabla *Números de Teléfono*.
- **Tablas con demasiados renglones**. Las tablas grandes son indicativos de problemas de rendimiento. Por ejemplo, si se consume mucho tiempo cuando se busca en una tabla con millones de renglones. Se podría separar la tabla verticalmente moviendo algunas columnas a otra tabla o dividirla horizontalmente moviendo algunos renglones a otra tabla. Ambas estrategias reducen el tamaño de la tabla, mejorando potencialmente el rendimiento.
- **Columnas "Inteligentes"**. Una columna inteligente en la cual diferentes posiciones en los datos representan diferentes conceptos. Por ejemplo, si los primeros cuatro dígitos del identificador de un cliente indica el número de la casa del cliente, entonces el identificador del cliente es una columna inteligente porque se puede analizar para descubrir información más detallada. Otro ejemplo incluye una columna

de texto usada para almacenar estructuras de datos XML; claramente se puede analizar esta estructura para campos de datos más pequeños. Las columnas inteligentes frecuentemente necesitan ser organizadas en sus campos de datos a tal grado que la base de datos puede fácilmente tratarlos como elementos separados.

- **Miedo al cambio**. Si se tiene miedo de cambiar el esquema de la base de datos porque se tiene miedo de descomponer algo, por ejemplo, las cincuenta aplicaciones que la accesan, eso es en sí la señal más segura de que se necesita refactorizar el esquema. El miedo al cambio es un buen indicativo de que se tienen serios riesgos técnicos en nuestras manos, los cuales se pondrán peor con el paso del tiempo.

Es importante entender que antes de realizar una refactorización, es necesario evaluarla, pensar al respecto, y realizar la refactorización si tiene sentido.

1.3 EL PROCESO DE REFACTORIZACION DE BASES DE DATOS.

Refactorizar una base de datos es un proceso que toma 3 pasos:

1.- Empezar en el ambiente de desarrollo
2.- Implementar en nuestro ambiente de integración
3.- Instalar en producción

Paso 1. Empezar en el ambiente de desarrollo

El ambiente de desarrollo es el ambiente técnico donde el software, incluidos el código de la aplicación y el esquema de la base de datos, es desarrollado y probado. La necesidad de refactorizar el esquema de la base de datos es típicamente identificado por un desarrollador de aplicaciones quien está tratando de implementar un nuevo requerimiento o está arreglando un defecto. Por ejemplo, un desarrollador podría necesitar extender su aplicación para aceptar direcciones de correo canadienses y también direcciones americanas. La principal diferencia es que las direcciones canadienses tienen códigos postales como R2D 2C3 en lugar de códigos como 90210-1134. Si la columna destinada para este dato es numérica, no podrá soportar códigos canadienses ya que se necesita un campo de tipo cadena. El desarrollador de la aplicación describe el cambio necesario en la base de datos y se lo comunica a un DBA; aquí es donde empieza el esfuerzo para refactorizar la base de datos.
Como se ilustra en la figura(Fig.1.3-1) , el DBA y el desarrollador de la aplicación trabajarán a través de algunos o todos los siguientes pasos para implementar la refactorización:

- Verificar que una refactorización a la base de datos es requerida
- Escoger la refactorización más adecuada
- Censurar el esquema original
- Escribir unidades de pruebas
- Modificar el esquema de la base de datos
- Migrar la fuente de datos
- Actualizar los programas de acceso externo
- Actualizar los scripts de migración de datos
- Correr las pruebas de regresión
- Anunciar la refactorización
- Establecer un control de versión del trabajo

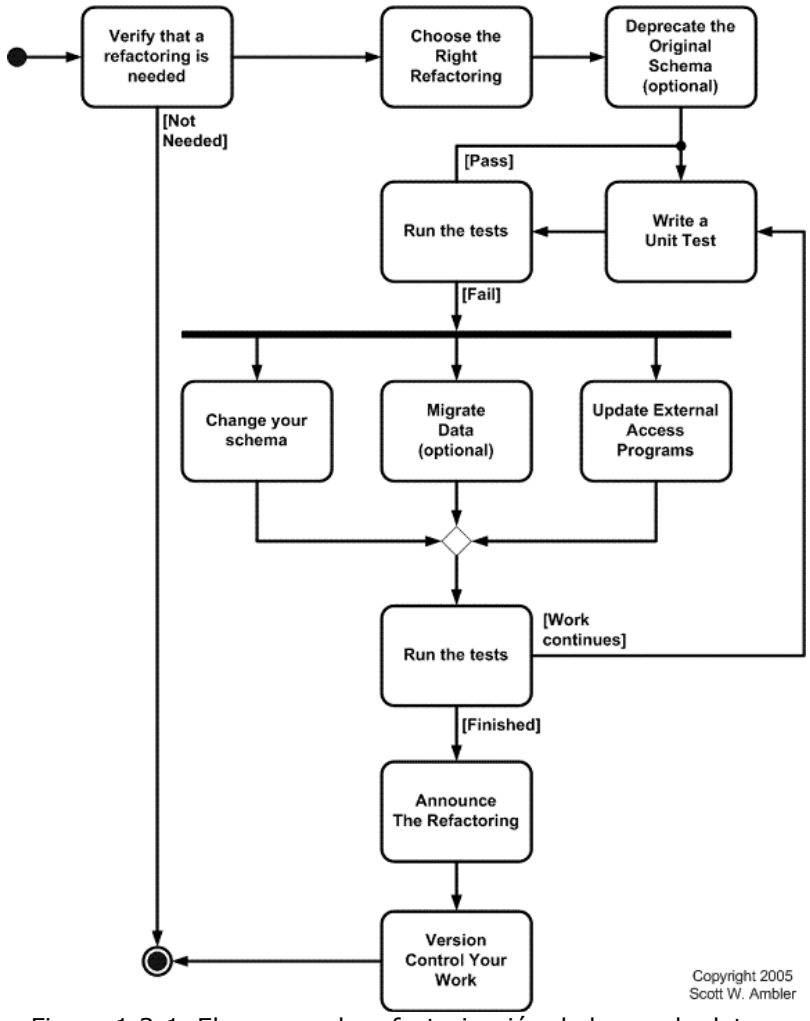


Figura 1.3-1. El proceso de refactorización de bases de datos.

Verificar que una refactorización de base de datos es requerida.
El objetivo de esto es asegurar que no se esté intentando una refactorización de base de datos que probablemente no sea posible realizar. Por ejemplo, si se va a necesitar actualizar, probar y redesplegar otras 20 aplicaciones para realizar esta refactorización, entonces probablemente no es viable continuar.

Elegir la refactorización de base de datos más adecuada
Una habilidad importante que requiere un DBA ágil es el entender que se tienen muchas opciones para implementar nuevas estructuras de datos y nueva lógica en una base de datos.

Escribir unidades de pruebas.
Así como la refactorización de código, la refactorización de base de datos está habilitada por la existencia de una amplia suite de pruebas – se sabe que es posible cambiar el esquema de la base de datos de manera segura, si se puede validar que la base de datos aún funciona después del cambio.

Censurar el esquema original
Una técnica efectiva que Pramod Sadalage y Peter Schuh (2002) promueven es un periodo de censura, aunque periodo de transición es un mejor término, para la porción original del esquema que se está cambiando. Ellos observaron que no se puede simplemente hacer los cambios instantáneamente al esquema de la base de datos. En lugar de eso, se necesita trabajar tanto con el viejo como con el nuevo esquema en paralelo y darle tiempo a los otros equipos de la aplicación para refactorizar y redesplegar sus sistemas.

La Figura 1.3-2 ilustra el ciclo de vida de una refactorización de base de datos. Primero se implementa en el alcance del proyecto y si tiene éxito eventualmente se despliega en producción. Durante el periodo de transición existen tanto el esquema original como el esquema nuevo, con suficiente código de apoyo para asegurar que cualquier actualización está correctamente soportada. Durante el periodo de transición algunas aplicaciones trabajarán con los campos nuevos y otros con los campos originales, pero no con ambos al mismo tiempo. Sin importar con qué campo estén trabajando, las aplicaciones deberían correr apropiadamente. Una vez que el periodo de transición ha expirado, el esquema original más el código de apoyo son retirados y la base de datos se prueba nuevamente. En este punto se asume que todas las aplicaciones trabajan con el campo nuevo.

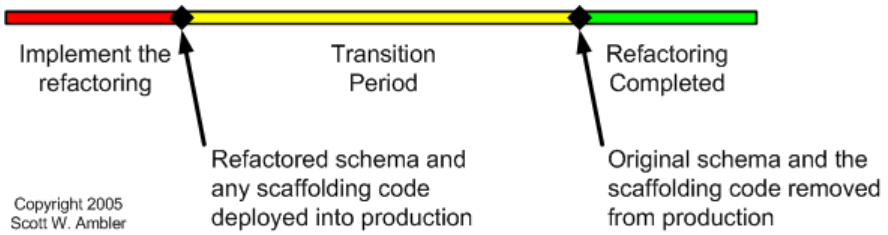


Fig.1.3-2. El ciclo de vida de una refactorización de base de datos en un escenario multiaplicación.

Modificar el esquema de la base de datos.
Para hacer esto se necesita actualizar dos bitácoras:
1.- La bitácora de cambios en la base de datos. Este es el código fuente que implementa todos los cambios en el esquema de la base de datos en el orden en que fueron aplicados a través del curso del proyecto. Cuando se está implementando una refactorización base de datos se incluyen solamente los cambios inmediatos en esta bitácora. Por ejemplo DDL para agregar columnas y triggers en el caso de la refactorización de base de datos reemplazar columna.
2.- La bitácora de actualizaciones. Esta bitácora contiene el código fuente para futuros cambios al esquema de base de datos que están para ser ejecutados después del periodo de transición para las refactorizaciones. Estos podrían ser el código fuente requerido para eliminar una columna que ya no se ocupa y los disparadores introducidos.

Migrar los datos.
Muchas refactorizaciones requieren que se migren o copien datos de la versión anterior del esquema a la nueva.

Actualizar programas externos.
Los programas que accesan a la parte del esquema de base de datos, la cual se está refactorizando debe ser actualizada para que trabaje con la nueva versión. Todos estos programas deben ser reconstruidos y entonces desplegarlos en producción antes de que el proceso de producción expire.

Correr las pruebas de regresión.
Una vez que los cambios al código de la aplicación y al esquema de las bases de datos han sido puestos en su lugar, entonces es preciso correr las pruebas de regresión. Como las pruebas exitosas descubren problemas, se necesitará retrabajar los elementos hasta obtener el resultado deseado. Una ventaja significativa de las refactorizaciones de bases de datos es que cómo son cambios pequeños los que se realizan, si al aplicar la prueba tenemos una falla, tendremos oportunidad de ubicar mejor el problema.

Avisar los cambios que se pretenden
Como la base de datos es un recurso compartido, se requiere mantener comunicación constante con los afectados, acerca de los cambios que se van a realizar.

Control de versión del trabajo
Se debe mantener un control de las versiones de cualquier DDL que se haya creado, scripts de cambios, scripts de migración de datos, datos de prueba, casos de prueba, código para generación de datos de prueba, documentación y modelos.

Paso 2. Implementación en el ambiente de integración
Es importante que se les dé tiempo a los compañeros de equipo tengan tiempo para refabricar su propio código para utilizar el nuevo esquema. El objetivo es validar que el trabajo de todos en el equipo funciona de manera conjunta.

Paso 3. Instalar en producción
Aplicar la refactorización en el ambiente de producción es difícil, principalmente cuando se tiene mucho acoplamiento en el sistema. Generalmente, las refactorizaciones se desplegarán como parte de un despliegue general de uno o más sistemas.

1.4 Liquibase

Liquibase es una biblioteca *open source*, para el seguimiento, administración y aplicación de los cambios de una base de datos (Sitio web de Liquibase).

En Liquibase, todos los cambios a la base de datos, son almacenados en archivos XML, y son identificados por una combinación de una etiqueta "id" y de una etiqueta "autor", así como el nombre del archivo mismo. Una lista de todos los cambios aplicados se guarda en cada base de datos, esta lista es consultada en todas las actualizaciones a la base de datos para determinar qué cambios necesitan ser aplicados.

Los cambios pueden ser aplicados a través de varios métodos. En este trabajo se utilizará un programa de línea de comandos.

Los cambios son almacenados en un archivo XML llamado "changelog" o bitácora de cambios. Cada entrada en el archivo changelog contiene un atributo "id" y un atributo "autor". Estos dos atributos mas el nombre y la ubicación del archivo de cambios, forman un identificador único para un cambio en particular. Cuando el migrador lo ejecuta, compara entradas en una tabla llamada "DataBaseChangeLog" definida en la base de datos. Esta tabla contiene el "id", "autor" y un identificador de archivo de todos los cambios previos. Si un cambio en el archivo log no está en esta tabla, el migrador lo ejecutará y grabará el cambio, si ya fue ejecutado será omitido durante corridas futuras.

PRINCIPALES FUNCIONALIDADES.

- Actualizar la base de datos.
- Deshacer los últimos cambios a la base de datos.
- Deshacer los cambios a la base de datos a una fecha y tiempo particular.
- "contextos" para incluir / excluir conjuntos de cambios a ejecutar.
- Reporte de las diferencias de bases de datos.
- Generación de la bitácora de cambios(changelog) de la diferencia de bases de datos.
- Habilidad para crear bitácoras de cambios (changelog) para generar una base de datos existente.
- Generación de la documentación de los cambios de la base de datos.
- Habilidad para separar la bitácora de cambios en múltiples archivos para un mejor manejo.
- Ejecutable a través de línea de comandos, Ant, Maven, Servlet o Spring.
- Soporte para diversos sistemas manejadores de bases de datos (DBMS).

BASE DE DATOS SOPORTADAS.

Debido a variaciones en los tipos de datos y en la sintaxis SQL, las siguientes bases de datos son soportadas actualmente (Sitio web de Liquibase).

- MySQL.
- PostgreSQL.
- Oracle.
- MS-SQL.
- Sybase Entreprise.
- Sybase Anywhere.
- DB2.
- Apache Derby.
- HSQL.
- H2.
- INFORMIX.
- FIREBIRD.
- SQLite.

ARQUITECTURA DE LIQUIBASE.

El migrador liquibase es el sistema principal y está diseñado para que no requiera dependencias para ejecutarse y que sea fácil de integrar en varios procesos tanto de construcción como de despliegue.

La clase primaria para liquibase es "liquibase.Liquibase". Todas las maneras de interactuar con liquibase (línea de comados, ant task, etc) son envolturas para esta clase.

Cada cambio o refactorización a la base de datos es implementado por una clase que se encuentra en el paquete liquibase.change. Tan pronto el migrador liquibase se ejecuta, usa un parser SAX XML para parsear "los changelogs", instanciar las clases de cambios necesarias y ejecutar o salvar el SQL correspondiente.

Como un proyecto "Apache-Licensed", se puede extender o integrar mientras se adapten o se mantengan a estos cambios en privado; o contribuirlos a la comunidad.

EXTENSION.

En la Figura 1.4-1 está la ruta que toma Liquibase desde el archivo changelog hasta la base de datos. Cada etapa puede ser modificada para soportar nuevas funcionalidades o para modificar la lógica existente (Sitio web de Liquibase).

Mientras Liquibase provee implementaciones por default de todos estos pasos, cualquiera puede ser omitido o anulado con una subclase personalizada. En cada paso en el proceso, Liquibase buscará clases en los paquetes conocidos (liquibasecore.jar, así como, en todos los paquetes de estructuras liquibase.ext.*) para clases que implementan las interfases correctas. Si encuentra más de una que concuerde con la interfase correcta usará un método llamado *getPriority()* que está definido para cada clase. La clase con mayor prioridad es usada. El núcleo de clases Liquibase usa prioridades de 1 a 5.

La ruta básica es:

1. Un archivo changelog es pasado a un parser changelog. El parser changelog correcto es seleccionado basado en la extensión del archivo.
2. El parser changelog convierte el texto original en objetos Change. A partir de este momento no hay conexión de regreso al archivo original changelog. Esto permite a los archivos changelog ser escritos en cualquier formato.
3. Los objetos change son ejecutados lo cual genera objetos SqlStatement. El propósito de estos objetos es describir lógicamente que tipo de sentencias serán necesitadas para ejecutar un cambio. Por ejemplo, un objeto CreateTableChange puede regresar una CreateTableStatement. Un MergeTableChange puede regresar un AddColumnStatement un UpdateStatement y un DropTableStatement.
4. Para cada objeto SqlStatement, se encontrará el SqlGenerator correcto y creará uno o más objetos Sql para la sentencia. El número de objetos Sql, puede depender de la base de datos, por ejemplo MySQL regresará un solo objeto Sql "Drop index", pero DB2 regresará una sentencia "Drop index" y una" organized table".

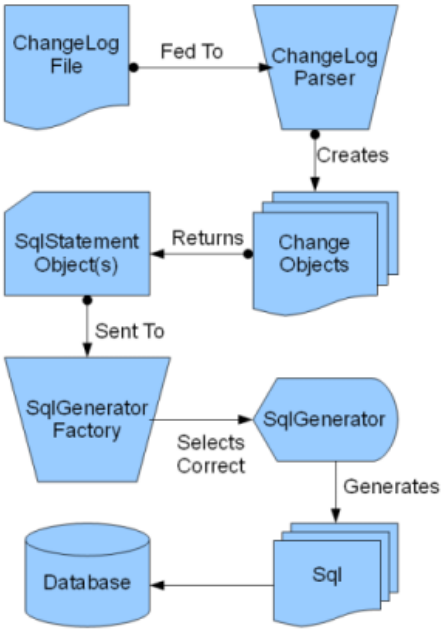


Fig.1.4-1. Ruta que toma Liquibase desde el archivo changelog hasta la base de datos. Fuente:[imagen de ruta que sigue Liquibase]. Recuperada de http://www.liquibase.org

1.4.1 Uso de Liquibase

Instalación

Como primer paso se obtiene la aplicación desde el sitio www.liquibase.org. Una vez que se obtiene el archivo se descomprime. Para saber que Liquibase está preparado para comenzar a trabajar se abre una terminal, se cambia al directorio donde se descomprimió la carpeta de Liquibase y tecleamos liquibase. Se abrirá una ventana con los comandos disponibles(Figura 1.4.1-1).

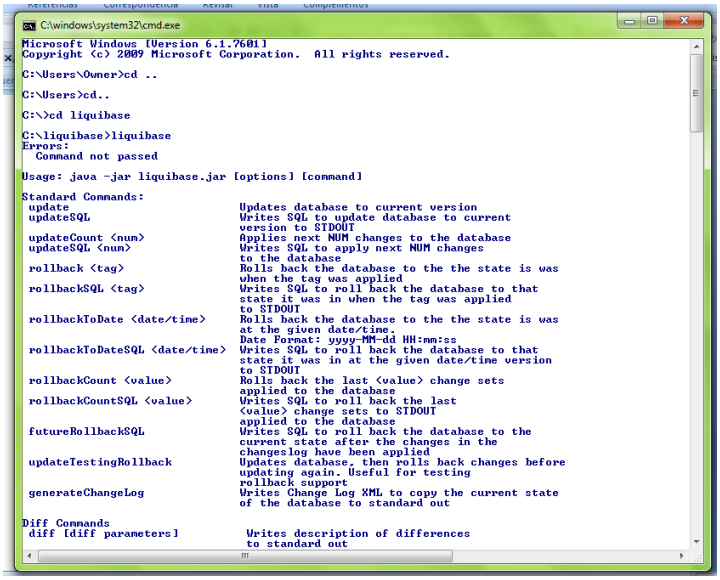


Fig.1.4.1-1. Terminal de Liquibase en funcionamiento

Construyendo el ChangeLog

El archivo changelogs.xml es la base sobre la que trabaja Liquibase para saber cuáles son los cambios que se deben realizar sobre la base de datos a refactorizar. A continuación se muestra un ejemplo en donde se crea una tabla llamada usuarios la cual está compuesta por los campos o atributos id y name.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<databaseChangeLog
   xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
   xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd
   http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd">

<preConditions>
<dbms type="postgresql" />
<runningAs username="postgres" />
</preConditions>

<changeSet author="roman" id="1">

<createTable tableName="Usuarios">
<column name="name" type="VARCHAR(255)"/>
<column name="id" type="INT">
<constraints nullable="false" primaryKey="true"/>
</column>
</createTable>

</changeSet>

</databaseChangeLog>
```

Etiquetas Liquibase

Liquibase proporciona una serie de etiquetas con las que se puede construir el conjunto de cambios, las cuales se describen a continuación.

<databaseChangeLog>

Es la etiqueta raíz a partir de la cual Liquibase comienza a analizar y ejecutar cada uno de los cambios.

<preConditions>

Comprueba que se cumplan cada una de las condiciones definidas. Si alguna de las precondiciones fallase Liquibase acabaría con un mensaje de error donde se indicará cual ha sido la condición que ha fallado. Las precondiciones son muy útiles para comprobar por ejemplo, el tipo de base de datos y el usuario que realizará las distintas modificaciones.

<changeSet>

Con esta etiqueta se definen cada uno de los cambios que se realizan en la base de datos. Por cada changeSet se insertará una nueva fila en el histórico de cambios. Por ello es recomendable hacer un solo cambio por cada changeSet, teniendo en cuenta que se permiten mas y puede tener sentido por ejemplo, si se van a insertar varias filas que forman parte de una misma transacción.
Un changeset es un conjunto de cambios que se identifica por dos atributos, el "id" y el "autor". Si solamente se utilizara el "id" para identificar un changeset, sería demasiado fácil duplicar un conjunto de cambios de manera accidental, sobre todo cuando se está trabajando dentro de un grupo de desarrollo.
Esta es una de las etiquetas más importantes y que más se utiliza con Liquibase. En su interior se pueden usar otro conjunto de etiquetas para ir definiendo cada uno de los cambios de la base de datos. En la sección "Available Database Refactorings" del manual (Sitio web de Liquibase) se puede encontrar la información sobre cada una de las etiquetas disponibles para usar en un changeSet y conformar un conjunto de cambios. A continuación se muestran algunas de las más utilizadas:

<createTable>: Nos va a permitir crear una nueva tabla en la base de datos.
Ejemplo:

```
<createTable tableName="person">
   <column name="id" type="int">
      <constraints primaryKey="true" nullable="false"/>
   </column>
   <column name="firstname" type="varchar(255)"/>
   <column name="lastname" type="varchar(255)"/>
   <column name="username" type="varchar(255)">
     <constraints unique="true" nullable="false"/>
   </column>
    <column name="testid" type="int" />
</createTable>
```

<dropTable>: Nos permite eliminar una tabla de la base de datos.
Ejemplo:

```
<dropTable tableName="person" schemaName="mySchema"/>
```

<addColumn>: Nos permite añadir una columna a una tabla.
Ejemplo:

```
<addColumn tableName="person">
   <column name="firstname" type="varchar(255)"/>
</addColumn>
```

<renameColumn>: Nos permite renombrar una columna en una tabla.
Ejemplo:

```
<renameColumn tableName="person"
   oldColumnName="fname" newColumnName="firstName"/>
```

<modifyColumn>: Nos permite modificar una columna en una tabla.
Ejemplo:

```
<modifyColumn tableName="person">
   <column name="firstname" type="varchar(5000)"/>
</modifyColumn>
```

<dropColumn>: Nos permite eliminar una columna de una tabla.
Ejemplo:

```
<dropColumn tableName="person" columnName="ssn"/>
```

<context>

Esta etiqueta básicamente permite distinguir entre los distintos entornos de ejecución. Normalmente en cualquier desarrollo suele haber distintos entornos en los que la aplicación es ejecutada (desarrollo, preproducción y producción por ejemplo). Mediante esta etiqueta se puede indicar a un changeSet a qué entorno de ejecución pertenece; después, al ejecutar Liquibase, se indicará, por parámetro, cuál es el entorno en el que se realizarán los cambios, de esta manera Liquibase solo ejecutará los changeSet que pertenezcan al entorno indicado como parámetro.

FUNCIONALIDAD DE LIQUIBASE

Las funciones que se pueden realizar con Liquibase son:

- **Update**

El *update* es la funcionalidad principal de la biblioteca. Permite aplicar los cambios en la base de datos en función del archivo changelogs.xml. Como se explicó anteriormente, cada uno de los cambios viene definido por los changeset, que se identifican por el autor y el id. Cada vez que se ejecuta un *update,* Liquibase verifica cada uno de los identificadores de los changeset. Si el identificador no existe se inserta una nueva fila en el histórico de cambios (DATABASECHANGELOG) que contiene el identificador del chageset mas un MD5Sum[1]. De esta manera Liquibase puede identificar de manera unívoca cada uno de los cambios realizados.
Si el identificador del changeset ya existe en el histórico, se compara el MD5Sum actual con el que ya existe en el histórico. Si los MD5Sum son diferentes Liquibase mostrará un error para indicar que se ha modificado de manera inesperada un changeset específico.

Si tras realizar el update, el changeset se ejecuta correctamente pero el nombre de una de las columnas está equivocado, es fácil caer en el error de modificar el changeset directamente, lo que hará que Liquibase muestre el error anteriormente mencionado ("Los MD5Sum son distintos").
Por tanto, para remediar el error cometido en la base de datos solo hay que crear un nuevo changeset con el cambio que se quiera realizar.

[1] **md5sum** es un programa originario de los sistemas Unix que tiene versiones para otras plataformas, realiza un hash MD5 de un archivo. La función de hash devuelve un valor que es prácticamente único para cada archivo, con la particularidad que una pequeña variación en el archivo provoca una salida totalmente distinta, lo que ayuda a detectar si el archivo sufrió alguna variación. Es una herramienta de seguridad que sirve para verificar la integridad de los datos.

Ejecutando update.

La primera vez que se ejecuta este comando contra una base de datos Liquibase creará dos tablas en el esquema correspondiente. Una llamada DATABASECHANGELOG y otra DATABASECHANGELOGLOCK. La primera es donde se guardarán todos los cambios y la segunda es usada por Liquibase para controlar que solo se realicen un conjunto de cambios al mismo tiempo.

- **RollBack**

Liquibase permite deshacer los cambios realizados en la base de datos, ya sea de manera automática o de manera manual. Algunos de los tags de Liquibase como <createTable>, <addColumn>, <renameColumn> llevan implícitamente un rollback, lo que quiere decir que, si por algún motivo se produce un error, se desharán los cambios de manera automática.
Otros tags como <dropTable> o <insertData> no llevan el rollback implícito por lo que habrá que definirlo de manera manual.

- **Diffs**

Diff es una utilidad que permite realizar una comparación entre dos bases de datos para encontrar diferencias entre ellas.
La mejor manera de controlar los cambios en la base de datos es mediante la adición de conjuntos de cambios durante el desarrollo. Habrá ocasiones en que será muy valioso poder realizar diffs de distintas bases de datos, principalmente cuando se está finalizando el proyecto, dado que permite comprobar que todos los cambios necesarios están incluidos en el changelog.
Es importante tener en cuenta que la ejecución del diff solo está disponible desde la línea de comandos.
En la actualidad Liquibase realiza las siguientes comparaciones:

-Diferencias de versión
-Ausencia de tablas
-Ausencia de vistas
-Ausencia de columnas
-Ausencia de llaves primarias
-Ausencia de restricciones de unicidad.
-Ausencia de llaves foráneas
-Ausencia de secuencias
-Ausencia de índices
-Diferencias en la definición de las columnas (data type, auto-increment, etc.)
-Diferencias en la definición de las vistas

- **Generate Change Logs**

Esta funcionalidad de Liquibase es muy útil cuando se comienza a trabajar con una base de datos que ya tiene creadas una serie de tablas. De tal manera, que se pueda generar el archivo changelogs.xml a partir del modelo de datos existente. Actualmente, esta

funcionalidad tiene algunas limitaciones. Por ejemplo, no exporta procedimientos almacenados, funciones ni disparadores.

- **DBDoc**

Liquibase puede generar documentación sobre los cambios realizados en la base de datos, utilizando la información existente en el histórico. La documentación generada es tipo JavaDoc.

Usando Liquibase desde la línea de comandos

Liquibase puede ser ejecutado desde la línea de comandos por medio de:
liquibase [options] [command] [command parameters]

Cualquier valor encontrado después de "command" en la invocación de la línea de comando será considerado un parámetro de éste. El procesador de la línea de comando validará si los parámetros están permitidos para dicho comando. Si éste no permite parámetros o si el parámetro tiene un formato incorrecto, se irá a la bitácora un mensaje de error de "unexpected command parameter" y se terminará la ejecución.

Los parámetros requeridos son:

- changeLogFile=<path and filename> : el archivo changelog a usar
- username=<value> : el usuario de la base de datos
- password=<value> : la contraseña del usuario de la base de datos
- url=<value> : la ruta de la base de datos
- driver=<jdbc.driver.ClassName> : el nombre de la clase del driver de la base de datos.

Se puede crear un archivo de propiedades (properties file) que contenga los valores por default si no siempre se quieren especificar opciones en la línea de comandos. Inicialmente, Liquibase buscará un archivo llamado "liquibase.properties" en el directorio actual de trabajo. Si se ha especificado una opción en un archivo de propiedades y también en la línea de comandos, el valor para esta opción, se sobrescribirá por el valor del archivo de propiedades.

CAPITULO 2: Tecnologías en Java relacionadas con XML.

2.1 CARACTERISTICAS DE JAVA

A continuación se listan algunas de las características más importantes de Java (Wang, 2000):

- **Orientación a objetos**: Java está totalmente orientado a objetos. No hay funciones sueltas en un programa de Java. Todos los métodos se encuentran dentro de clases. Los tipos de datos primitivos como enteros o dobles, tienen empaquetadores de clases, siendo estos objetos por sí mismos, lo que permite que el programa las manipule.
- **Simplicidad**. La sintaxis de Java es similar a ANSI C y C++ y, por tanto, fácil de aprender; aunque es mucho más simple y pequeño que C++. Elimina encabezados de archivos, preprocesador, aritmética de apuntadores, herencia múltiple, sobrecarga de operadores, *struct, unión* y plantillas. Además, realiza automáticamente la recolección de basura, lo que hace innecesario el manejo explícito de memoria.
- **Compactibilidad**: Java está diseñado paras ser pequeño. La versión más compacta puede utilizarse para controlar pequeñas aplicaciones. El intérprete de Java y el soporte básico de clases se mantienen pequeños al empaquetar por separado otras bibliotecas.
- **Portabilidad**: sus programas se compilan en el código de bytes de arquitectura neutra y se ejecutarán en cualquier plataforma con un intérprete de Java. Su compilador y otras herramientas están escritas en Java. Su intérprete está escrito en ANSI C. De cualquier modo, la especificación del lenguaje Java *no tiene características dependientes de la implantación.*
- **Carga y vinculación incremental dinámica**: las clases de Java se vinculan dinámicamente al momento de la carga. Por tanto, la adición de nuevos métodos y campos de datos a clases, no requieren de recompilación de clases del cliente. En C++, por ejemplo, un archivo de encabezado modificado necesita reunir todos los archivos del cliente. Además, las aplicaciones pueden ejecutar instrucciones para buscar campos y métodos y luego utilizarlos de manera correspondiente.
- **Internacionalización**: los programas de Java están escritos en *Unicode*, un código de carácter de 16 bits que incluye alfabetos de los lenguajes más utilizados en el mundo. La manipulación de los caracteres de Unicode y el soporte para fecha/hora local, etcétera, hace que Java sea bienvenido en todo el mundo.
- **Seguridad**: entre las medidas de seguridad de Java se incluyen restricciones en sus applets, implantación redefinible de sockets y objetos de administrador de seguridad definidos por el usuario. Hacen que las applets sean confiables y permiten que las aplicación es implanten y se apeguen a reglas de seguridad personalizadas.

2.2 XML

¿Qué es XML?

Podemos definir a XML, como un estándar para describir información de una manera estructurada y jerárquica mediante la utilización de etiquetas personalizadas (Sierra, 2008).

Las etiquetas no están determinadas por el estándar, sino que es el propio creador del documento el que las "inventa" en función del tipo de información que quiere describir.

En XML las etiquetas no tienen un significado predefinido, solo sirven para marcar los datos dejando libertad a las aplicaciones receptoras del documento para que interpreten y manipulen los datos según sus propias necesidades. Por ejemplo una aplicación que lea los datos del siguiente ejemplo podría utilizar esos datos para almacenarlos en una base de datos de cursos, mientras que otra los utilizaría para generar un informe para un posible alumno (Sierra, 2008).

```
<curso>
        <titulo>Programación con  XML</titulo>
        <duración>20 horas</duración>
        <comentarios>Es necesario conocer XML</comentarios>
<curso>
```

Ejemplo de un documento XML

DOCUMENTOS XML

Al conjunto formado por los datos referentes a la información que se quiere describir con sus respectivas etiquetas de marcado se le conoce como **documento XML.**

Un documento XML es un bloque de texto que puede ser tratado desde cualquier editor ASCII, de hecho pueden ser almacenados en disco como archivos de texto con extensión .xml y puede incluir cualquier flujo de datos basado en texto: un artículo de una revista, un resumen de reservaciones de avión, un conjunto de registros de una base de datos, etc.

¿Por qué utilizar XML?

Hasta ahora, Internet ha sido utilizado como medio para acceso y presentación de información, basándose en un diálogo entre el servidor de información y el elemento de presentación (navegador).

Este esquema ofrece muchas posibilidades como consulta de información, compra por internet, etc. Aún así presenta muchas limitaciones desde el punto de vista de los datos, ya que estos están codificados en un formato (HTML) que sólo es entendible por un determinado tipo de aplicación que es el navegador.

Para que la Web pueda aprovechar todo su potencial, es necesario disponer de un estándar que permita representar información estructurada en la Web, de modo que pueda ser interpretada y manipulada por diversos tipos de aplicaciones y dispositivos, sin la intervención de un usuario. Tal como se observa en la Figura 2.2-1 a continuación, ese estándar es el XML.

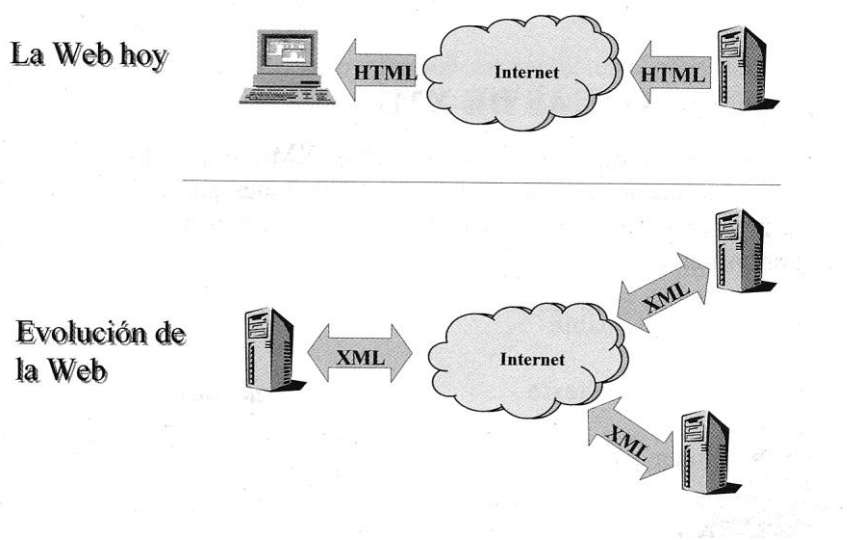


Fig.2.2-1. Estándar XML
Fuente: Sierra, A. J. (2008). *Ajax en J2EE.* Alfaomega.

XML vs HTML

Al hablar de XML puede surgir una confusión al pensar que pueda tratarse de una especie de HTML "mejorado". Aunque se parecen, (ambos se basan en la utilización de etiquetas) esto no es así ya que cada tecnología ha sido desarrollada con un objetivo diferente.

Mientras que HTML está orientado a la presentación de datos en un navegador Web, XML es más genérico al utilizar las etiquetas simplemente para delimitar piezas de datos, dejando la interpretación de estos datos a la aplicación que lee el documento.

CARACTERISTICAS DE XML

Como se mencionó anteriormente, XML viene siendo el estándar para el intercambio de información entre aplicaciones y la Web. Esto es posible gracias a las características que presenta esta tecnología de las cuales podemos resumir en tres puntos clave:

- Comprensible
- Basado en texto
- Independiente

Comprensible

Un documento XML permite describir la información de manera que ésta sea fácilmente comprensible, no sólo por una persona que lea el documento sino, lo que es más importante, por las posibles aplicaciones que lo van a procesar.

Para cualquier programa será más fácil extraer los datos del pedido en el documento que se muestra a continuación a diferencia del documento en el que no se utilizan etiquetas de marcado para identificar los datos.

Documento en formato XML:

```
<pedido>
        <material>Monitor VGA</material>
        <codigo_pedido>B-35P</codigo_pedido>
        <tel_contacto>
                        <fijo>51959635 </fijo>
                        <móvil>0445529343465 </móvil>
        <tel_contac to>
<pedido>
```

Documento sin marcado:

Monitor VGAB-35p 51959635 0445529343465

Basado en texto.

Los documentos XML están basados en texto, esto significa que no es necesario disponer de herramientas específicas de ningún fabricante para crear, interpretar y manipular información.

Comparemos por ejemplo el documento XML del ejemplo anterior con un documento escrito en Word. La diferencia es evidente; mientras que el documento XML podrá ser interpretado por cualquier aplicación, el documento escrito en Word sólo puede ser interpretado si se dispone de la aplicación específica de Microsoft Word.

Independiente

XML es una tecnología desarrollada por un organismo internacional e independiente, W3C (Word Wide Web Consortium). El W3C ha desarrollado una serie de estándares abiertos para ayudar a los programadores en el desarrollo de aplicaciones para XML. Estos estándares han servido para que los programadores, apoyándose en ellos puedan utilizar lenguajes y herramientas de desarrollo de diversos fabricantes para implementar aplicaciones que generen, manipulen e interpreten documentos XML. Esto permite que un documento generado por una aplicación escrita con un determinado lenguaje pueda ser interpretado por otra aplicación basada en una tecnología diferente.

APLICACIONES DEL XML

XML encuentra multitud de aplicaciones en el mundo de la programación, especialmente en el entorno Web. El intercambio de datos de aplicaciones es, sin duda alguna, la principal utilidad de XML, pero no la única. Entre las principales aplicaciones de XML podríamos destacar las siguientes:

- Intercambio de datos entre aplicaciones (B2B)
- Almacenamiento intermedio de datos en aplicaciones Web
- Presentación en la Web
- Utilización como base de datos

Intercambio de datos entre aplicaciones (B2B)

La principal aplicación para XML la encontramos en el intercambio de mensajes entre aplicaciones que se da al realizar transacciones comerciales entre compañías (B2B) a través de la Web. Estos mensajes son codificados mediante documentos XML.

XML es la tecnología ideal para codificar transacciones comerciales como órdenes de compra, pedidos, etc., porque representa información estructurada en la Web y puede transmitirla entre aplicaciones independientemente de la plataforma (Sierra, 2008).

Almacenamiento intermedio en aplicaciones Web.

Los datos almacenados en una base de datos se encuentran en un formato específico del fabricante, XML proporciona un soporte neutro y único para los datos, pudiéndolo transformar después en diferentes formatos dependiendo del medio al que se destina la información. Es posible generar a partir de un único documento XML documentos HTML en diversas versiones, documentos en formato para terminales móviles ó simplemente transformarlo en otro documento XML (Sierra, 2008).

Presentación en la Web.

XML, a diferencia de HTML no está orientado a la presentación.

Aún así, XML también puede aportar mucho en este campo, de hecho se han desarrollado diversos estándares basados en XML encaminados a la presentación de información en diferentes tipos de terminales, uno de ellos es XHTML (Sierra, 2008).

Utilización como base de datos

XML no pretende ser una alternativa a los sistemas de almacenamiento de datos tradicionales, de hecho, XML nunca podrá competir con estos sistemas en aspectos tales como rendimiento, seguridad, integridad de los datos, etc.

Sin embargo, en ciertos tipos de información con varios niveles de anidamiento, su almacenamiento en una base de datos relacional puede ser bastante complicado, siendo necesaria la utilización de múltiples tablas relacionadas para almacenar una cantidad pequeña de datos. Es en estos casos donde suele ser más recomendable utilizar XML en vez de una BD (Sierra, 2008).

Para acceder a esta información, las aplicaciones no necesitarán utilizar gestores de datos propios de ningún fabricante, tan sólo los estándares definidos por W3C(World Wide Web Consortium) (Sitio web de la W3C).

TECNOLOGIAS BASADAS EN XML

De todo lo anterior se desprende que las utilidades que se le pueden dar al XML son prácticamente ilimitadas. Sin embargo, nada de ello sería posible sin la existencia de toda una serie de tecnologías sobre las que se apoya este estándar, estas tecnologías pueden observarse en la Figura 2.2-2:

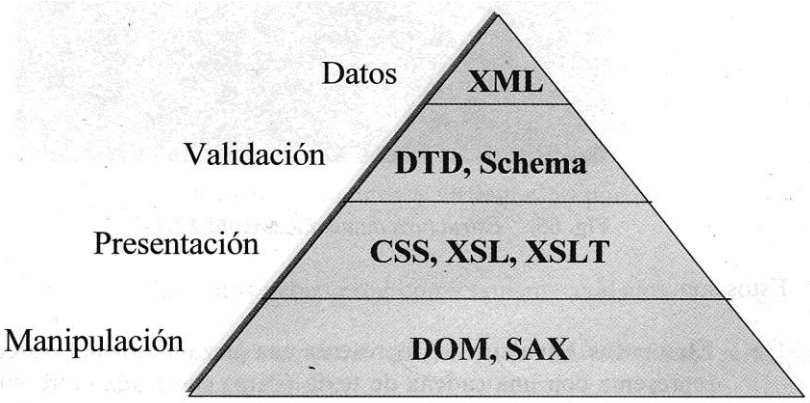


Fig.2.2-2. Tecnologías sobre las que se apoya el estándar XML.
Fuente: Sierra, A. J. (2008). *Ajax en J2EE.* Alfaomega.

Enseguida, se muestran las características desde el punto de vista sintáctico de XML.

ESTRUCTURA DE UN DOCUMENTO XML

A partir del documento de la Figura 2.2-3 que se observa enseguida, se analiza con detalle la estructura de un documento XML y sus componentes fundamentales.

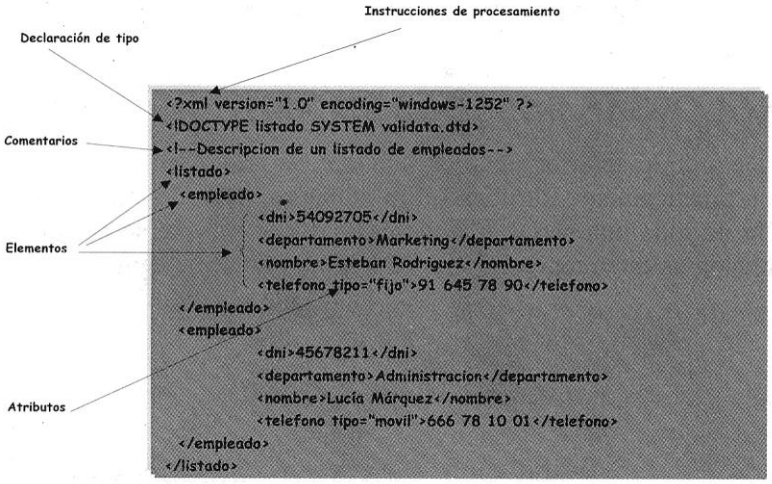


Fig.2.2-3. Estructura de un documento XML y sus componentes fundamentales.
Fuente: Sierra, A. J. (2008). *Ajax en J2EE.* Alfaomega.

Estos son:

- **Elementos**. Un elemento representa la pieza lógica del marcado, se representa con una cadena de texto (dato) encerrada entre etiquetas que pueden contener también atributos. Pueden existir elementos vacíos (
).
- **Instrucciones de procesamiento**. Se trata de órdenes especiales para ser utilizadas por la aplicación que procesa el documento XML. Comienzan por <? y terminan por ?>. La instrucción de procesamiento
-

que aparece en este ejemplo se llama **Declaración XML** y aunque no es obligatoriamente que aparezca, suele ser conveniente ponerla porque, además de identificar al documento como texto XML, aporta información relativa al mismo como versión de XML, codificación de caracteres, etc.

- **Comentarios**. Información que no forma parte del documento. Comienzan por <!—y terminan por -->.
- **Declaraciones de tipo**. Especifican información acerca del vocabulario utilizado por el documento. <!DOCTYPE persona SYSTEM "persona.dtd">
- **Atributos**. Los atributos indican características adicionales de un elemento. Se escriben dentro de la etiquetes que define al elemento antes del carácter ">". Los valores de los atributos deben escribirse entre comillas.

REGLAS SINTÁCTICAS XML

Las normas sintácticas para la construcción de documento XML están recogidas en la recomendación 1.0 del W3C (Sitio web de la W3C). A continuación se presentan algunos de los aspectos clave de esas normas:

- XML realiza distinción entre mayúsculas y minúsculas. Esto afecta a todos los componentes del documento, esto significa que no se puede escribir doctype, sino DOCTYPE, o que los elementos <persona> y <Persona> no se consideran iguales.
- Los nombres de elementos y atributos no pueden contener espacios ni signos de puntuación.
- Los elementos deben estar correctamente anidados.
- Todos los documentos deben tener un elemento principal que contenga a los demás.
- Los valores de los atributos deben estar escritos entre comillas.
- Un documento puede tener elementos vacíos, estos se representan por : <elemento/>.

DOCUMENTOS BIEN FORMADOS

Un documento bien formado es aquel que cumple con la sintaxis de XML. Cuando una aplicación recibe un documento XML, es necesario que compruebe que el documento está bien formado antes de procesarlo. Un documento mal formado puede tener errores como la omisión de la etiqueta de cierre o anidaciones incorrectas entre otros.

2.3 XSD

XSD es un formato para definir la estructura de un documento XML. XSD sustituye al anterior formato DTD, y añade funcionalidad para definir la estructura XML con más detalle (Sitio web w3schools).

Ejemplo de documento XSD:

```
1  <?xml version="1.0"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:element name="note">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element name="to" type="xs:string"/>
7         <xs:element name="from" type="xs:string"/>
8         <xs:element name="heading" type="xs:string"/>
9         <xs:element name="body" type="xs:string"/>
10      </xs:sequence>
11    </xs:complexType>
12  </xs:element>
13 </xs:schema>
```

Se puede observar, que el propio documento XSD está también escrito en XML.

Cómo hacer referencia en un documento XML a su especificación XSD

Una vez escrita una especificación XSD, se puede escribir un documento XML, y hacer constar en el mismo que debe ser conforme a dicha especificación. Esto se hace añadiendo algunos atributos al elemento raíz del documento XML.

En el ejemplo siguiente, se puede observar un documento XML que es conforme a la especificación "note.xsd" presentada anteriormente:

```
1  <?xml version="1.0"?>
2  <note
3  xmlns="http://www.openalfa.com"
4  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5  xsi:schemaLocation="http://www.openalfa.com note.xsd">
6    <to>Tove</to>
7    <from>Jani</from>
8    <heading>Reminder</heading>
9    <body>Don't forget me this weekend!</body>
10 </note>
```

En la línea 3, se indica que el "namespace" al que pertenecen los elementos que aparecen en el documento es "http://www.openalfa.com"

En la línea 4, se indica que también pueden aparecer elementos del namespace "http://www.w3.org/2001/XMLSchema-instance", y deben ir precedidos por el prefijo "xsi".

En la línea 5, el atributo xsi:schemaLocation indica la ubicación del esquema XSD contra el que se debe validar el documento XML (En este caso un documento denominado "note.xsd"). Se puede ver que este atributo utiliza el prefijo "xsi:", y por lo tanto se trata de un atributo definido en el namespace de la línea 4.

Elementos, atributos y tipos de datos simples

En el esquema se definen los elementos de que puede constar el documento XML, y los tipos de datos que pueden contener, mediante líneas de la forma:
<xs:element name="nombre_del_elemento" type="tipo_de_datos" />

Los tipos de datos más comunes son:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Un elemento también puede tener atributos. Los atributos se definen igual que los elementos, sustituyendo xs:element por xs:attribute. Ejemplos:

```
1 <xs:attribute name="lang" type="xs:string" use="required"/>
2 <xs:attribute name="lang" type="xs:string" use="optional" default="es"/>
```

También se pueden crear nuevos tipos de datos, estableciendo restricciones sobre los valores posibles de un tipo de datos predefinido. Ejemplos:

```
1   <xs:element name="age">
2     <xs:simpleType>
3       <xs:restriction base="xs:integer">
4         <xs:minInclusive value="0"/>
5         <xs:maxInclusive value="120"/>
6       </xs:restriction>
7     </xs:simpleType>
8   </xs:element>
9   <xs:element name="car">
10    <xs:simpleType>
11      <xs:restriction base="xs:string">
12        <xs:enumeration value="Audi"/>
13        <xs:enumeration value="Golf"/>
14        <xs:enumeration value="BMW"/>
15      </xs:restriction>
16    </xs:simpleType>
17  </xs:element>
18
19  <xs:element name="letter">
20  <xs:simpleType>
21  <xs:restriction base="xs:string">
22  <xs:pattern value="([a-z][A-Z])+"/>
23  </xs:restriction>
```

```
24 </xs:simpleType>
25 </xs:element>
```

Tabla 1.4-1 Restricciones aplicables a tipos de datos simples

Constraint	Descripción
enumeration	Define una lista de valores aceptables.
fractionDigits	Especifica el número máximo de decimales permitidos. Debe ser igual ó mayor a cero.
length	Especifica el número exacto de caracteres permitidos ó ítems en una lista. Debe ser igual ó mayor a cero.
maxExclusive	Especifica los límites máximos para valores numéricos. El valor debe ser menor que este valor.
maxInclusive	Especifica los límites máximos para valores numéricos. El valor debe ser menor ó igual a este valor.
maxLength	Especifica el máximo número de caracteres permitidos ó ítems en una lista. Debe ser igual ó mayor a cero.
minExclusive	Especifica el límite mínimo de valores numéricos. El valor debe ser mayor ó igual a este valor.
minInclusive	Especifica el límite mínimo para valores numéricos. El valor debe ser mayor ó igual a este valor.
minLength	Especifica el número mínimo de caracteres permitidos ó ítems en una lista. Debe ser igual ó mayor a este número.
pattern	Define la secuencia exacta de los caracteres que son aceptables.
totalDigits	Especifica el número exacto de dígitos permitidos. Debe ser mayor a cero.
whiteSpace	Especifica cuanto espacio en blanco (líneas, tabuladores, espacios y avances de renglón) se maneja.

Tipos de Datos Compuestos

Se pueden definir tipos de datos compuestos mediante elementos <xs:complexType>. Los tipos de datos compuestos contienen un conjunto de tipos de datos simples o compuestos. Estos se pueden agrupar mediante indicadores de orden como <xs:sequence> y también pueden emplear indicadores de ocurrencia, como se explica más adelante.

```
1  <xs:element name="employee" type="fullpersoninfo"/>
2
3  <xs:complexType name="personinfo">
4    <xs:sequence>
5      <xs:element name="firstname" type="xs:string"/>
```

```
6       <xs:element name="lastname" type="xs:string"/>
7     </xs:sequence>
8   </xs:complexType>
9
10  <xs:complexType name="fullpersoninfo">
11    <xs:complexContent>
12      <xs:extension base="personinfo">
13        <xs:sequence>
14          <xs:element name="address" type="xs:string"/>
15          <xs:element name="city" type="xs:string"/>
16          <xs:element name="country" type="xs:string"/>
17        </xs:sequence>
18      </xs:extension>
19    </xs:complexContent>
20  </xs:complexType>
```

En el ejemplo anterior se pueden ver las definiciones de:

- Un tipo compuesto denominado "personinfo"
- Un tipo compuesto "fullpersoninfo", definido como una extensión del tipo "personinfo". Esto se hace empleando los tags <xs:complexContent> y <xs:extension>.
- Un elemento "employee" del tipo "fullpersoninfo"

Indicadores

Existen siete indicadores que pueden ser utilizados en la definición de un tipo compuesto (Sitio web w3schools).

Indicadores de orden:

- <xs:all>…</xs:all> - Los elementos que contiene pueden aparecer en cualquier orden.
- <xs:choice>…</xs:choice> – Sólo puede aparecer uno de los elementos que contiene
- <xs:sequence>…</xs:sequence> – Los elementos que contiene deben aparecer exactamente en el mismo orden en que están definidos

Indicadores de ocurrencia:

Por defecto, los elementos definidos como parte de un tipo compuesto deben aparecer exactamente una vez. Los atributos maxOccurs and minOccurs modifican este requisito:

- maxOccurs="n" – Atributo que indica que el elemento puede aparecer varias veces, hasta un máximo de "n" veces. Si especificamos maxOccurs="unbounded", el elemento puede aparecer un número indefinido de veces.
- minOccurs="n" – Atributo que indica que el elementos debe aparecer un mínimo de "n" veces. minOccurs="0", significa que el elemento es opcional, y puede no aparecer.

Indicadores de grupo:

- Group name – Podemos definir y asignar un nombre a un grupo de elementos de la forma <xs:group name="nombre_grupo">…</xs:group>. Una vez definido un grupo, en la definición de un tipo de datos

compuesto podemos hacer referencia al mismo utilizando la sintaxis <xs:group ref="nombre_grupo"/>

- attributeGroup name – De la misma forma, podemos definir y asignar un nombre a un grupo de atributos de la forma <xs:attributeGroup name="nombre_grupo">…</xs:attributeGroup>
  Una vez definido, podemos hacer referencia a dicho grupo con la sintaxis <xs:attributegroup ref="nombre_grupo"/>

2.3 XSOM

XSOM (XML Schema Object Model) es una biblioteca de Java que permite a las aplicaciones hacer una revisión sintáctica de documentos XML y recupera información del mismo (Sitio web de XSOM). Se espera que sea muy útil para aplicaciones que necesitan tomar documentos XML como entradas.
La biblioteca es una implementación apegada al esquema de componentes definido en el documento:"the XML Schema spec part 1" de la W3C (XML Schema Part 1: Structures Second Edition).A continuación se muestra en la Figura 2.3-1 un esquema de uso de la biblioteca.

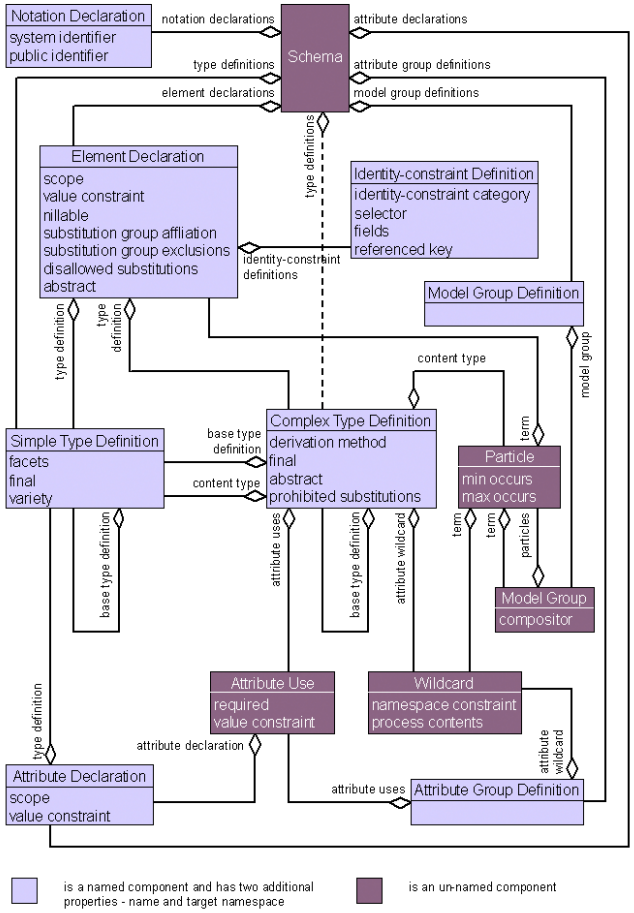


Fig.2.3-1. Modelo de datos de los componentes de un esquema XML en XSOM.
Fuente: [imagen de componentes de XSOM]. Recuperado de https://xsom.java.net

2.4 JDOM

JDOM es una herramienta de Java para trabajar con documentos XML que permite generar rápidamente desarrollo de aplicaciones XML. Su diseño se apega al lenguaje Java desde su sintaxis hasta su semántica (Sitio web de JDOM).

JDOM está optimizado y centrado para Java. Se comporta como Java, usa las colecciones de Java. Es la API completamente natural para los desarrolladores que usan Java y provee un punto de entrada de bajo costo al usar XML.

Mientras JDOM interactúa bien con los estándares existentes como SAX (Simple API for XML) y el DOM (Documento Object Model), no es una capa de abstracción ó mejoramiento para esas APIs. En su lugar, ofrece una robusta y ligera forma de leer y escribir documentos XML sin la complejidad y el consumo de memoria que tienen las APIs actuales.

JDOM provee una manera de representar un documento XML para que su lectura, escritura y manipulación sea fácil y eficiente.

CAPITULO 3: Construcción de la aplicación para generar refactorizaciones en formato XML.

3.1 DEFINICIÓN DE REQUERIMIENTOS

El modelo de requisitos tiene como objetivo delimitar el sistema y capturar la funcionalidad que ofrecerá desde la perspectiva del usuario. Este modelo puede trabajar como un contrato entre el desarrollador y el cliente, o usuario del sistema, por lo que deberá proyectar lo que el cliente desea según la percepción del desarrollador (Weitzenfeld, 2005).

3.1.1 DEFINICIÓN DEL PROBLEMA

Se realizó una búsqueda de herramientas existentes que permitieran realizar refactorización de bases de datos, encontré que Liquibase era una de estas herramientas. Las razones principales por las que se eligió esta herramienta son:

- Liquibase es una biblioteca *open source*, desarrollada en Java.
- Dentro de las herramientas existentes es la más completa.
- Liquibase es fácil de usar y configurar.
- Soporta varios manejadores de bases de datos.

Un requerimiento para que esta herramienta funcione es que uno de los parámetros necesarios es un archivo XML; dentro de este archivo debe indicarse la refactorización deseada. Dicho archivo XML debe de estar formado de acuerdo a las reglas existentes para cada refactorización; estas reglas se encuentran definidas en un esquema XML(XSD) .

El problema que se tiene en este momento es que el usuario, además de tener conocimientos de diseño de bases de datos relacionales y SQL, también debe de tener conocimientos de archivos XML y esquemas XML para poder realizar una refactorización (Figura 3.1.1-1), debido a esto:
Se requiere el desarrollo de una herramienta o aplicación que automatice y facilite la creación de los archivos XML(changelogs[2]), tomando en cuenta la reglas del esquema XSD (Figura 3.1.1-2). El desarrollo de esta herramienta no hará necesario que el usuario tenga conocimientos previos en estos temas para llevar a cabo una refactorización en la base de datos y disminuirá la cantidad de errores que se pudieran presentar si estos archivos se crearan de manera manual.

Situación actual

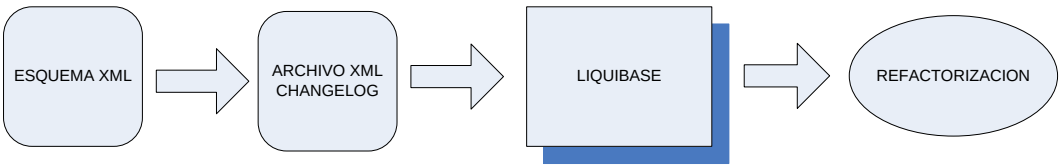


Fig. 3.1.1-1 Flujo que se sigue para crear un archivo changelog de manera manual.

Situación final

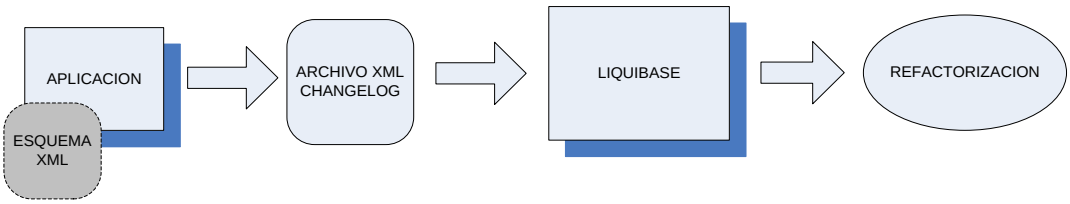


Fig. 3.1.1-2 Flujo que se sigue para crear un archivo changelog de manera automatizada.

[2] Este tipo de archivos se describen con mayor detalle en la sección Liquibase.

3.1.2 DIAGRAMA GENERAL DE CASOS DE USO

El modelo de casos de uso describe un sistema en términos de sus distintas formas de utilización, cada una de las cuales se conoce como un caso de uso.
Cada caso de uso o flujo se compone de una secuencia de eventos iniciada por el usuario. Para comprender los casos de uso de un sistema primero es necesario saber quiénes son sus usuarios. Para ello, se define el concepto de actor, que es el tipo de usuario que está involucrado en la utilización de un sistema, y que es además una entidad externa al propio sistema. Juntos el actor y el caso de uso, representan los dos elementos básicos de este modelo (Weitzenfeld, 2005).

La Figura 3.1.2-1 muestra el diagrama de casos de uso general para la aplicación a desarrollar.

Este diagrama está compuesto por 8 casos de uso en los cuales el actor principal es el DBA o Administrador de Bases de Datos.

Los casos de uso: "Generar Refactorización de Arquitectura", "Generar Refactorización de Calidad de los Datos", "Generar Refactorización de Integridad Referencial", "Generar Refactorización de Estructura" y "Generar Refactorización de Transformación", representan una generalización, ya que estos casos de uso se componen de otros casos de uso. Por ejemplo, el caso de uso Generar Refactorización de Arquitectura está compuesto de los casos de uso: Create Index y Drop Index.

Los casos de uso Create Index y Drop Index incluyen a su vez los casos de uso: Generar Elemento Preconditions y Generar Elemento ChangeSet[3].

Los casos de uso Generar Elemento Preconditions y Generar Elemento ChangeSet son generados de manera automática por el sistema y se incluyen en cada uno de los elementos que se derivan de los casos de uso "Generar Refactorización".

[3] En la sección Detalles de casos de uso, se describen con más detalle estos elementos.

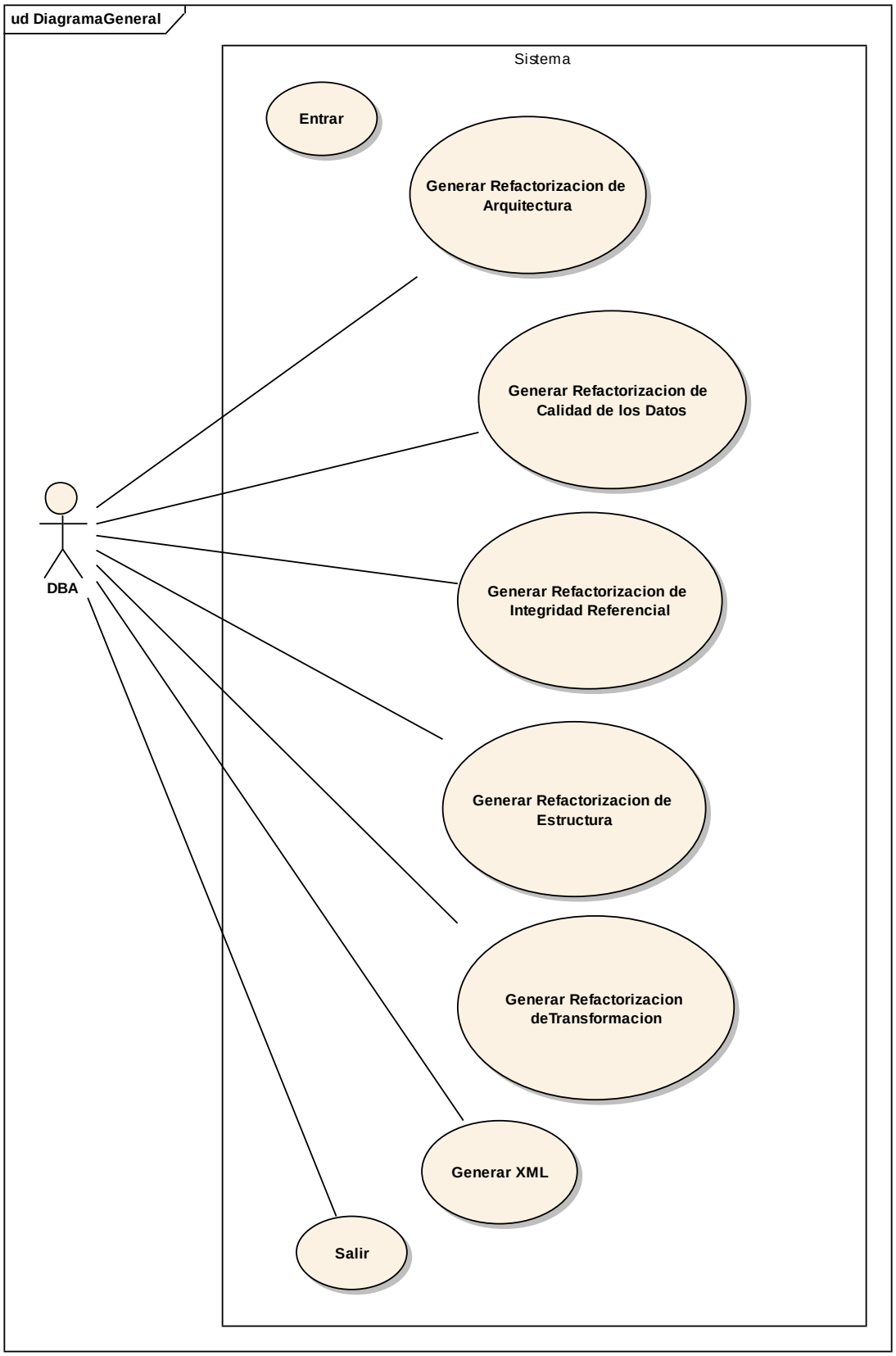


Fig.3.1.2-1. Diagrama de casos de uso general del sistema a desarrollar.

3.1.3 PROTOTIPO

En esta sección se muestra el prototipo generado para la aplicación web.
Pantalla principal (Figura 3.1.3-1) en la cual se muestra el menú principal y la pantalla con los campos que se tienen que llenar para el elemento de Precondiciones.

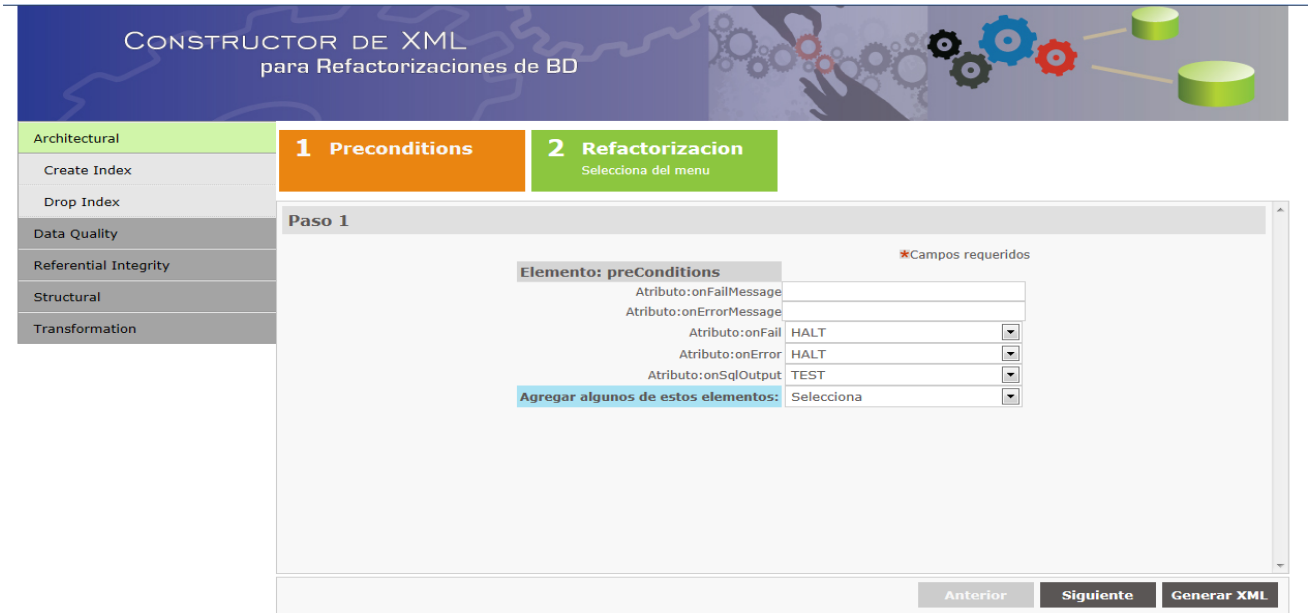


Fig. 3.1.3-1. Pantalla principal del sistema

Pantalla que se obtiene después de seleccionar el **paso 2 (Refactorización)** o cuando se presiona el botón **siguiente** de la pantalla principal (Figura 3.1.3-2).

Fig. 3.1.3-2 Pantalla correspondiente al paso 2 (Refactorización)

3.1.4 DETALLE DE CASOS DE USO

La especificación o detalle de casos de uso permite visualizar, especificar y documentar el comportamiento de un elemento. Presentan una vista de cómo pueden utilizarse estos elementos en un contexto o escenario dado.

Sólo se muestran los casos de uso más relevantes, dado que la funcionalidad de los demás casos de uso es similar.

CASO DE USO: Entrar

Actor: DBA

Descripción: El actor debe teclear la URL del sitio Web, para poder visualizar la página principal, véase Tabla 3.1.4-1.

Precondiciones: El actor cuenta con un equipo de cómputo, una conexión a Internet y un navegador Web.

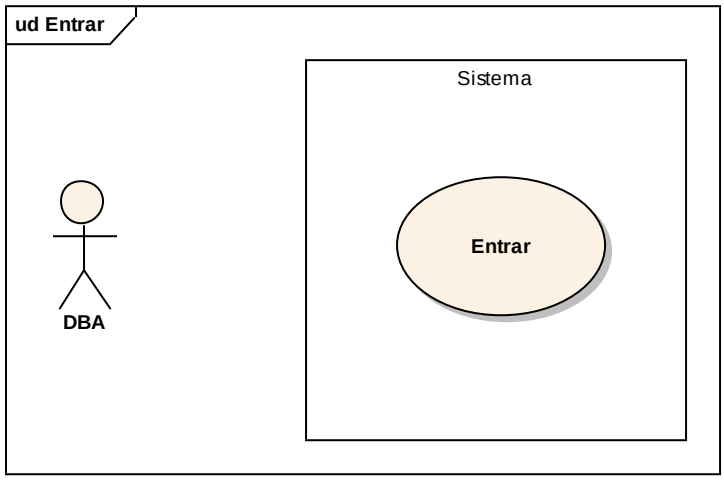


Fig. 3.1.4-1 Diagrama UML del caso de Uso: Entrar

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Teclear la URL en la barra de direcciones del Navegador Web.	2	Despliega la pantalla principal.	

Tabla 3.1.4-1 Detalle de caso de uso entrar.

Postcondiciones: El actor se encuentra en la pantalla principal del sistema Web.

CASO DE USO: Generar Elemento Preconditions

Actor: DBA

Descripción: El actor necesita incorporar un elemento Preconditions al archivo XML de la refactorización que se aplicará, véase Tabla 3.1.4-2

Precondiciones: El actor entró al sistema.

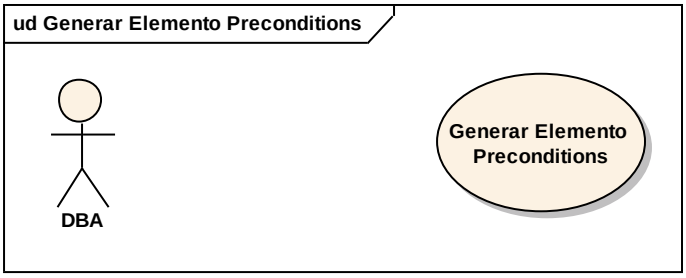


Fig. 3.1.4-2 Diagrama UML del Caso de Uso: Generar Elemento Preconditions

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
		1	Muestra la pantalla, con los campos que deben llenarse para crear este elemento	
2	Llena los campos correspondientes			E1
3	Selecciona un elemento a agregar de la lista	4	Muestra la pantalla, con los campos que deben llenarse para crear este elemento	E1
5	Llena los campos correspondientes			E1
6	Se repite el paso 4 hasta que se hayan agregado por lo menos los elementos "dbms" y "running As".			
7	Se presiona el botón "Siguiente"	8	Muestra la pantalla en la cual se indica que se debe seleccionar una opción (refactorización) del menú.	

Tabla 3.1.4-2 Detalle del caso de uso Generar Elemento Preconditions.

Excepciones:

Id	**Nombre**	**Acción**
E1	El actor presiona el botón "Siguiente"	Muestra la pantalla en la cual se indica que se debe seleccionar una opción (refactorización) del menú.

Tabla 3.1.4-3 Excepciones del caso de uso Generar Elemento Preconditions.

Postcondiciones: Se tiene el elemento Preconditions con los atributos mínimos requeridos llenos, listo para incorporarlo al archivo XML.

CASO DE USO: Generar Elemento ChangeSet

Actor: DBA

Descripción: El actor necesita incorporar un elemento ChangeSet al archivo XML de la refactorización que se aplicará, véase Tabla 3.1.4-4

Precondiciones: El actor entró al sistema.

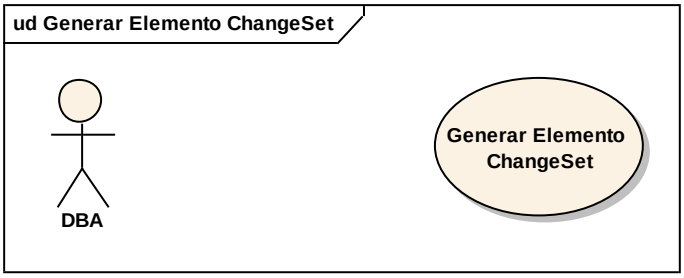


Fig. 3.1.4-3 Diagrama UML del Caso de Uso: Generar Elemento ChangeSet

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Presiona cualquier opción disponible(refactorización) del menú principal de la pantalla principal	2	Muestra la pantalla, con los campos que deben llenarse para crear este elemento	
3	Llena los campos correspondientes (por lo menos los campos id y autor)			E1

Tabla 3.1.4-4 Detalle del caso de uso Generar Elemento ChangeSet.

Excepciones:

Id	**Nombre**	**Acción**
E1	El actor presiona el botón "Anterior"	Muestra la pantalla en la cual se muestran los campos a llenar para el elemento Preconditions

Tabla 3.1.4-5 Excepciones del caso de uso Generar Elemento ChangeSet.

Postcondiciones: Se tiene el elemento ChangeSet con los atributos mínimos requeridos llenos, listo para incorporarlo al archivo XML.

CASO DE USO: Generar Refactorización de Arquitectura

Actor: DBA

Descripción: El DBA necesita generar un archivo XML para poder realizar un refactorización de Arquitectura, véase Tabla 3.1.4-6

Precondiciones: El DBA se encuentra en la pantalla principal del sistema.

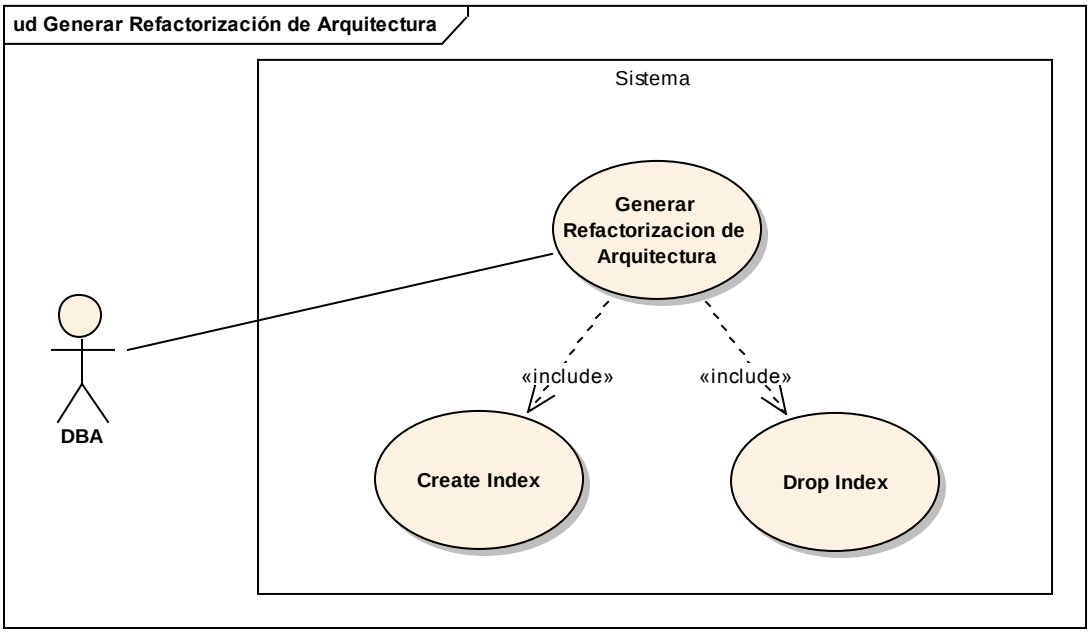


Fig. 3.1.4-4 Diagrama UML del Caso de Uso: Generar Refactorización de Arquitectura

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Ver las opciones(refactorizaciones) del menú principal de la pantalla principal del sistema.			
2	El DBA puede elegir entre crear una refactorización Create Index o Drop Index	3	Dependiendo de la opción elegida, se despliega la pantalla correspondiente	
A continuación de esto se realiza la transacción correspondiente; cuyos pasos estarán descritos en los casos de uso incluidos dentro de éste.				

Tabla 3.1.4-6 Detalle del caso de uso Generar Refactorización de Arquitectura

Postcondiciones: Se genera un archivo XML con la refactorización elegida de esta categoría elegida.

CASO DE USO: Create Index

Actor: DBA

Descripción: El DBA necesita crear una refactorización con la cual crea un índice en una tabla, véase Tabla 3.1.4-7.

Precondiciones: El actor entró al sistema.

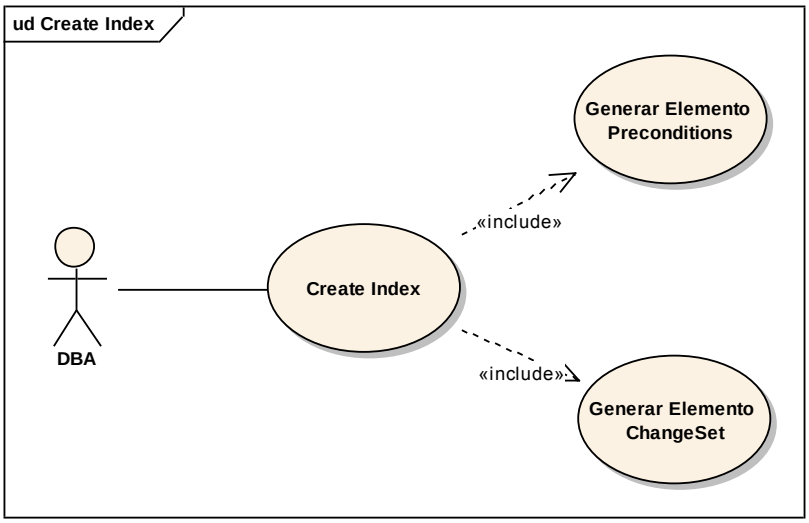


Fig. 3.1.4-5 Diagrama UML del Caso de Uso: Create Index

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Elige la opción "Create Index".	2	Muestra la pantalla, con los campos que deben llenarse para crear este elemento y sus subelementos	
3	Llena los campos requeridos para este elemento y llena los campos requeridos para sus subelementos.			E1
4	Presiona el botón "+" para generar otro subelemento igual que el mostrado.	5	Muestra la pantalla, con los campos que deben llenarse para crear este elemento y sus subelementos	E1
6	Repite el paso 4 hasta que haya agregado los subelementos necesarios			
7	Si agrego subelementos puede presionar botón "-" para quitarlos	8	Esconde la pantalla con los campos que deben llenarse para crear este elemento y sus subelementos	E1
9	Repite el paso 6 hasta que haya quitado los subelementos innecesarios			

Tabla 3.1.4-7 Detalle del caso de uso Create Index

Excepciones:

Id	Nombre	Acción
E1	El actor presiona el botón "Anterior".	Muestra la pantalla en la cual se muestran los campos a llenar para el elemento Preconditions

Tabla 3.1.4-8 Excepciones del caso de uso Create Index

Postcondiciones: Se tienen llenos los campos de los elementos necesarios para poder generar el archivo XML de la refactorización correspondiente.

CASO DE USO: Drop Index

Actor: DBA

Descripción: El DBA necesita crear una refactorización con la cual elimina un índice de una tabla, véase Tabla 3.1.4-9.

Precondiciones: El actor entró al sistema.

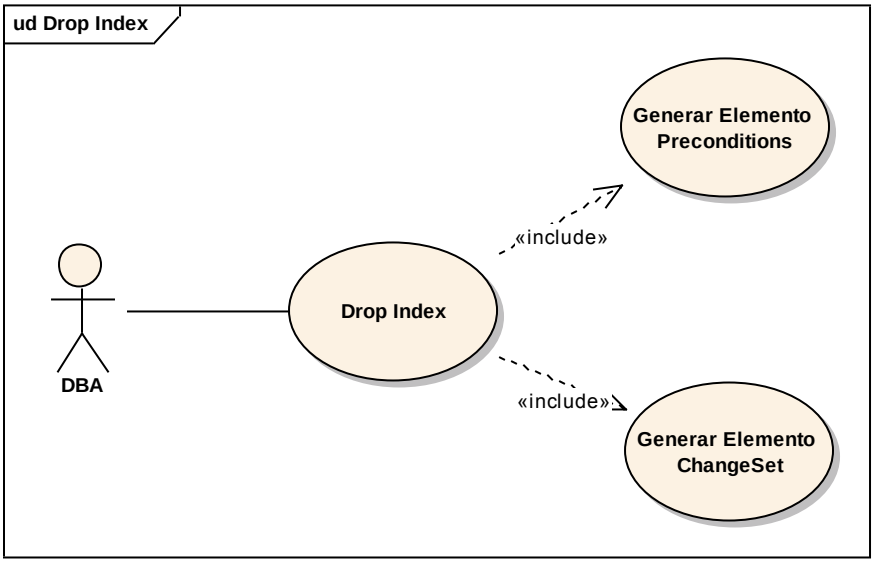


Fig. 3.1.4-6 Diagrama UML del Caso de Uso: Drop Index

Actor		Sistema		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Elige la opción "Drop Index".	2	Muestra la pantalla, con los campos que deben llenarse para crear este elemento y sus subelementos	
3	Llena los campos requeridos para este elemento y llena los campos requeridos para sus subelementos.			E1

Tabla 3.1.4-9 Detalle del caso de uso Drop Index

Excepciones:

Id	Nombre	Acción
E1	El actor presiona el botón "Anterior".	Muestra la pantalla en la cual se muestran los campos a llenar para el elemento Preconditions

Tabla 3.1.4-10 Excepciones del caso de uso Drop Index

Postcondiciones: Se tienen llenos los campos de los elementos necesarios para poder generar el archivo XML de la refactorización correspondiente.

CASO DE USO: Generar Refactorización de Calidad de los Datos

Actor: DBA

Descripción: El DBA necesita generar un archivo XML para poder realizar un refactorización de Calidad de los Datos, véase Tabla 3.1.4-11.

Precondiciones: El DBA se encuentra en la pantalla principal del sistema.

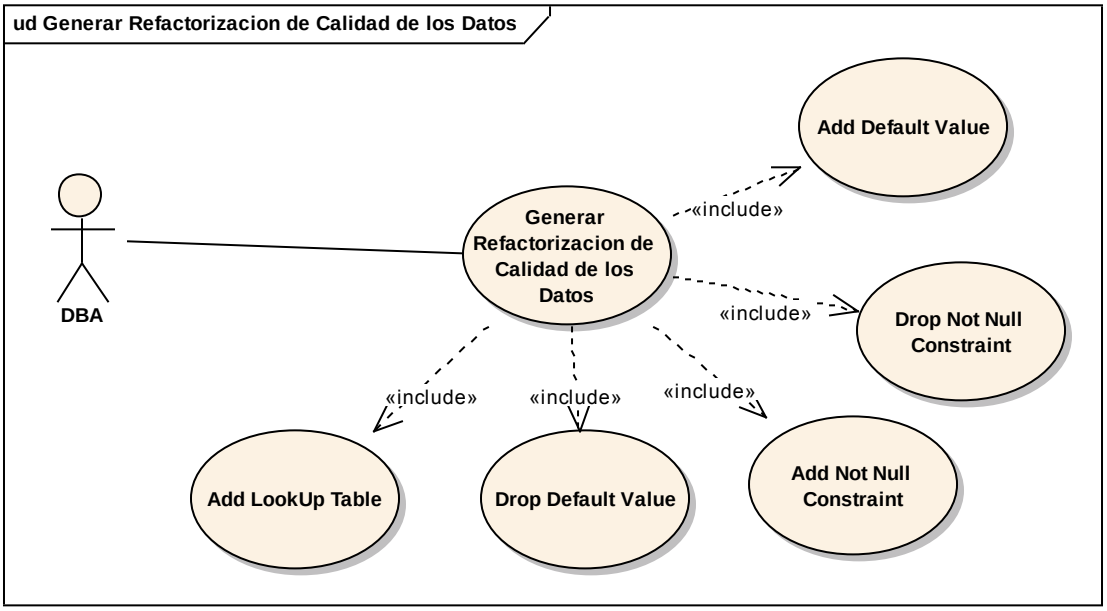


Fig. 3.1.4-7 Diagrama UML del Caso de Uso: Generar Refactorización de Calidad de los Datos

Actor		Sistema		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Ver las opciones del menú izquierdo de la pantalla principal del sistema.			
2	El DBA puede elegir entre crear una refactorización Add LookUp Table, Drop Default Value, Add Not Null Constraint, Drop Not	3	Dependiendo de la opción elegida, se despliega la pantalla correspondiente	

	Null Constraint, Add Default Value o Drop Default Value.			
A continuación de esto se realiza la transacción correspondiente, cuyos pasos estarán descritos en los casos de uso incluidos dentro de éste.				

Tabla 3.1.4-11 Detalle del caso de uso Generar Refactorización de Calidad de los Datos

Postcondiciones: Se genera un archivo XML con la refactorización elegida de esta categoría elegida.

CASO DE USO: Add Lookup Table

Actor: DBA

Descripción: El DBA necesita crear una tabla de búsqueda para una columna existente, véase Tabla 3.1.4-12.

Precondiciones: El actor entró al sistema.

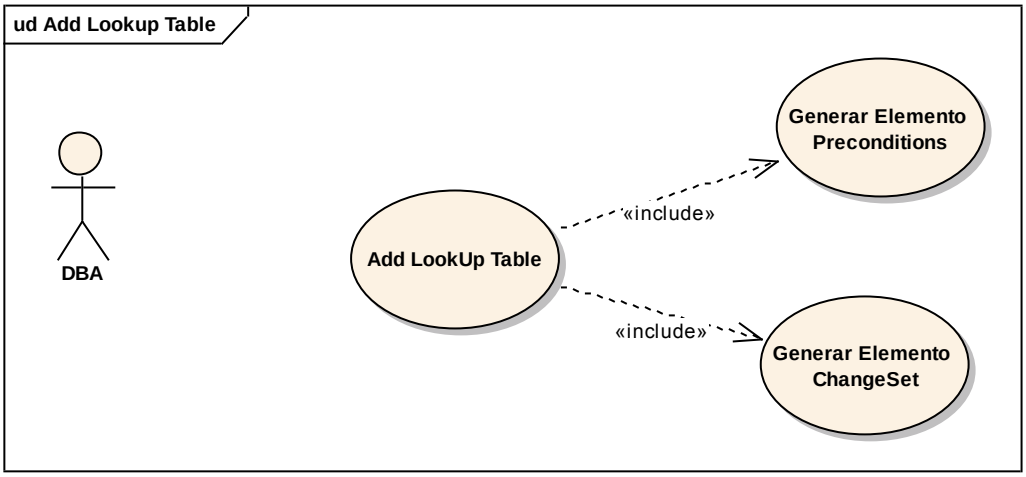


Fig. 3.1.4-8 Diagrama UML del Caso de Uso: Add LookUp Table

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Elige la opción "Add Lookup Table".	2	Muestra la pantalla, con los campos que deben llenarse para crear este elemento	
3	Llena los campos requeridos para este elemento			E1

Tabla 3.1.4-12 Detalle del caso de uso Add LookUp Table

Excepciones:

Id	Nombre	Acción
E1	El actor presiona el botón "Anterior".	Muestra la pantalla en la cual se muestran los campos a llenar para el elemento Preconditions

Tabla 3.1.4-13 Excepciones del caso de uso Add LookUp Table

Postcondiciones: Se tienen llenos los campos de los elementos necesarios para poder generar el archivo XML de la refactorización correspondiente.

CASO DE USO: Drop Default Value

Actor: DBA

Descripción: El DBA necesita crear una refactorización con la cual elimine un valor por default de un campo de una tabla, véase Tabla 3.1.4-14.

Precondiciones: El actor entró al sistema.

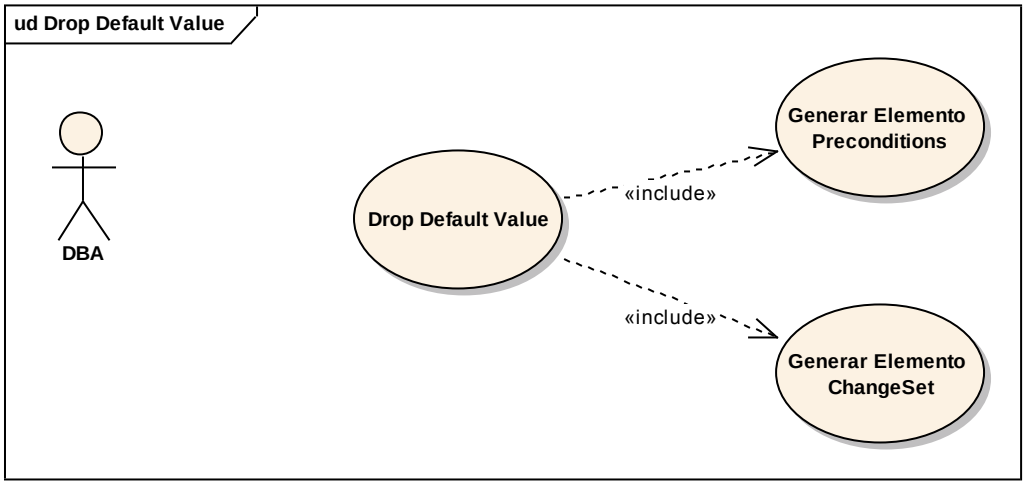


Fig. 3.1.4-9 Diagrama UML del Caso de Uso: Drop Default Value

Actor		Sistema		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Elige la opción "Drop Default Value".	2	Muestra la pantalla, con los campos que deben llenarse para crear este elemento	
3	Llena los campos requeridos para este elemento			E1

Tabla 3.1.4-14 Detalle del caso de uso Drop Default Value

Excepciones:

Id	Nombre	Acción
E1	El actor presiona el botón "Anterior".	Muestra la pantalla en la cual se muestran los campos a llenar para el

		elemento Preconditions

Tabla 3.1.4-15 Excepciones del caso de uso Drop Default Value

Postcondiciones: Se tienen llenos los campos de los elementos necesarios para poder generar el archivo XML de la refactorización correspondiente.

CASO DE USO: Add Not Null Constraint

Actor: DBA

Descripción: El DBA necesita crear una refactorización con la cual agregue una restricción de que no permita nulos a un campo de una tabla, véase Tabla 3.1.4-16.

Precondiciones: El actor entró al sistema.

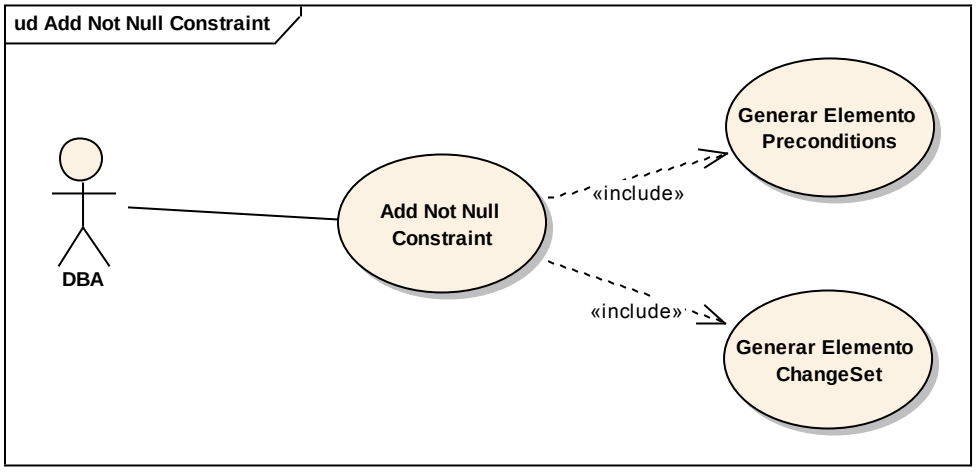


Fig. 3.1.4-10 Diagrama UML del Caso de Uso: Add Not Null Constraint

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Elige la opción "Add Not Null Constraint".	2	Muestra la pantalla, con los campos que deben llenarse para crear este elemento	
3	Llena los campos requeridos para este elemento			E1

Tabla 3.1.4-16 Detalle del caso de uso Add Not Null Constraint

Excepciones:

Id	**Nombre**	**Acción**
E1	El actor presiona el botón "Anterior".	Muestra la pantalla en la cual se muestran los campos a llenar para el elemento Preconditions

Tabla 3.1.4-17 Excepciones del caso de uso Add Not Null Constraint

Postcondiciones: Se tienen llenos los campos de los elementos necesarios para poder generar el archivo XML de la refactorización correspondiente.

CASO DE USO: Drop Not Null Constraint

Actor: DBA

Descripción: El DBA necesita crear una refactorización con la cual elimine una restricción de que no permita nulos a un campo de una tabla, véase Tabla 3.1.4-18.

Precondiciones: El actor entró al sistema.

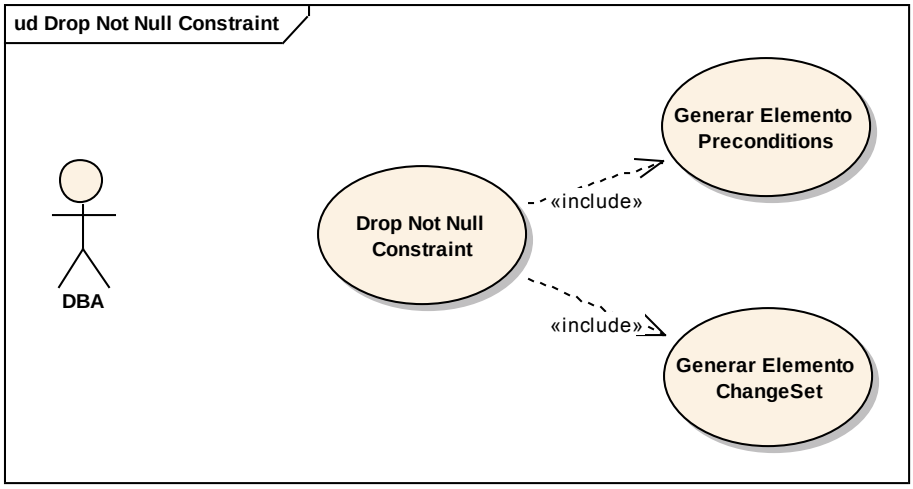


Fig. 3.1.4-11 Diagrama UML del Caso de Uso: Drop Not Null Constraint

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Elige la opción "Drop Not Null Constraint".	2	Muestra la pantalla, con los campos que deben llenarse para crear este elemento	
3	Llena los campos requeridos para este elemento			E1

Tabla 3.1.4-18 Detalle del caso de uso Drop Not Null Constraint

Excepciones:

Id	**Nombre**	**Acción**
E1	El actor presiona el botón "Anterior".	Muestra la pantalla en la cual se muestran los campos a llenar para el elemento Preconditions

Tabla 3.1.4-19 Excepciones del caso de uso Drop Not Null Constraint

Postcondiciones: Se tienen llenos los campos de los elementos necesarios para poder generar el archivo XML de la refactorización correspondiente.

CASO DE USO: Add Default Value

Actor: DBA

Descripción: El DBA necesita crear una refactorización con la cual agregue un valor por default a un campo de una tabla, véase Tabla 3.1.4-20

Precondiciones: El actor entró al sistema.

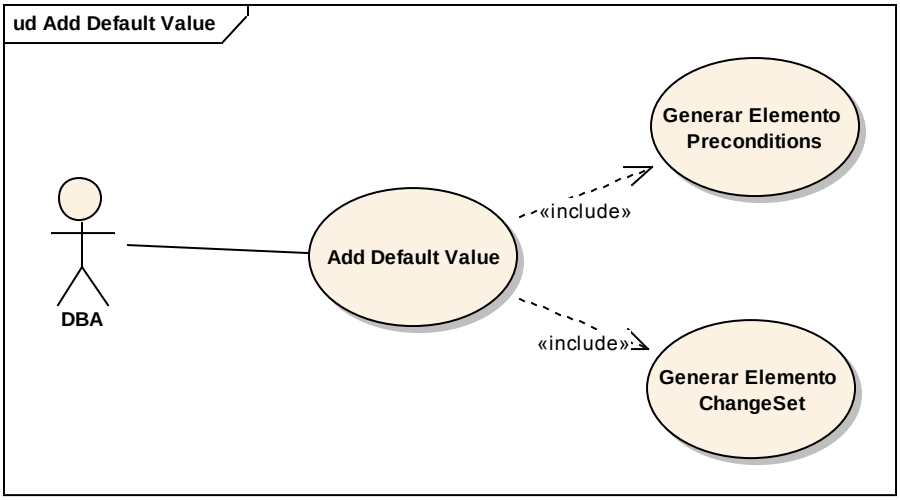


Fig. 3.1.4-12 Diagrama UML del Caso de Uso: Add Default Value

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Elige la opción "Add Default Value".	2	Muestra la pantalla con los campos que deben llenarse para crear este elemento	
3	Llena los campos requeridos para este elemento			E1

Tabla 3.1.4-20 Detalle del caso de uso Add Default Value

Excepciones:

Id	**Nombre**	**Acción**
E1	El actor presiona el botón "Anterior".	Muestra la pantalla en la cual se muestran los campos a llenar para el elemento Preconditions

Tabla 3.1.4-21 Excepciones del caso de uso Add Default Value

Postcondiciones: Se tienen llenos los campos de los elementos necesarios para poder generar el archivo XML de la refactorización correspondiente.

CASO DE USO: Generar XML

Actor: DBA

Descripción: El DBA necesita crear un archivo XML con los elementos necesarios para llevar a cabo una refactorización, véase Tabla 3.1.4-22.

Precondiciones: El actor llenó los campos requeridos en los elementos: Preconditions, ChangeSet y la refactorización a realizar.

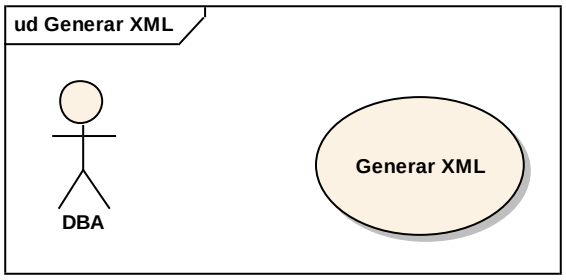


Fig. 3.1.4-13 Diagrama UML del Caso de Uso: Generar XML

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Presiona el botón "Generar XML".	2	Crea un archivo XML con los elementos Precondiciones, ChangeSet y la refactorización a realizar.	E1

Tabla 3.1.4-22 Detalle del caso de uso Generar XML

Excepciones:

Id	**Nombre**	**Acción**
E1	Si existen campos obligatorios vacíos	Muestra un aviso en el cual se informa que hay campos obligatorios vacíos.

Tabla 3.1.4-23 Excepciones del caso de uso Generar XML

Postcondiciones: Se tiene un archivo XML de la refactorización correspondiente y el sistema vuelve a mostrar la pantalla del elemento Preconditions para poder realizar una nueva refactorización.

CASO DE USO: Salir

Actor: DBA

Descripción: El actor desea abandonar el sistema, véase Tabla 3.1.4-23.

Precondiciones: El actor se encuentra en la página principal.

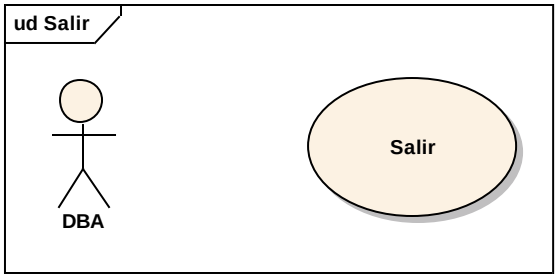


Fig. 3.1.4-14 Diagrama UML del Caso de Uso: Salir

Actor		**Sistema**		
Paso	**Acción**	**Paso**	**Acción**	**Excepción**
1	Presionar el botón cerrar del navegador		Se cierra la aplicación	

Tabla 3.1.4-23 Detalle del caso de uso Salir

Postcondiciones: El actor cerró la aplicación.

3.2 ANALISIS

Cuando ya se ha desarrollado y aceptado el modelo de requisitos, se inicia el desarrollo del modelo de análisis siguiendo el modelo de casos de uso, cuyo objetivo es comprender y generar una arquitectura de objetos para el sistema con base en lo especificado en el modelo de requisitos (Weitzenfeld, 2005). Durante esta etapa no se considera el ambiente de implementación, modelando el sistema bajo condiciones ideales. Tarde o temprano el sistema tendrá que adaptarse a las condiciones de implementación deseadas, lo que se hará durante el modelo de diseño.

Es importante enfatizar que el modelo de análisis es una representación conceptual, correspondiente al problema y modelo de requisitos, en término de clase de objetos.

3.2.1 DIAGRAMAS DE CLASES

Los diagramas de clases ayudan a representar elementos involucrados en cada capa del sistema y su relación entre ellos. Para esta aplicación se agruparon en tres capas de acuerdo a su función principal.

Clases de Interfaz

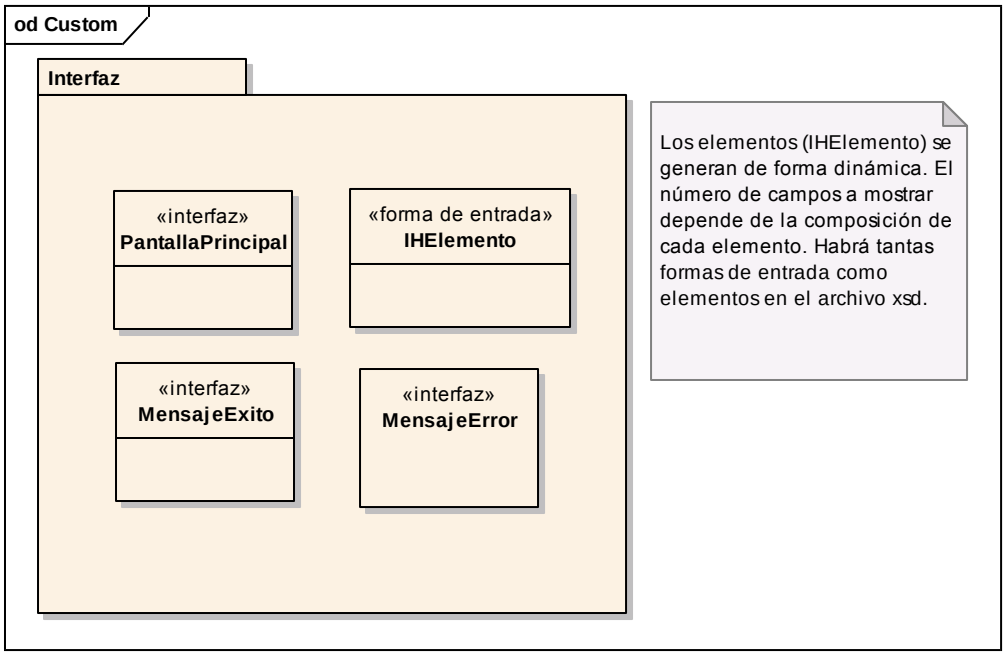


Fig. 3.2.1-1 Diagrama UML del paquete de clases de interfaz.

Clases de control

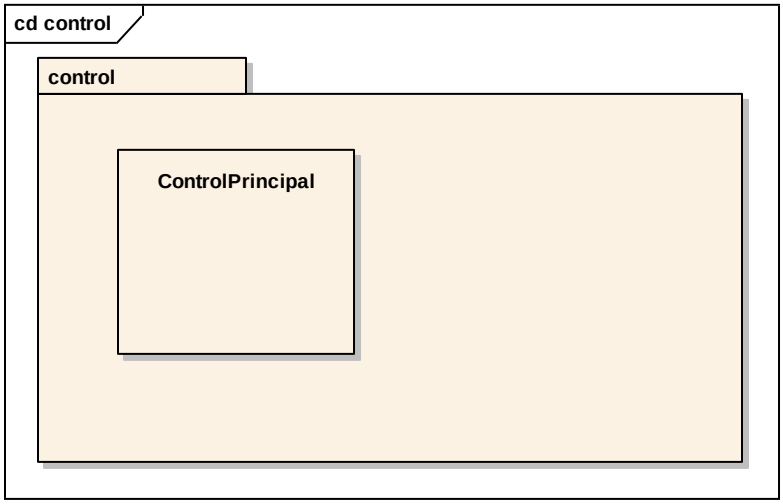


Fig. 3.2.1-2 Diagrama UML del paquete de clases de control.

Clases de entidad

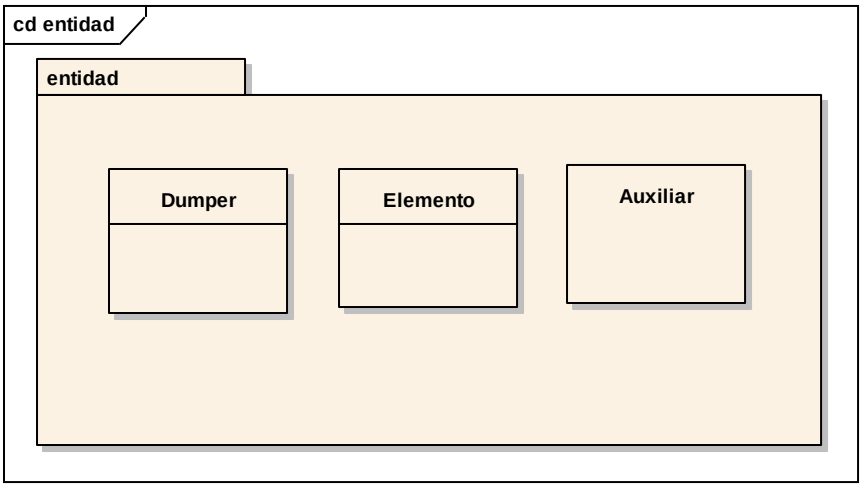


Fig. 3.2.1-3 Diagrama UML del paquete de clases de entidad.

3.2.2 Diagramas de secuencia

Una vez que se identificaron las clases de interfaz, control y entidad el siguiente paso es mostrar cómo se realiza la interacción entre estos tres tipos de elementos.

Los diagramas de secuencia muestran las clases que participan en un caso de uso, el usuario (actor) que puede realizar la petición, la secuencia y los mensajes que se enviarán cada una de las clases.

Sólo se muestran los diagramas de secuencia más relevantes, dado que la interacción entre las clases de los demás casos de uso es similar.

CASO DE USO: Generar Elemento Preconditions

Este caso de uso se presenta al momento de iniciar la aplicación. El sistema inicia la creación de este elemento de tal forma que cuando se carga la pantalla principal el actor puede visualizar en la pantalla principal la forma de entrada que corresponde a este elemento(Figura 3.2.2-1).

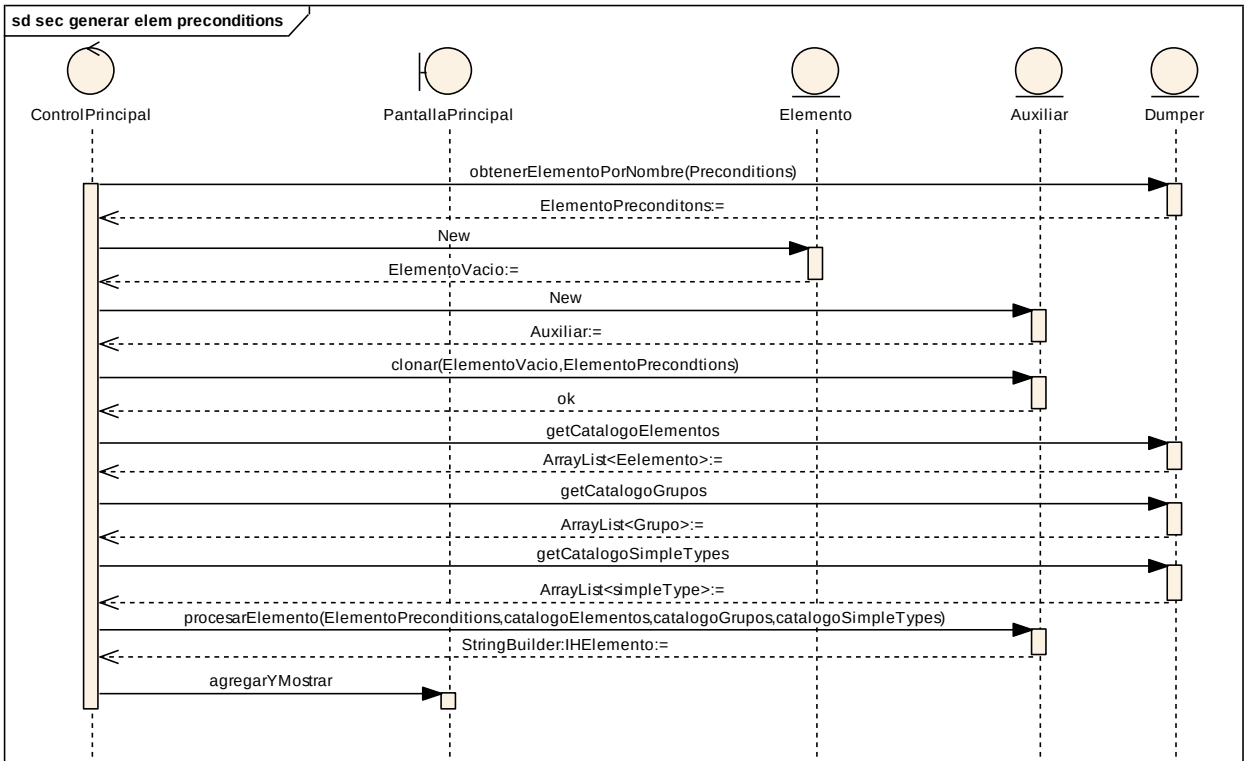


Fig. 3.2.2-1 Diagrama de secuencia del Caso de Uso: Generar Elemento Preconditions.

CASO DE USO: Generar Elemento ChangeSet

Para este caso de uso(Figura 3.2.2-2), el diagrama de secuencia es igual que el anterior, solo que el parámetro que se le pasa ahora es ChangeSet.

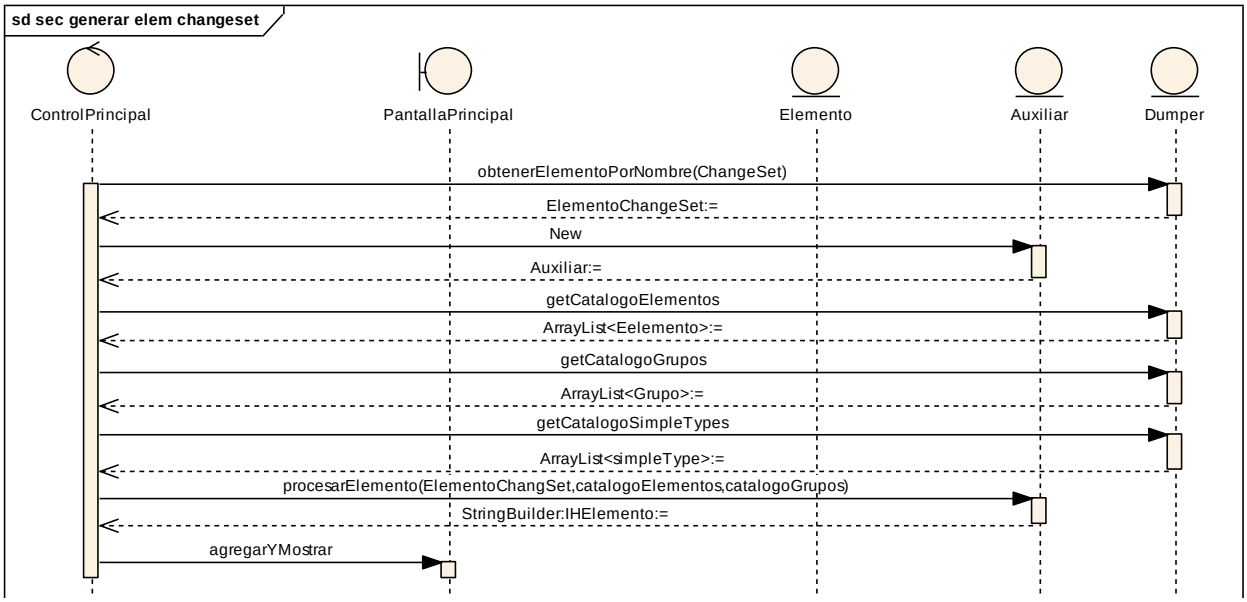


Fig. 3.2.2-2 Diagrama de secuencia del Caso de Uso: Generar Elemento ChangeSet.

Una de las características del sistema o de la aplicación es que permite la creación de formularios de forma dinámica. Cada elemento puede contener o no subelementos. Un elemento puede repetirse dentro del archivo XML. Este evento se desencadena cuando el actor da clic en la imagen con el signo de más. Esta funcionalidad se consigue pasándole un parámetro numérico para controlar la secuencia de elementos creados. Esto se ilustra en el siguiente diagrama de secuencia(Figura 3.2.2-3).

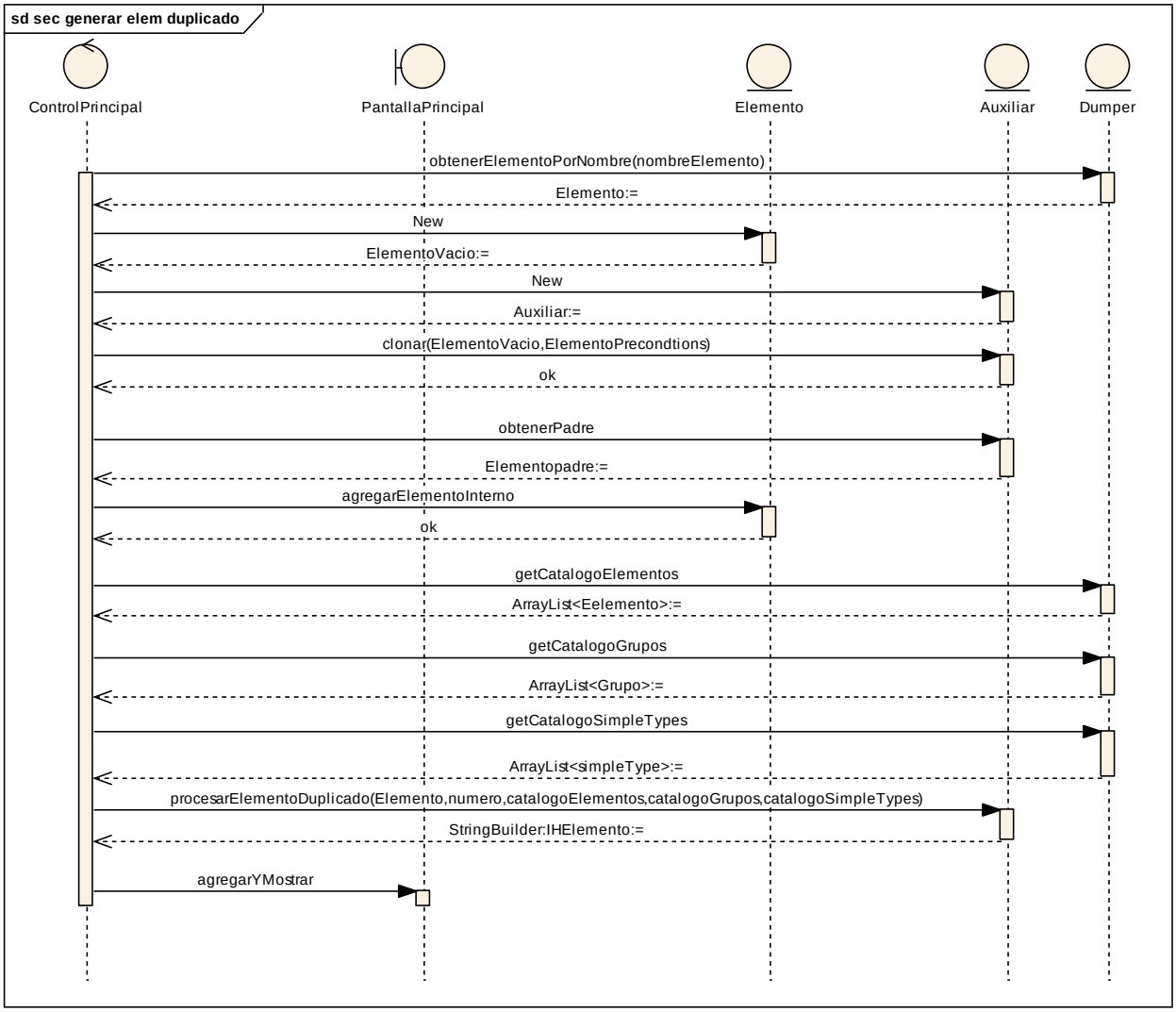


Fig. 3.2.2-3 Diagrama de secuencia que muestra la generación de elementos duplicados.

Diagrama de secuencia para quitar un formulario de elemento de la pantalla principal. Este evento se desencadena cuando el actor da clic en la imagen con el signo de menos(Figura 3.2.2-4).

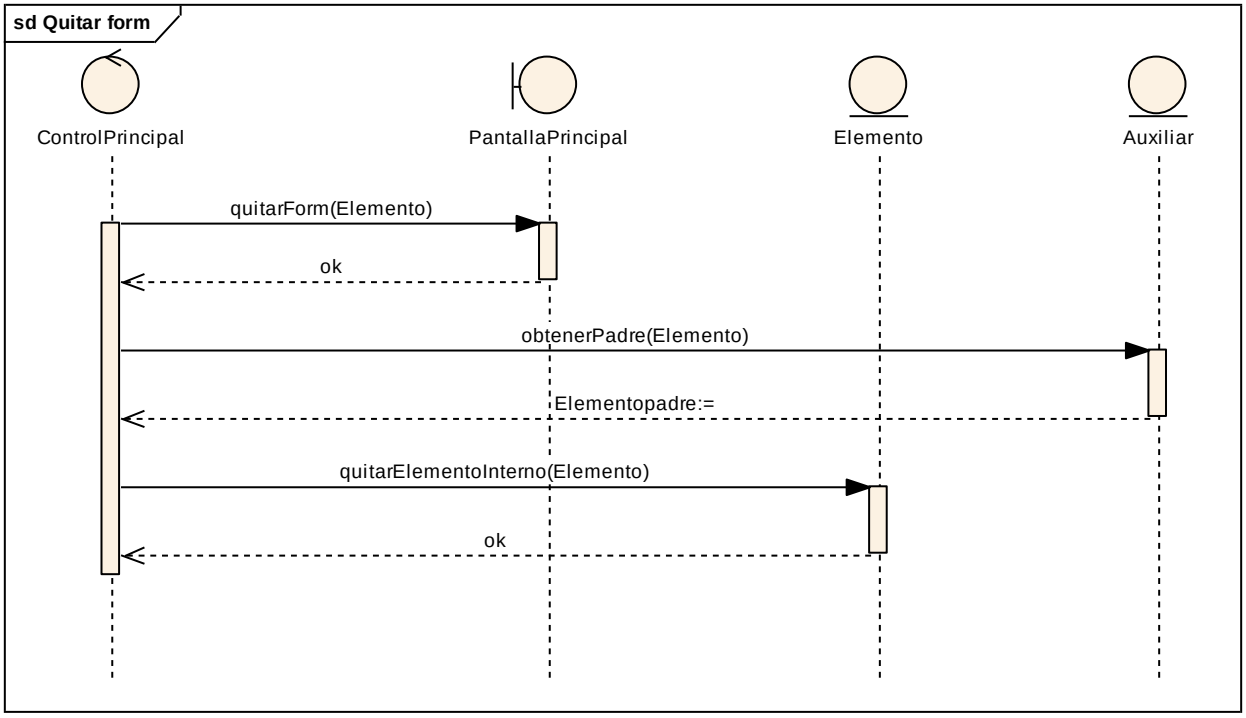


Fig. 3.2.2-4 Diagrama de secuencia para quitar un formulario de la pantalla principal.

Diagrama de secuencia para generar un elemento (refactorización)

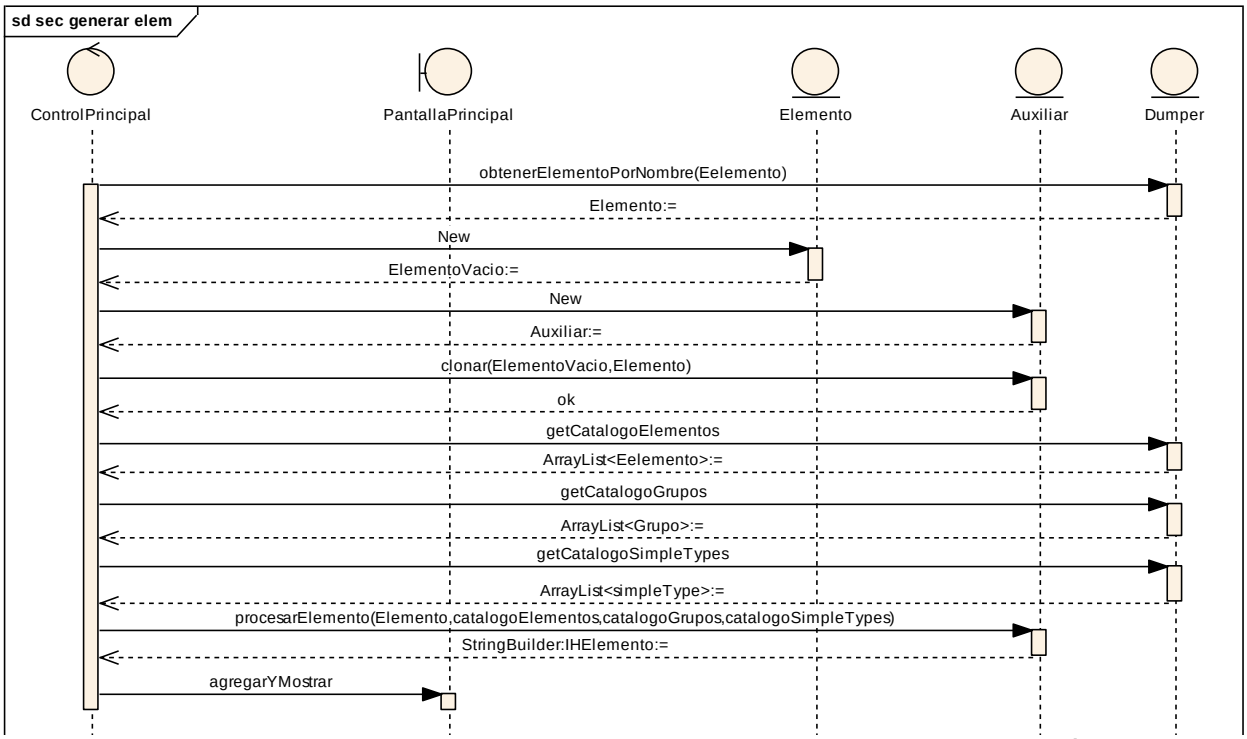


Fig. 3.2.2-5 Diagrama de secuencia para generar un elemento o refactorización.

Diagrama de secuencia para el caso de uso : Create Table

Para este diagrama (Figura 3.2.2-6) se asume que el actor se encuentra en la pantalla principal, el sistema creó el formulario para el elemento preconditions, el elemento ChangeSet y el elemento Create Table.

Los diagramas de secuencia de cada uno de estos elementos ya se mostraron anteriormente.

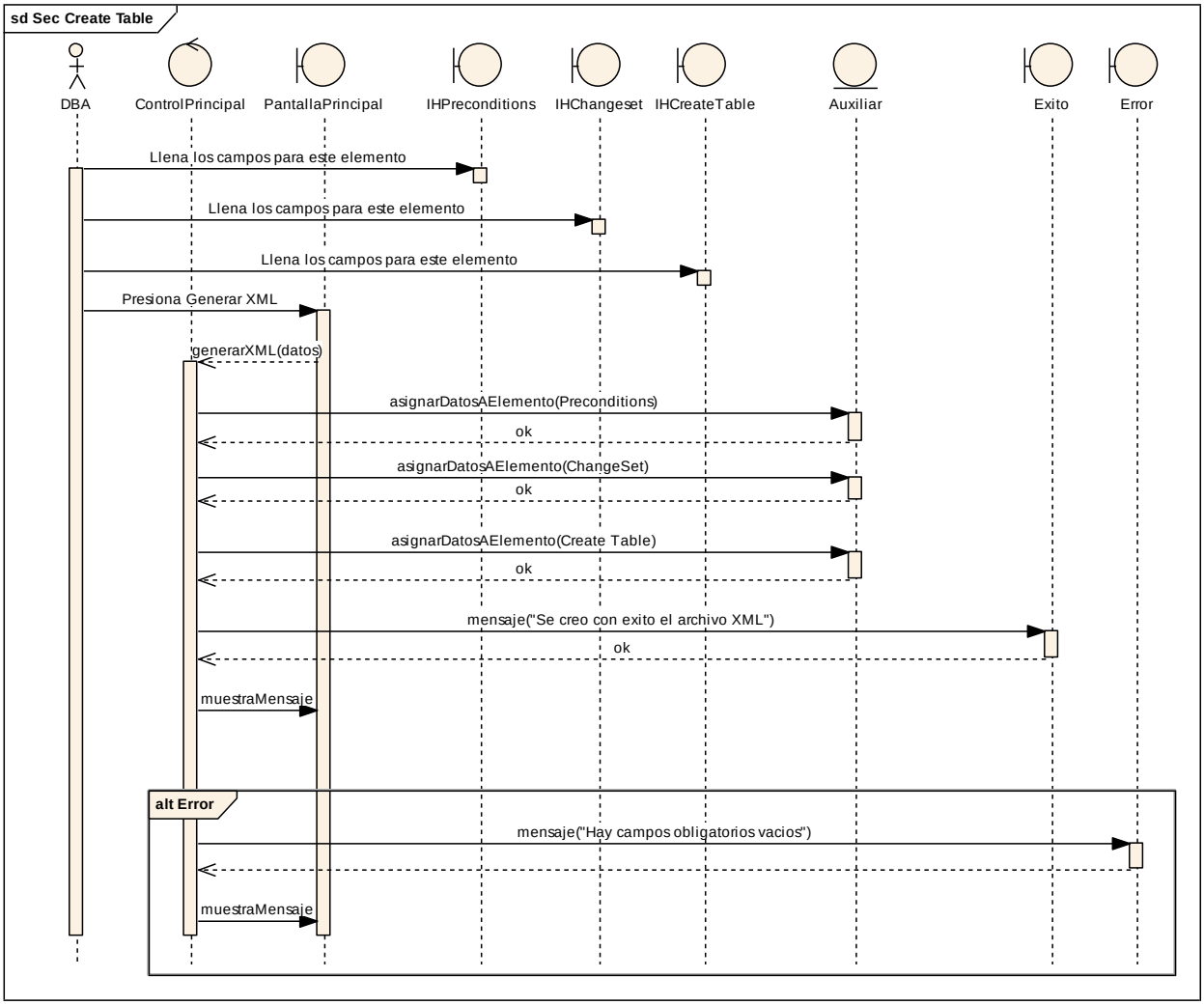


Fig. 3.2.2-6 Diagrama de secuencia para el caso de uso Create Table.

3.3 DISEÑO

El modelo de diseño es un refinamiento y formalización adicional del modelo de análisis, donde se toman en cuenta las consecuencias del ambiente de implementación. El resultado del modelo del diseño son especificaciones muy detalladas de todos los objetos, incluyendo sus operaciones y atributos (Weitzenfeld, 2005).

3.3.1 DESCRIPCIÓN DE LA ARQUITECTURA DEL DISEÑO

ARQUITECTURA DE TRES CAPAS.

Una aplicación Web es un programa informático que puede dar servicio simultáneamente a múltiples usuarios que lo ejecutan a través de Internet. Este tipo de aplicaciones se basa en lo que se conoce como una arquitectura de tres capas (Sierra, 2008), donde los diferentes actores y elementos implicados en la misma se encuentran distribuidos en tres bloques o capas (Fig.3.3.1-1).

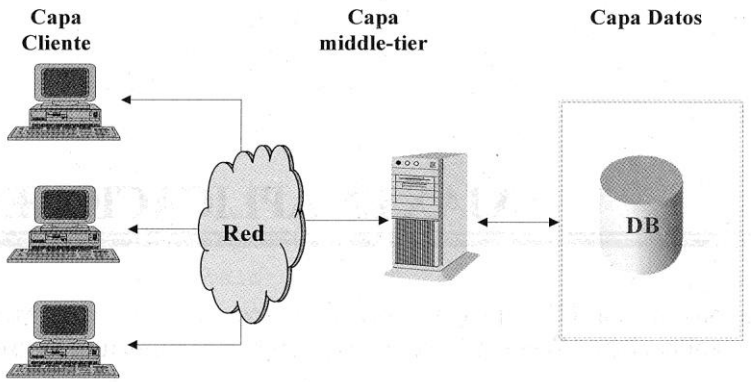


Fig. 3.3.1-1. Arquitectura de tres capas
Fuente: Sierra, A. J. (2008). *Ajax en J2EE.* Alfaomega.

Estas tres capas son:

- **Capa Cliente.**
- **Capa intermedia.**
- **Capa de datos.**

CAPA CLIENTE.

Se trata de la capa con la que interactúa el usuario de la aplicación, normalmente a través de un navegador Web. Realiza principalmente dos funciones: por un lado se encarga de capturar los datos de usuario con los que opera la capa intermedia y enviárselos a ésta, y por otro presentar al usuario los resultados generados por la aplicación.

Las páginas Web, construidas mediante HMTL/CSS, son las encargadas de implementar esta funcionalidad, ayudándose de código JavaScript/AJAX para mejorar la experiencia del usuario con la aplicación.

CAPA INTERMEDIA.

En una arquitectura de tres capas la capa intermedia está constituida por la aplicación en sí. Ésta se encuentra instalada en una máquina independiente, conocida como servidor, a la que acceden los clientes a través de la red utilizando el protocolo HTTP.

La aplicación de la capa intermedia es ejecutada por un motor de la aplicación especial capaz de permitir que una misma instancia de ella pueda dar servicio a múltiples clientes. Además de este motor, los servidores necesitan de otro software conocido como servidor Web, que sirva de interfaz entre la aplicación y el cliente, realizando el diálogo HTTP con éste.

Así pues, de forma resumida podríamos decir que las funciones de la capa intermedia consisten en:

- Recoger los datos enviados desde la capa Cliente.
- Procesar la información y, en general, implementar la lógica de aplicación, incluyendo el acceso a los datos.
- Generar las respuestas para el cliente.

Es precisamente en el desarrollo de esta capa intermedia donde J2EE encuentra su aplicabilidad, pues las distintas bibliotecas que la componen están orientadas a realizar estas funcionalidades.

CAPA DE DATOS.

La capa de datos tiene como misión el almacenamiento permanente de la información manejada por la aplicación y la gestión de la seguridad de los mismos.

Para esta tarea se utilizan, en la mayoría de los casos, las llamadas bases de datos relacionales. Una base de datos relacional distribuye la información entre diferentes tablas, relacionándolas entre sí a través de un campo común que permita identificar los registros de una tabla que se corresponden con los de la otra.

En la Figura 3.3.1-2 aparecen algunos de los tipos de bases de datos relacionales más utilizados actualmente:

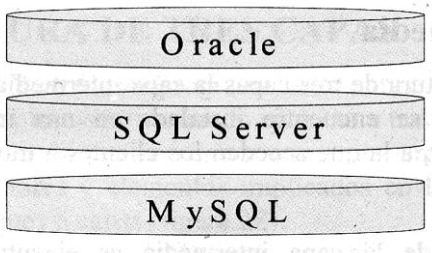


Fig. 3.3.1-2. Tipos de bases de datos relacionales
Fuente: Sierra, A. J. (2008). *Ajax en J2EE.* Alfaomega.

Cada una de las tecnologías de desarrollo de capa intermedia proporciona sus propias bibliotecas para poder acceder a una base de datos desde la aplicación. En el caso de J2EE esta biblioteca es JDBC.

ARQUITECTURA MODELO VISTA CONTROLADOR.

De cara a afrontar con éxito el desarrollo de la capa intermedia de una aplicación Web, se hace necesario establecer un modelo ó esquema que permita estructurar esta capa en una serie de bloques o componentes, de modo que cada uno de estos bloques tenga unas funciones definidas dentro de la aplicación y pueda desarrollarse de manera independiente.

Uno de estos esquemas y, con toda seguridad, el más utilizado por los desarrolladores de aplicaciones J2EE, es la arquitectura Modelo Vista Controlador (MVC), la cual proporciona una clara separación entre las distintas responsabilidades de la aplicación como se observa en la Figura 3.3.1-3:

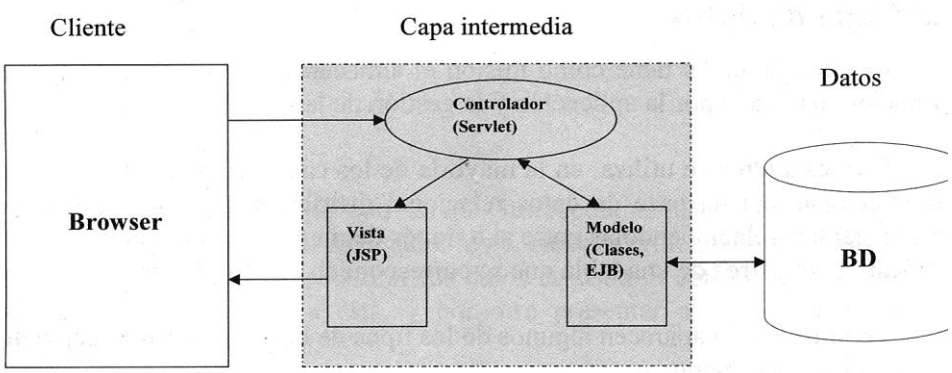


Fig.3.3.1-3. Arquitectura MVC
Fuente: Sierra, A. J. (2008). *Ajax en J2EE.* Alfaomega.

Según esta arquitectura, la capa intermedia de una aplicación Web puede dividirse entre grandes bloques funcionales:

- Controlador
- Vista
- Modelo

EL CONTROLADOR. Se puede decir que el controlador es el "cerebro" de la aplicación. Todas las peticiones a la capa intermedia que se realicen desde el cliente son dirigidas al controlador, cuya misión es determinar las acciones a realizar para cada una de las peticiones e invocar el resto de los componentes de la aplicación (Modelo y Vista) para que realicen las acciones requeridas en cada caso, encargándose también de la coordinación de todo el proceso.

Por ejemplo, en el caso de que una petición requiera enviar determinada información existente en una base de datos como respuesta al cliente, el controlador solicitará los datos necesarios al modelo y, una vez recibidos, se los proporcionará a la vista para que ésta les aplique el formato de presentación correspondiente y envíe la respuesta al cliente.

En aplicaciones J2EE el controlador es implementado mediante un servlet central que, dependiendo de la cantidad de tipos de peticiones que debe gestionar, puede apoyarse en otros servlets auxiliares para procesar cada petición.

LA VISTA. Tal y como se puede deducir de su nombre, la vista es la encargada de generar las repuestas (habitualmente XHTML) que deben ser enviadas al cliente. Cuando esta repuesta tiene que incluir datos proporcionados por el controlador, el código XHTML de la página no será fijo, sino que deberá ser generado de forma dinámica, por lo que su implementación correrá a cargo de una página JSP. Cuando la información que se va a enviar es estática, es decir, no depende de datos extraídos de un almacenamiento externo, podrá ser implementada por una página o documento XHTML.

Es precisamente en la vista donde AJAX juega un papel importante. Tanto las páginas JSP como las XHTML pueden incluir código script de cliente que se comunique en segundo plano con el servidor para obtener datos de él.

En este caso, una vez que la vista ha generado la respuesta y la ha enviado al cliente, las peticiones realizadas al servidor desde el código AJAX serán dirigidas al controlador, de este modo, cuando el servlet recibe la petición desde una aplicación AJAX cliente, recuperará los datos necesarios a través del modelo y, en vez de encaminar la petición a la vista para que vuelva a generar la respuesta, lo que supondría una recarga de la página, aplicará a los datos el formato correspondiente (texto plano, XML, JSON…) y los enviará directamente a la página cliente que hizo la petición. Esto se observa en la Figura 3.3.1-4.

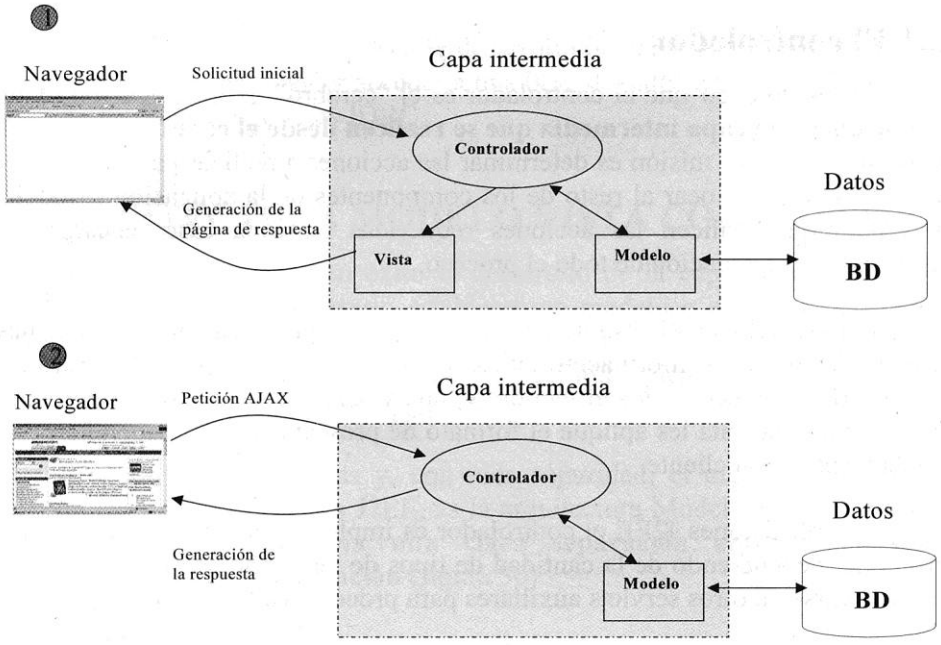


Fig.3.3.1-4. Ajax y MVC
Fuente: Sierra, A. J. (2008). *Ajax en J2EE.* Alfaomega.

EL MODELO.

En la arquitectura MVC la lógica de negocio de la aplicación, incluyendo el acceso a los datos y a su manipulación, está encapsulada dentro del modelo. En una aplicación J2EE el modelo puede ser implementado mediante clases estándar Java o a través de Enterprise JavaBeans.

La aplicación para desarrollar los archivos XML (Refactorizaciones) de entrada de Liquibase, se construirá aplicando AJAX junto con el modelo vista controlador. La plataforma de desarrollo elegida es Java, de Sun Microsystems, el cual es un lenguaje de programación sumamente robusto, no sólo para aplicaciones de escritorio sino para aplicaciones de Internet. El IDE de desarrollo NetBeans y el Servidor de Aplicaciones Apache Tomcat. La persistencia de los datos, dada la naturaleza de la aplicación, se manejará en memoria por lo que no será necesario utilizar ningún manejador de bases de datos.

3.3.2 DIAGRAMA DE CLASES DE DISEÑO DETALLADO

Es el diagrama que muestra un conjunto de clases con sus atributos y métodos.

Clases de Interfaz

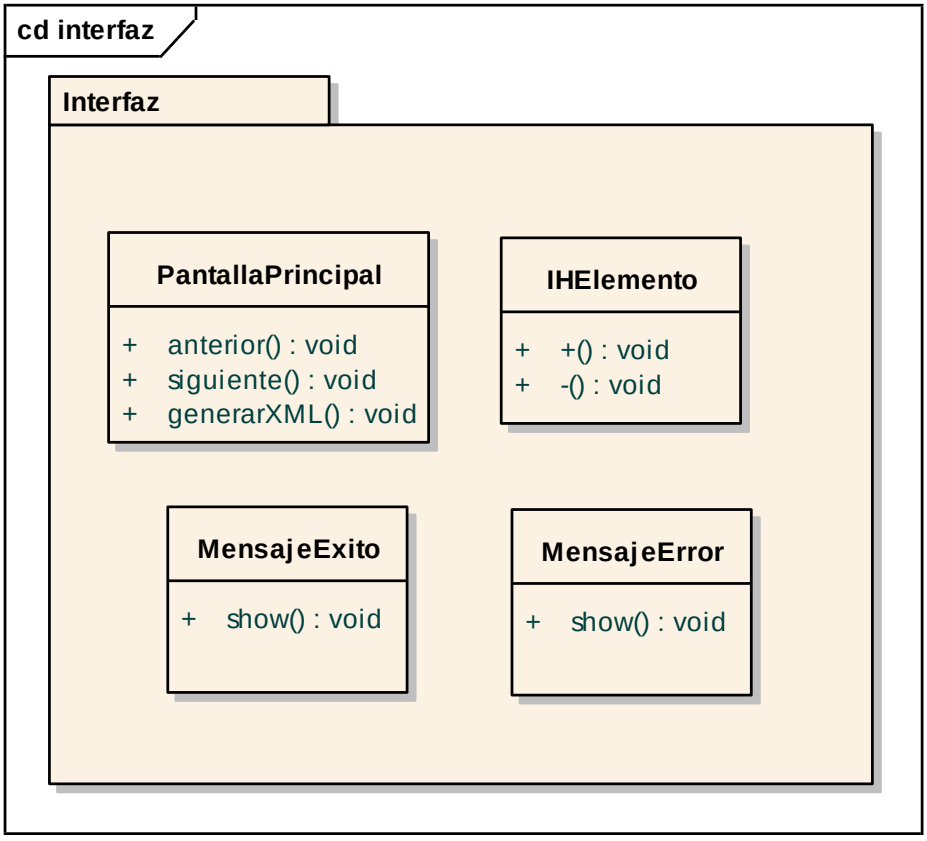


Fig. 3.3.2-1 Diagrama UML del paquete detallado de clases de entidad.

Clases de Control

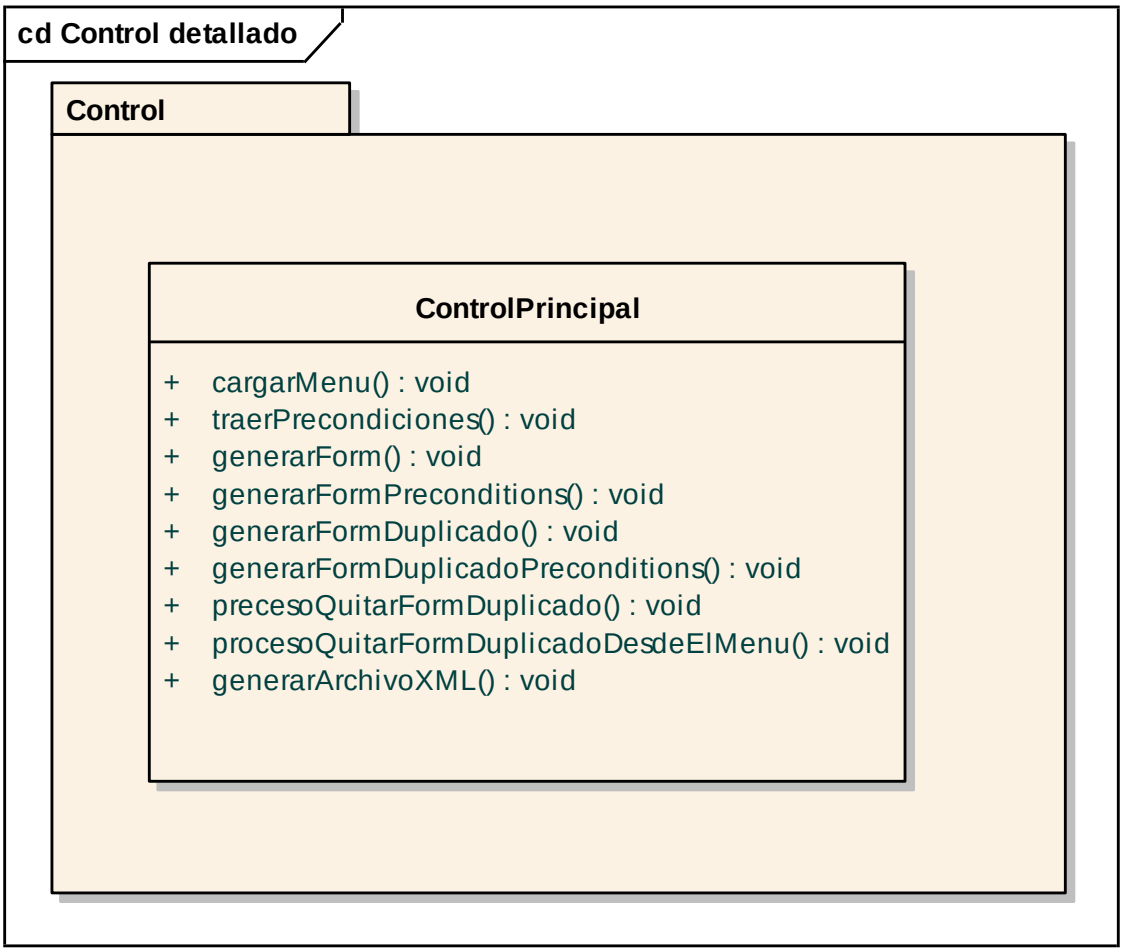


Fig. 3.3.2-2 Diagrama UML del paquete detallado de clases de control.

Clases Entidad

Fig. 3.3.2-3 Diagrama UML del paquete detallado de clases de entidad.

3.4 IMPLEMENTACION Y PRUEBAS

El modelo de implementación toma el resultado del modelo de diseño para generar el código final. Esta traducción debe ser relativamente sencilla y directa, ya que las decisiones mayores han sido tomadas durante las etapas previas. Durante el modelo de implementación se adapta al lenguaje de programación y/o base de datos, según la especificación del diseño y las propiedades del lenguaje de implementación y base de datos (Weitzenfeld, 2005).

3.4.1 Código de clases

En esta sección el número encerrado entre paréntesis es una notación empleada para relacionar el código mostrado en las figuras con su descripción correspondiente.

A continuación se mostrará parte del código empleado para el caso de uso Generar XML descrito en la sección 3.1.4.

Este caso de uso se inicia cuando el actor presiona el botón etiquetado con el texto "Generar XML" localizado en la pantalla principal del sistema. Esta acción envía una llamada a una función JavaScript llamada *generarArchivoXML(1)* localizada en la página principal del programa. Dentro de esta función se crea un objeto *XMLHttpRequest(3)* para poder utilizar las funciones de Ajax. También se le hace una llamada a la función JavaScript *obtenerDatos(2)* con la cual se genera una cadena con los nombres y valores de cada uno de los campos de los formularios. Una vez que ya se tiene el objeto y los datos, se hace una llamada al controlador(*4*) y se le pasa como parámetro la operación a realizar. El método *onreadystatechange* del objeto Ajax nos indica que una vez que se haya terminado el proceso, deberá regresar a la función *cargarGenerarXML(5)* para su proceso final. En la Figura 3.4.1-1 se muestra el código utilizado.

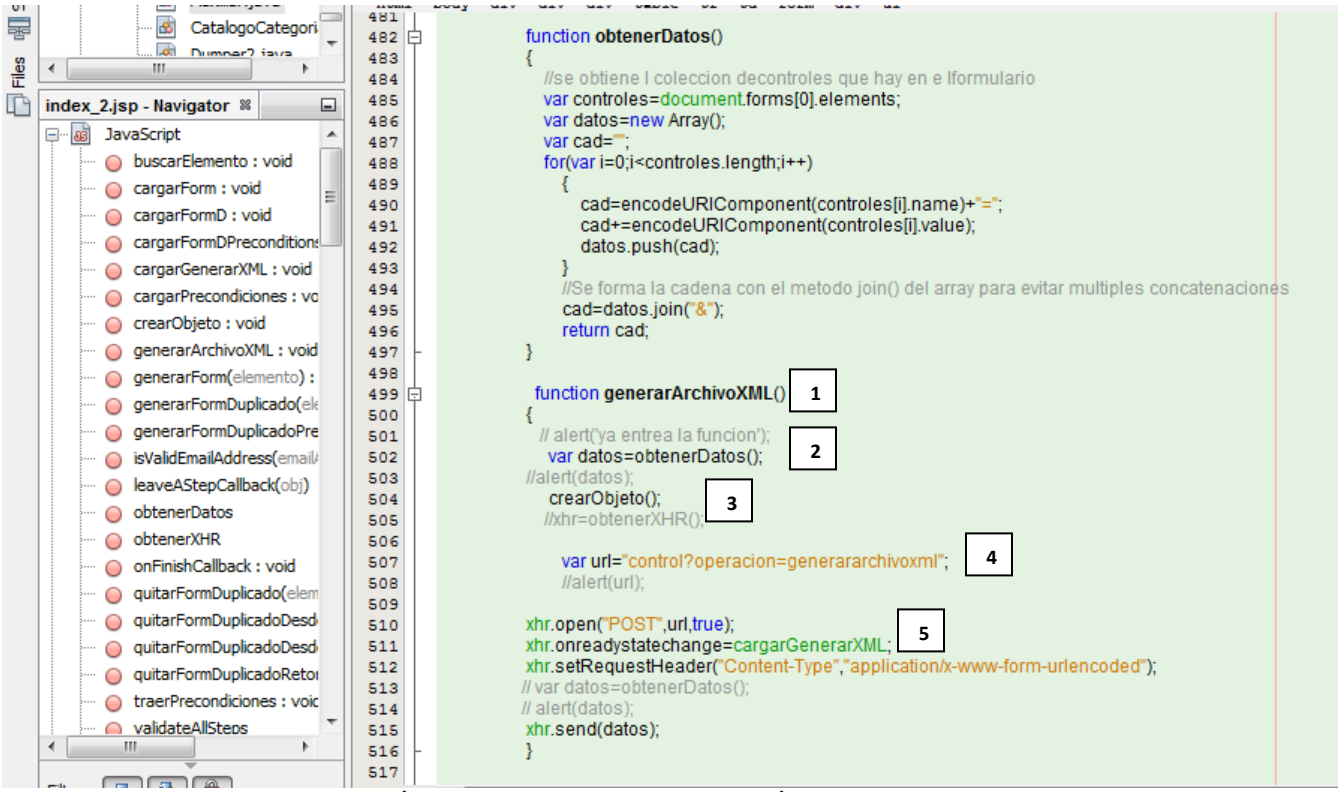


Fig.3.4.1-1 Código utilizado para la generación del archivo XML.

Dentro del controlador, lo primero que se hace es obtener de la sesión los elementos creados(*6*). Es necesario usar el recurso sesión de Java porque HTML no tiene estados, por lo que sin la ayuda de este mecanismo no sería posible recuperar los elementos (refactorizaciones) con los valores asignados. Los elementos obtenidos de la sesión son : elementoPreconditions, el elemento ChangeSet y el elemento refactorización. El arreglo *arrayconcamposobligatorios* es un atributo de la clase Auxiliar en el cual se van almacenando los atributos que son obligatorios para algunos elementos utilizados en la refactorización a realizar.

En el HashTable *datos(7)* se guardarán los datos provenientes de los campos de los formularios, utilizando para ello un formato de campo = valor. Se utilizó HashTable debido a que es una estructura que permite guardar un objeto y asignarle una llave única para poder recuperarlo. Para este caso la llave será el nombre del campo del formulario y su valor será el valor asignado para este campo en el formulario.

Una vez que están listos estos elementos(el HastTable y el ArrayList), se utilizará un método de la clase Auxiliar llamado "*hayCamposObligatoriosvacios*" para saber si existen atributos obligatorios que no se les haya asignado algún valor. Dependiendo del resultado de este método se enviará un mensaje a la función de retorno de llamada del objeto Ajax.

Si hay campos obligatorios vacíos se enviará el mensaje false y se saldrá de la operación *generararchivoxml* perteneciente al controlador(*8*). En caso contrario se continuará con la generación del archivo XML.

Por medio del método *asignarDatosAElemento(9)* de la clase Auxiliar, se asignará valor a cada uno de los campos que forman al elemento, haciendo uso de la información guardada en la variable *datos* de tipo HashTable. En la Figura 3.4.1-2 se muestra la primera parte del código utilizado para la generación del archivo XML[4].

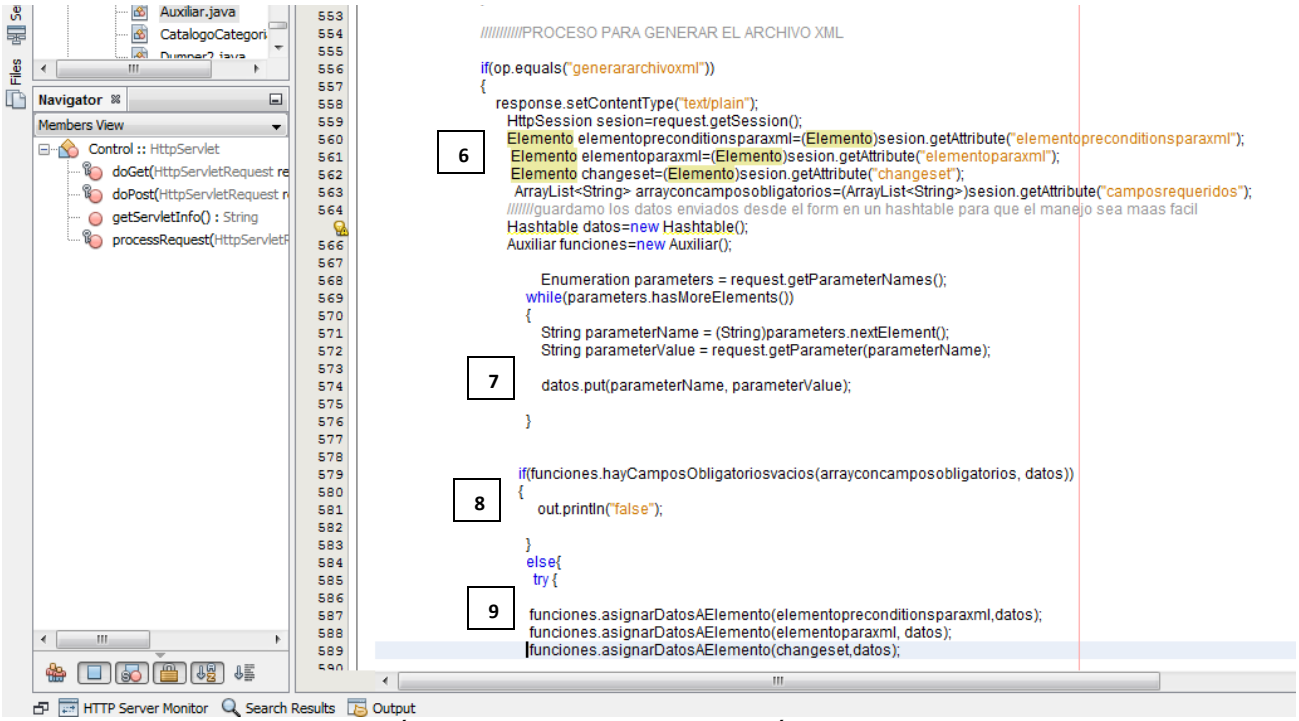


Fig.3.4.1-2 Código utilizado para la generación del archivo XML.

Para utilizar JDOM, el primer paso es crear un elemento raíz y asignarle los espacios de nombre necesarios(*10*). Después por medio de método *procesarElementoParaArchivoXML (11)* de la clase Auxiliar se obtendrán objetos Element definidos en JDOM. Cada uno de estos objetos se agregará al documento XML por medio del método addContent(*12*), de JDOM, dependiendo de la estructura jerárquica de cada elemento.

Finalmente por medio del método output del objeto XMLOutputter(*13*) se creará el archivo XML requerido con nombre file y se guardará en la ruta C:\\users\Owner\Documents.

En la figura 3.4.1-3 se muestra el código.

[4] En el proceso para generar el archivo XML se utilizará JDOM.

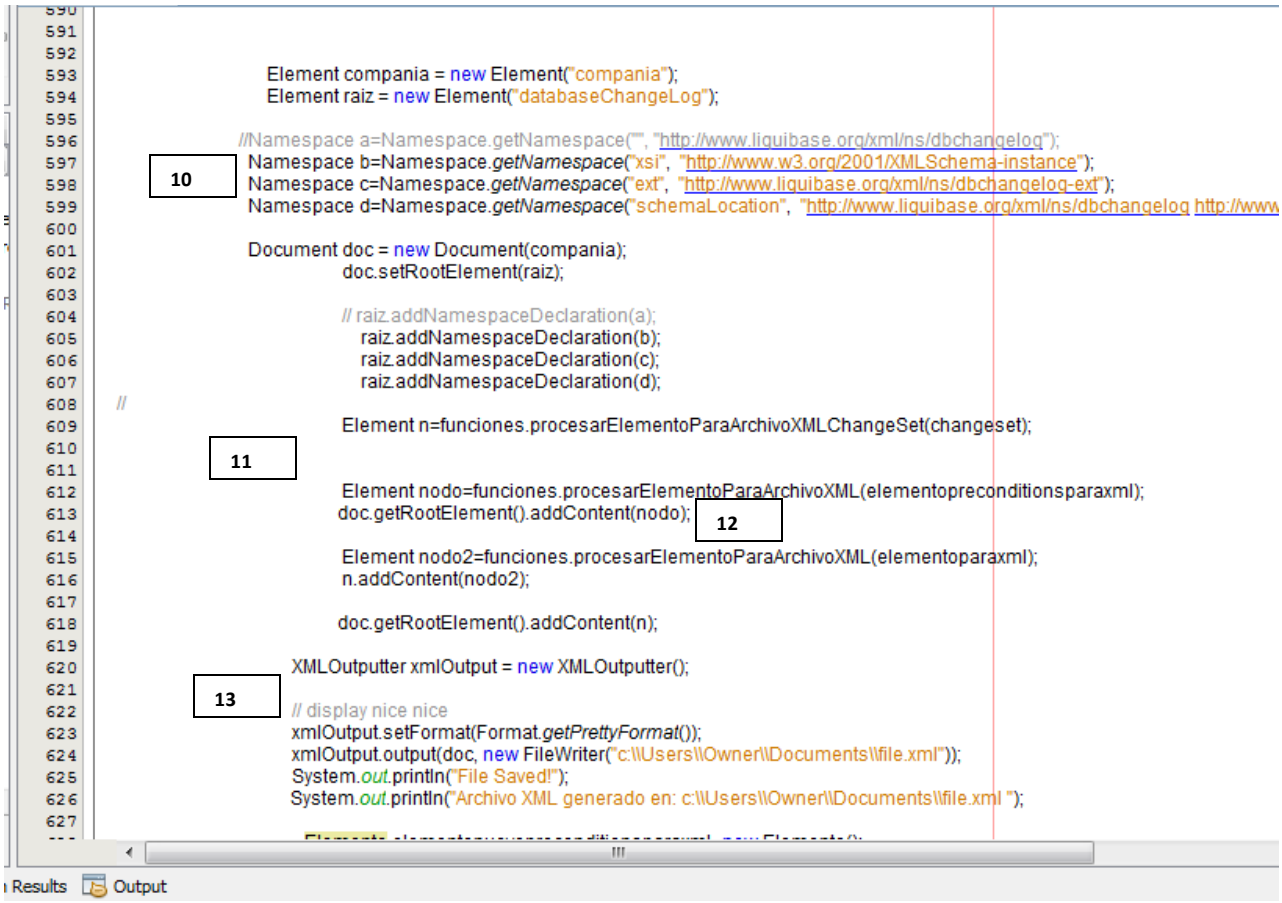


Fig.3.4.1-3 Código utilizado para la generación del archivo XML.

Para finalizar con la creación del archivo XML, se quitan de sesión los elementos utilizados por medio del método removeAttributte(*14*) y se vuelven a poner en sesión elementos nuevos, por medio del método setAttribute(*15*), para poder iniciar el proceso de creación de refactorizaciones(Figura 3.4.1-4).

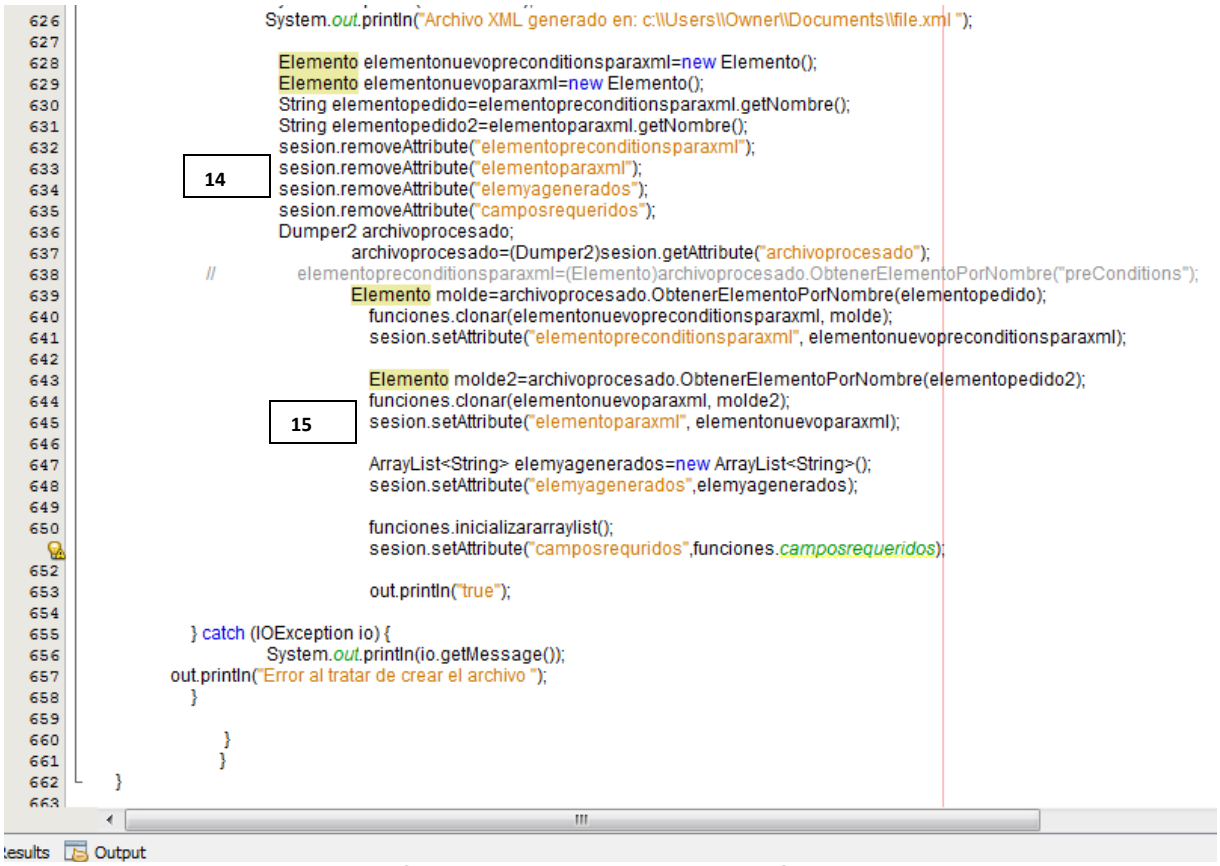


Fig.3.4.1-4 Código utilizado para la generación del archivo XML.

De regreso al objeto Ajax, se obtiene la respuesta enviada desde el control. Si la respuesta es "false" se mostrará una ventana con el mensaje "Hay campos obligatorios vacíos"(*16*). En caso contrario, se mostrará una ventana con el mensaje" Se generó el archivo XML"(*17*); además de que el contenido del div donde se encuentra el formulario de la refactorización se limpiará y también se invocará la llamada a una función JavaScript llamada *traerPrecondiciones*, la cual se encarga de dibujar el formulario para el elemento Preconditions y de esta manera iniciar nuevamente el proceso de la generación del archivo XML(Figura 3.4.1-5).

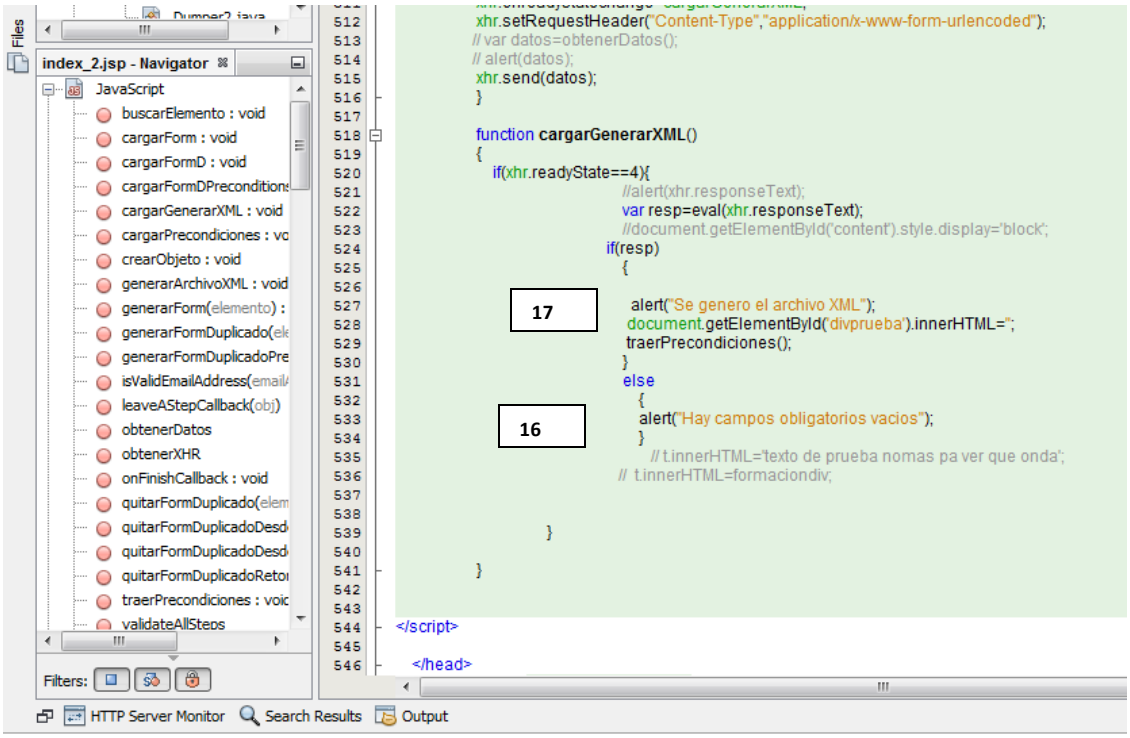


Fig.3.4.1-5 Código utilizado para la generación del archivo XML.

En esta parte del código se trata de ilustrar cómo es el proceso de funcionamiento de la aplicación por medio de Ajax, además de la interacción entre las clases que la componen.

Un proceso que considero de gran importancia para el desarrollo de esta aplicación es el que se aplica en la clase Dumper. Esta clase es la encargada de leer y analizar la estructura del esquema XML por medio de la biblioteca XSOM(XML Schema Object Model). Una vez que se carga el archivo con el esquema, esta clase hace uso de funciones recursivas para poder formar los elementos que componen el esquema (Figura 3.4.1-6).

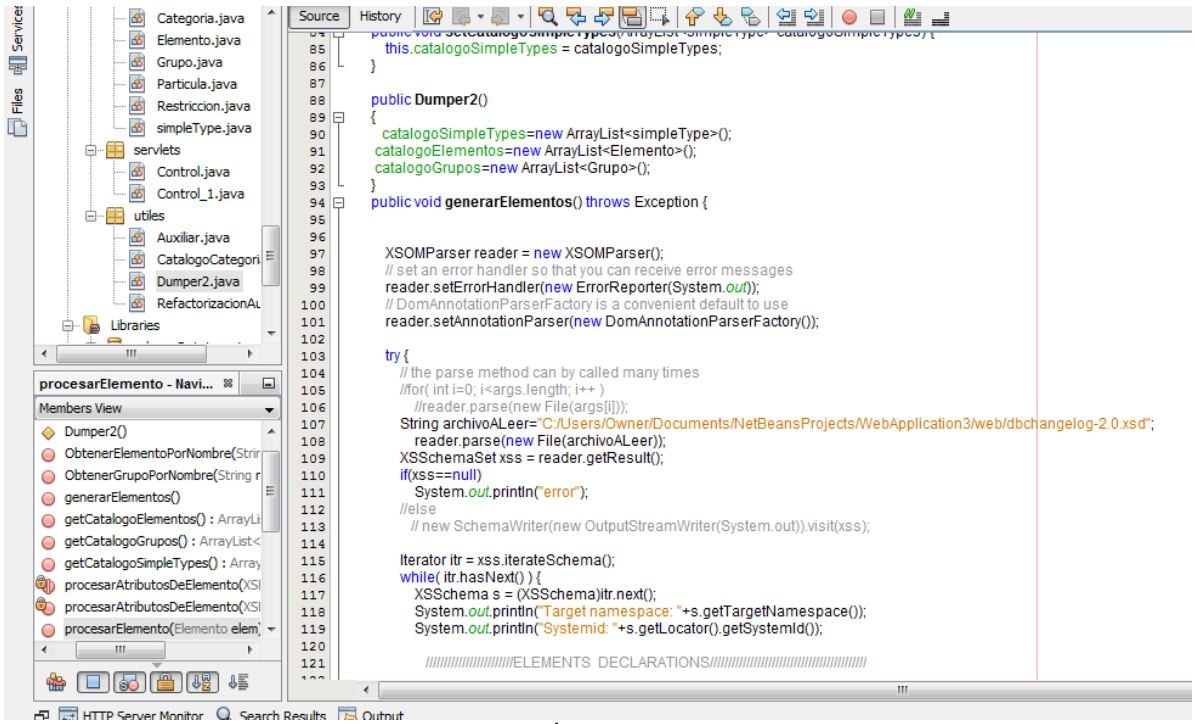


Fig.3.4.1-6 Parte del código utilizado en la clase Dumper.

3.4.2 Reporte del plan de pruebas del sistema

Probar un producto es relativamente independiente de la metodología de desarrollo utilizada para construirlo.
El modelo de pruebas debe ser planificado con anticipación y de manera integral junto con el desarrollo del sistema. Es un error pensar que las pruebas son la última actividad del desarrollo, ya que no se puede lograr software de alta calidad sólo mediante pruebas finales y depuraciones (Weitzenfeld, 2005). Además, las pruebas finales deben tener como objetivo la certificación final de la calidad del producto y no la búsqueda de errores.
Las pruebas son un flujo fundamental cuyo propósito esencial es comprobar el resultado de la implementación mediante las pruebas de cada construcción.

A continuación se muestra el reporte de pruebas del sistema, del caso de uso Drop Table. El cual se puede consultar en la sección 3.1.4.

Entradas	Resultados esperados	Resultados obtenidos
	El sistema carga en la pantalla principal el formulario con los campos a llenar para el elemento precondiciones	El sistema carga en la pantalla principal el formulario con los campos del elemento precondiciones
Seleccionar algún elemento de la lista desplegable	El sistema carga o muestra en la pantalla principal el formulario con los campos a llenar para el elemento seleccionado	El sistema carga o muestra en la pantalla principal el formulario con los campos a llenar para el elemento seleccionado
Dar clic en la imagen con el signo (+)	El sistema carga en la pantalla principal otro elemento idéntico al que tiene la imagen con el signo (+)	El sistema carga en la pantalla principal otro elemento idéntico al que tiene la imagen con el signo (+)
Dar clic en la imagen con el signo (-)	El sistema quita de la pantalla principal el formulario correspondiente a este elemento.	El sistema quita de la pantalla principal el formulario correspondiente a este elemento.
Dar clic en el botón siguiente	El sistema muestra la pantalla correspondiente a la refactorización	El sistema muestra la pantalla correspondiente a la refactorización
Dar clic en el botón anterior	El sistema muestra la pantalla con el formulario del elemento precondiciones	El sistema muestra la pantalla con el formulario del elemento precondiciones
Dar clic en alguna categoría del menú	El sistema despliega un menú con subcategorías	El sistema despliega un menú con subcategorías
Dar clic en alguna subcategoría (refactorización)	El sistema despliega en la pantalla que corresponde a la refactorización el formulario con los campos a llenar para este elemento.	El sistema despliega en la pantalla que corresponde a la refactorización el formulario con los campos a llenar para este elemento.
Dar clic en el botón generar XML y todos los campos requeridos están llenos	El sistema muestra un mensaje de éxito.	El sistema muestra un mensaje de éxito.
Dar clic en el botón generar	El sistema muestra un	El sistema muestra un

XML y no todos los campos requeridos están llenos	mensaje avisando que faltan por llenar campos requeridos.	mensaje avisando que faltan por llenar campos requeridos.

Los demás casos de uso tienen el mismo comportamiento y se pueden probar de la misma manera que este caso de uso.

CAPITULO 4: **Caso práctico: Refactorizando una base de datos en un ambiente de una sola aplicación.**

4.1 Contexto de la aplicación

La aplicación web Sistema de Consulta a Documentos de Envase y Embalaje fué desarrollada con el lenguaje de programación PHP y MySQL como manejador de bases de datos. El objetivo de esta aplicación es ayudar a los miembros de cierto instituto a realizar búsquedas puntuales sobre la base de datos de libros con los que cuenta, para que una vez que el libro sea localizado, éste se pueda pedir para préstamo o para consulta.

En la primera pantalla se pueden elegir los criterios de búsqueda, como son:

- El texto a buscar.
- Una categoría dentro de las cuales se encuentran: autor, título y tema.
- El tipo de material: libro, tesis, carpeta o encuadernado.

Una vez que se han introducido valores en estos campos y se presiona el botón etiquetado con la leyenda "BUSCAR", el sistema nos presenta una pantalla con el resultado de la búsqueda como se muestra en la Figura 4.1-1. En esta pantalla se presenta un listado compuesto de los elementos: clasificación, autor y título de los recursos que cumplen con los criterios de búsqueda.

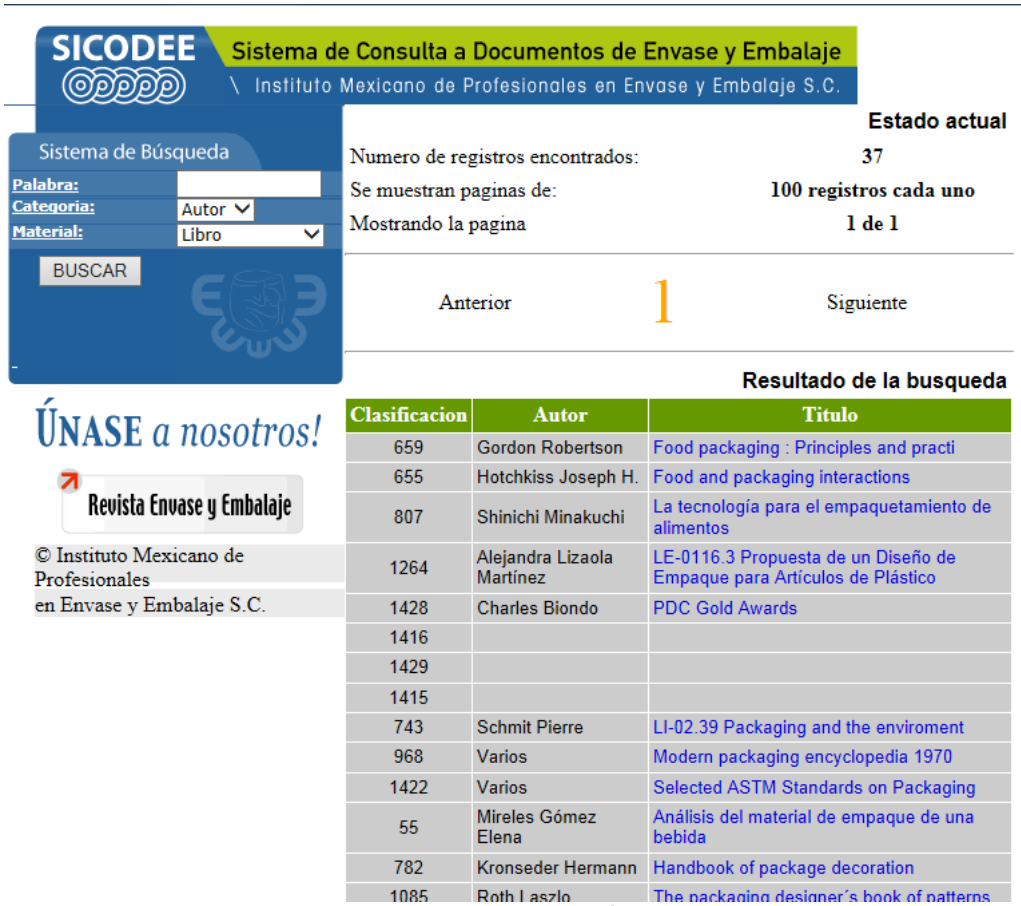


Fig. 4.1-1 Resultados de la búsqueda en el sistema

Posteriormente, el usuario puede dar clic en el título del recurso que sea de su interés para obtener más detalles, como son: Editorial, año de impresión, idioma y páginas, como se muestra en la Figura 4.1-2.

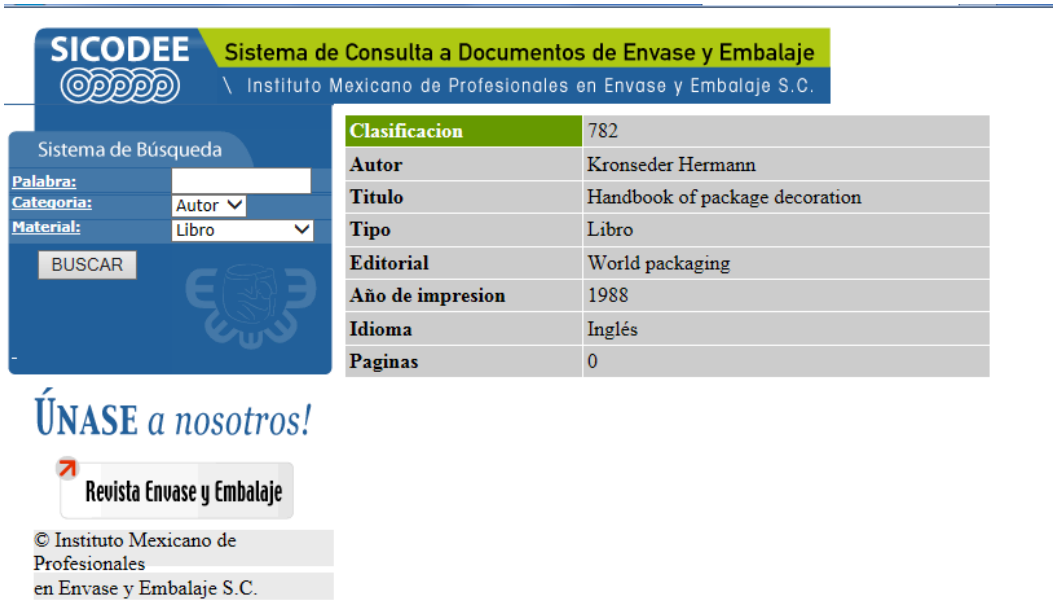


Fig. 4.1-2. Pantalla del sistema con detalle del libro.

4.2 Esquema de la base de datos a refactorizar

En la Figura 4.2-1 se muestra el esquema de la base de datos empleada en la aplicación web mencionada en el subtema anterior (4.1). Como se puede observar, se trata de una sola tabla con un índice único, por lo que su diseño mejorará notablemente al aplicarle el proceso de refactorización.

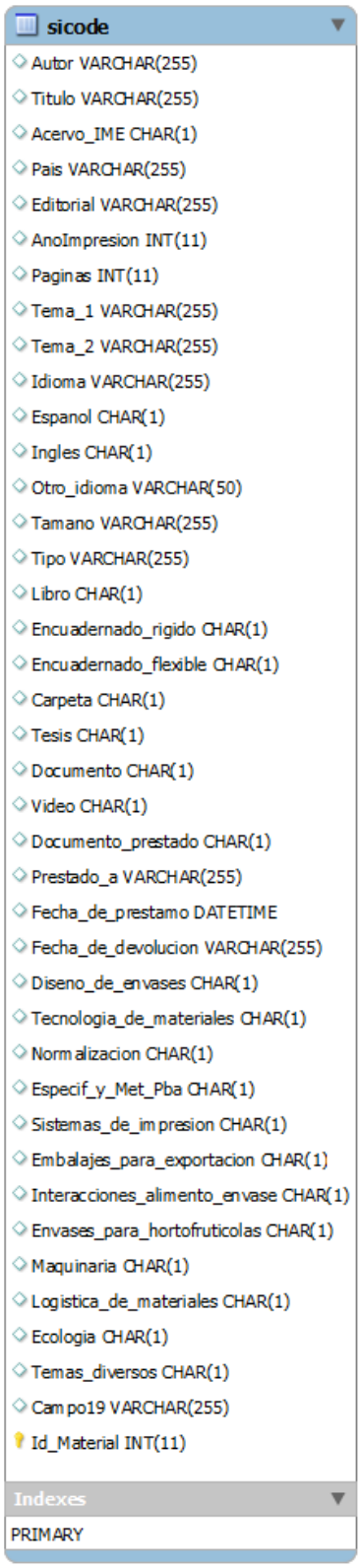


Fig. 4.2-1 Tabla de la base de datos a refactorizar

4.3 Problemas que presenta el esquema

El problema principal que presenta el esquema de base de datos de la aplicación es que no se encuentra normalizado, lo que nos lleva a tener que aplicar una serie de pasos sobre el diseño de las tablas, de manera que se pueda evitar lo siguiente:

- Las redundancias de información
- Las inconsistencias por eliminación de registros
- La pérdida de dependencias funcionales por descomposición
- La creación de relaciones o dependencias funcionales inexistentes, y
- La ineficiencia en el mantenimiento y el acceso a las relaciones.

Aunado al hecho de que el esquema no está normalizado se derivan otros inconvenientes, como por ejemplo:

- No tiene suficientes índices para que las búsquedas sean más rápidas.
- No tiene llaves foráneas para mantener la integridad referencial.
- Existencia de campos que no se utilizan.
- Valores diferentes para el mismo atributo.

En cuanto a la aplicación, los problemas que presenta son los siguientes:

- El acceso a la base de datos no se ha encapsulado, por lo que es más difícil actualizar las sentencias SQL que tienen que ver con la lógica del negocio.
- No cuenta con procedimientos almacenados, los cuales ayudarían en el rendimiento y evitarían duplicación de sentencias SQL.

4.4 Refactorizaciones a aplicar

Dentro de las refactorizaciones a aplicar se encuentran:

Eliminación de campos que no se usan
Creación de tablas
Creación de columnas.
Creación de llaves primarias
Creación de llaves foráneas
Creación de índices
Creación de procedimientos almacenados
Creación de valores por default
Creación de valores nulos
Inserción de datos
Respaldo de datos

La elección o el uso de estas refactorizaciones tiene que ver principalmente con el proceso de normalización que hay que llevar a cabo para conseguir un mejor diseño de la base de datos. También tiene que ver con el concepto de integridad referencial entre las tablas creadas y también con el concepto de eficiencia y rendimiento que se necesita en las consultas y en el acceso a datos.

4.5 Resultados después de aplicar las refactorizaciones

Aplicando el proceso de refactorización al esquema de bases de datos.

El proceso a seguir es el siguiente:

PASO 1.- Generar el archivo changeLog para la refactorización, por medio de la aplicación desarrollada. En la Figura 4.5-1 se muestra la pantalla principal de la aplicación.

Fig.4.5-1. Pantalla principal de la aplicación para generar refactorizaciones en formato XML.

El resultado es la generación del archivo XML que contiene los elementos necesarios para que el programa Liquibase efectúe las sentencias SQL sobre la base de datos para conseguir este cambio.

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd
http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd">
  <preConditions onFail="HALT" onError="HALT" onSqlOutput="TEST">
    <dbms type="mysql" />
    <runningAs username="root" />
  </preConditions>
  <changeSet id="1" author="roman">
    <sql>
      update sicode set idioma='Inglés' where idioma='Ingels'
    </sql>
  </changeSet>
</databaseChangeLog>
```

PASO 2.- Este archivo se pasa como parámetro al programa Liquibase a través de la línea de comandos.
Liquibase --changeLogFile=nombredelarchivoxml.xml update

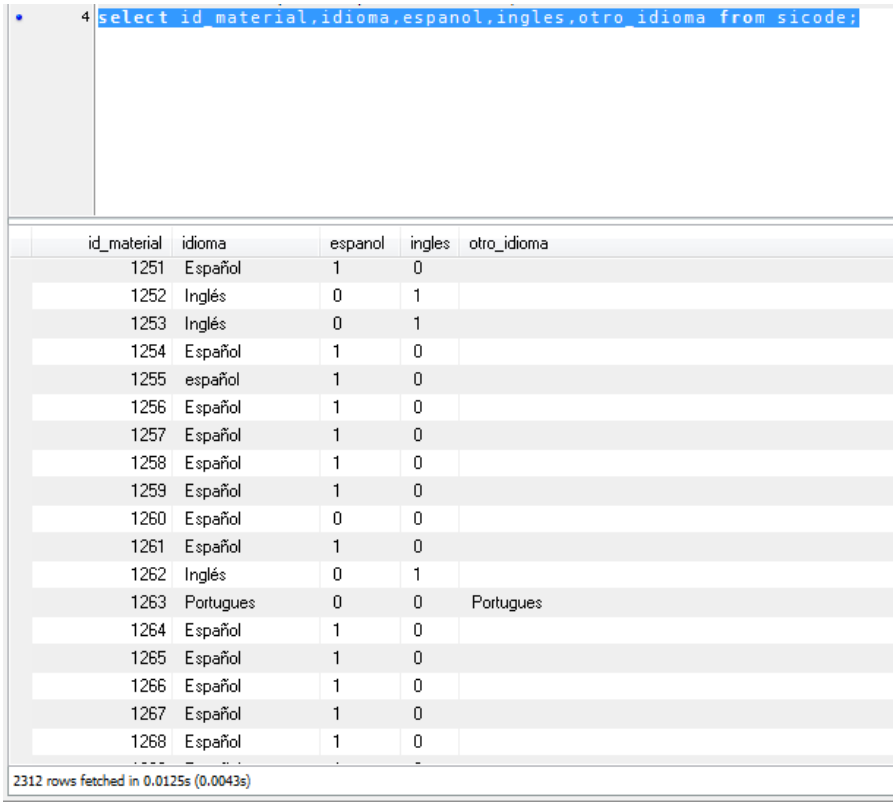
Si es la primera vez que se ejecuta la instrucción update de Liquibase, sobre una base de datos, el sistema creará automáticamente una tabla llamada databasechangelog en la cual se llevará el registro de cada uno de los cambios que se apliquen a la base de datos.

PASO 3.-Una vez que se realizan los cambios sobre el esquema, opcionalmente se puede requerir una migración de datos o también actualizar los programas de acceso externo.

Este es el proceso que se seguirá para cada una de las refactorizaciones que se apliquen al esquema.

Caso: Manejo de los diferentes idiomas en los que se puede encontrar un recurso dentro de la biblioteca.

En la Figura 4.5-2 se muestra el resultado de ejecutar la consulta: **select id_material,idioma,español,ingles,otro_idioma from sicode**. Del resultado de esta consulta se observa que se están tomando en cuenta 4 atributos de la tabla para referirse al idioma de un recurso, lo cual implica tener información repetida o información que se refiere a lo mismo pero está escrita de diferente manera. Al realizar búsquedas sobre el atributo idioma habría que considerar estas 4 columnas de la tabla lo cual podría afectar el desempeño. Si hubiera recursos en más de 4 idiomas, el diseño actual no podría modelarlo.



```
4  select id_material,idioma,espanol,ingles,otro_idioma from sicode;
```

id_material	idioma	espanol	ingles	otro_idioma
1251	Español	1	0	
1252	Inglés	0	1	
1253	Inglés	0	1	
1254	Español	1	0	
1255	español	1	0	
1256	Español	1	0	
1257	Español	1	0	
1258	Español	1	0	
1259	Español	1	0	
1260	Español	0	0	
1261	Español	1	0	
1262	Inglés	0	1	
1263	Portugues	0	0	Portugues
1264	Español	1	0	
1265	Español	1	0	
1266	Español	1	0	
1267	Español	1	0	
1268	Español	1	0	

2312 rows fetched in 0.0125s (0.0043s)

Fig.4.5-2. Resultado de consulta respecto a idiomas en la tabla sicode.

En la Figura 4.5-3 se muestra el modelo relacional correspondiente para los distintos idiomas en que podrían estar los recursos.

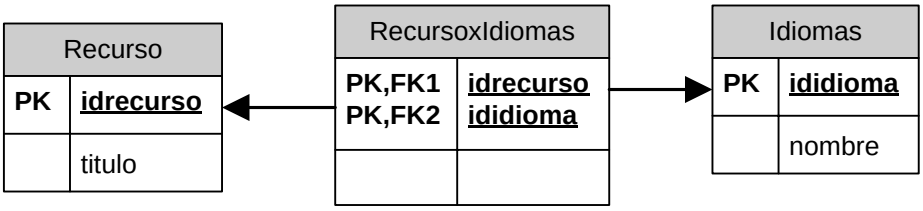


Fig.4.5-3. Modelo relacional de la tabla de idiomas

Para el caso de la tabla idioma se siguió el siguiente procedimiento:

1.- Obtener los diferentes idiomas que se manejan en la tabla, con la finalidad de no tener redundancia de información y también con la finalidad de unificar la información.

Como primer paso para lograr esta tarea se ejecutó la siguiente consulta: **select distinct(Idioma) from sicode**, cuyo resultado se muestra en la Figura 4.5-4. En la imagen se puede observar que además de Inglés, Español, Portugués y Alemán existe un valor vacío. Además, se encontraron diferentes valores para un mismo atributo. Por ejemplo, para el idioma Inglés también se tenía el valor Ingels.

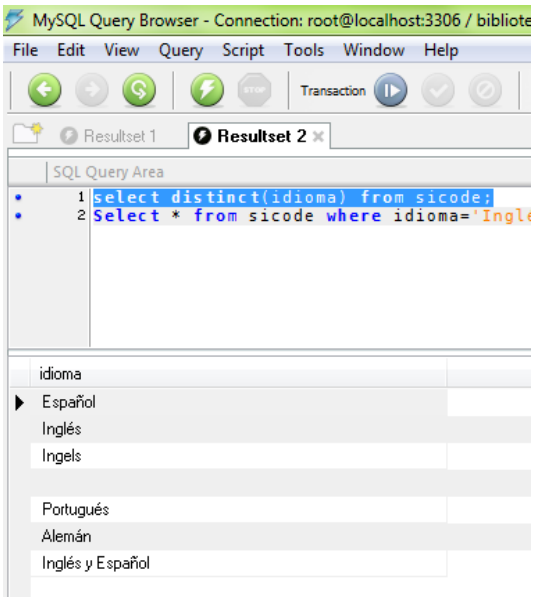


Fig.4.5-4 Resultado de la consulta que muestra los distintos idiomas que se manejan en la tabla sicode.

La primera refactorización que se emplea es Apply Standard Codes. Con ésta, se asegura que no exista variación en los valores para un mismo atributo.
Después de aplicar la refactorización(Figura 4.5-5) sólo tenemos los idiomas: Inglés, Español, Portugués, Alemán y el idioma vacío. Estos valores serán los valores por los que estará compuesta la tabla idiomas.

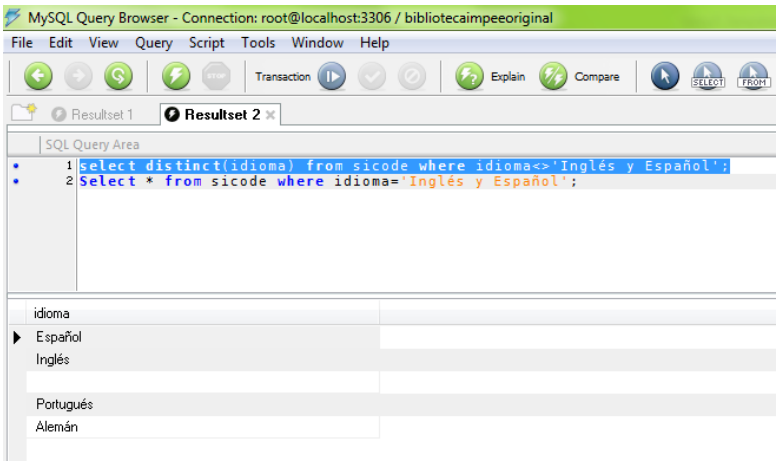


Fig.4.5-5 Resultado de la consulta que muestra los distintos idiomas que se manejan en la tabla sicode después de aplicar una refactorización .

Para crear la tabla idiomas se utilizó la refactorización **Create table**. Esta tabla está compuesta por el atributo *ididioma* y el atributo *nombre*. La llave primaria de la tabla es el atributo *ididioma* el cual es un campo auto numérico.
En la Figura 4.5-6 se muestra el resultado obtenido después de utilizar el programa Liquibase y pasarle como parámetro el archivo XML llamado 2creatablaidiomas.xml.

```
c:\liquibase>liquibase --changeLogFile=scripts/bibliotecaimpee/2creatablaidiomas.xml update
INFO 15/05/13 01:58 PM:liquibase: Successfully acquired change log lock
INFO 15/05/13 01:58 PM:liquibase: Creating database history table with name: `DATABASECHANGELOG`
INFO 15/05/13 01:58 PM:liquibase: Reading from `DATABASECHANGELOG`
INFO 15/05/13 01:58 PM:liquibase: Reading from `DATABASECHANGELOG`
INFO 15/05/13 01:58 PM:liquibase: ChangeSet scripts/bibliotecaimpee/2creatablaidiomas.xml::2::roman ran successfully in 91ms
INFO 15/05/13 01:58 PM:liquibase: Successfully released change log lock
Liquibase Update Successful
```

Fig.4.5-6 Salida de Liquibase después de ejecutar con éxito una refactorización.

Como resultado de la ejecución de esta llamada al programa Liquibase, se generan dos tablas nuevas en la base de datos. Una tabla se llama databaseChangeLog y la otra se llama dataBaseChangeLogLock. La tabla databasechangelog sirve para llevar una bitácora de cada uno de los cambios aplicados a la base de datos. En la Figura 4.5-7, se muestra el registro que contiene esta tabla después de realizar la refactorización **Create table**.

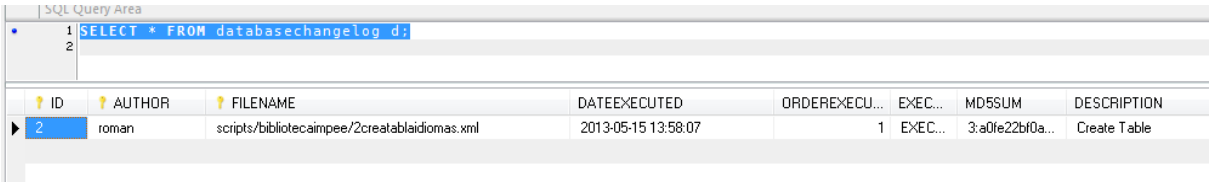


Fig.4.5-7 Resultado de la consulta que muestra el contenido de la tabla databasechangelog.

En la Figura 4.5-8 se muestra la base de datos con la tabla idiomas recién creada y sin ningún registro.

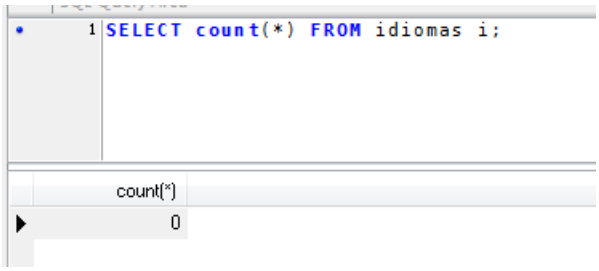


Fig.4.5-8 Resultado de la consulta que cuenta el numero de registros de la tabla idiomas.

Una vez que se ha creado la tabla, el siguiente paso es poblarla. Para lograr esta tarea se utilizó la refactorización **Custom Change** con la cual se puede ejecutar una sentencia SQL. Para este caso la sentencia a ejecutar es**: insert into idioma(nombre) select distinct(idioma) from sicode**.
En la Figura 4.5-9 se muestra el resultado de la ejecución del script en Liquibase.

```
c:\liquibase>liquibase --changeLogFile=scripts/bibliotecaimpee/3poblartablaidiomas.xml update
INFO 15/05/13 03:04 PM:liquibase: Successfully acquired change log lock
INFO 15/05/13 03:04 PM:liquibase: Reading from `DATABASECHANGELOG`
INFO 15/05/13 03:04 PM:liquibase: Reading from `DATABASECHANGELOG`
INFO 15/05/13 03:04 PM:liquibase: ChangeSet scripts/bibliotecaimpee/3poblartablaidiomas.xml::3::roman ran successfully in 110ms
INFO 15/05/13 03:04 PM:liquibase: Successfully released change log lock
Liquibase Update Successful
```

Fig.4.5-9 Salida de Liquibase después de ejecutar con éxito una refactorización.

En la tabla databasechangelog se insertó otro registro, el cual corresponde a la refactorización Custom change (Figura 4.5-10)

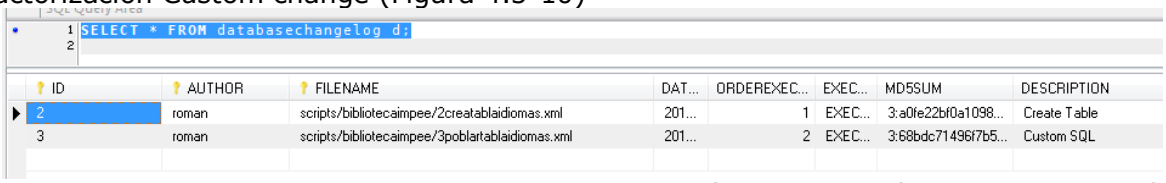


Fig.4.5-10. Contenido de la tabla databasechangelog después de la ejecución de la refactorización Custom change.

Realizando una consulta sobre la tabla idiomas se observa que ya están los 5 registros(Figura 4.5-11)

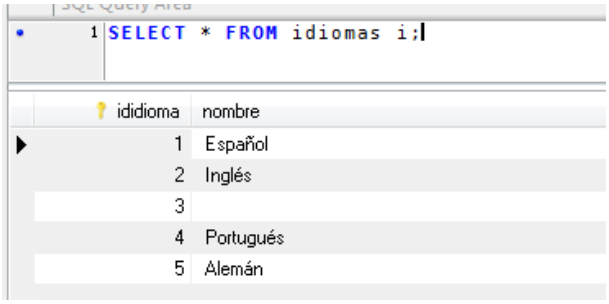


Fig.4.5-11. Resultado de la consulta que muestra el contenido de la tabla idiomas.

El siguiente paso es crear la tabla recursoxidiomas para lo cual se utilizó la refactorización **Create table**. Esta tabla recursosxidiomas se compone de dos atributos. El primero de ellos hace referencia al atributo *ididioma* de la tabla idiomas y el segundo atributo *id_material* hace referencia al atributo *id_material* de la tabla sicode. La llave primaria es una llave compuesta formada por estos dos atributos.

También se utilizaron las refactorizaciones **Create Foreign Key** para cada uno de los atributos de esta tabla, y la refactorización **Create Primary Key** para generar la llave primaria compuesta.
En la Figura 4.5-12 se muestra el resultado de aplicar estos cambios al esquema

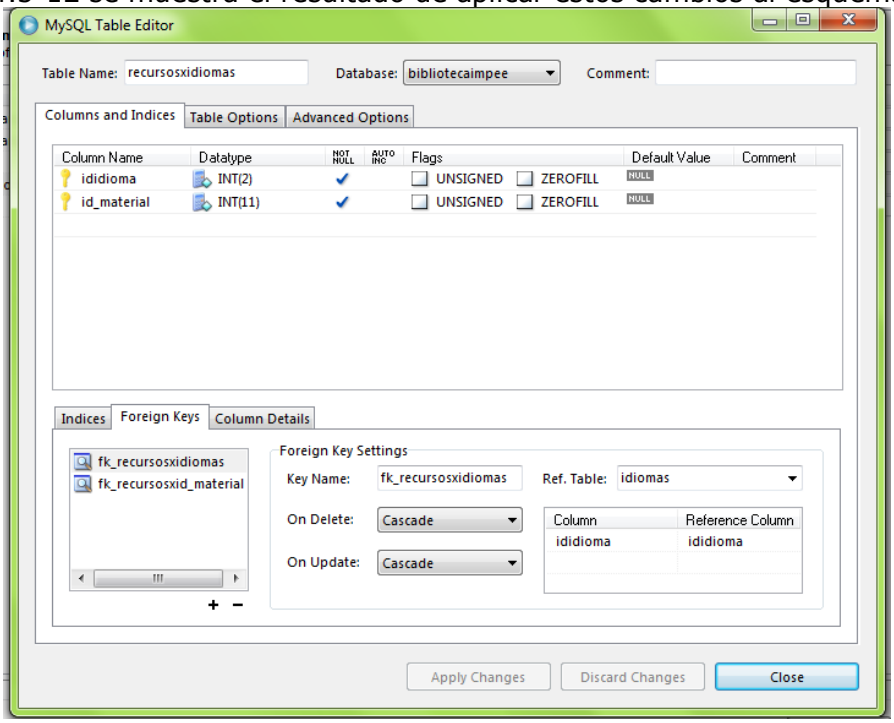


Fig.4.5-12. Estructura de la tabla recursosxidiomas.

En la tabla databasechangelog se agregaron los registros correspondientes a estos 3 últimos cambios(Figura 4.5-13).

```
1 SELECT * FROM databasechangelog d;
```

ID	AUTHOR	FILENAME	DA...	ORD...	EXECTYPE	MD5SUM	DESCRIPTION
2	roman	scripts/bibliotecaimpee/2creatablaidiomas.xml	201...	1	EXECUTED	3:a0fe22bf0a10986c1...	Create Table
3	roman	scripts/bibliotecaimpee/3poblartablaidiomas.xml	201...	2	EXECUTED	3:68bdc71496f7b5ac...	Custom SQL
4	roman	scripts/bibliotecaimpee/4creartablarecursosxidiomas.xml	201...	3	EXECUTED	3:29352c7f1bc0c190...	Create Table
5	roman	scripts/bibliotecaimpee/5creallaveforaneaenrecursosxidio...	201...	4	EXECUTED	3:3b9714829a9e0168...	Add Foreign Key Constraint
6	roman	scripts/bibliotecaimpee/6creallaveforaneaenrecursosxidio...	201...	5	EXECUTED	3:04b414cb73ba1186...	Add Foreign Key Constraint
7	roman	scripts/bibliotecaimpee/7creallaveprimariaenrecursosxidio...	201...	6	EXECUTED	3:fefd80a7c8df215e1...	Add Primary Key

Fig.4.5-13 Resultado de la consulta que muestra el contenido de la tabla databasechangelog.

Como último paso, para terminar con esta tabla (recursosxidiomas), se tuvo que poblar con la información correspondiente al identificador del idioma y al identificador del recurso o material.
Para esto se va a utilizar la refactorización **Custom change**. En esta refactorización se ejecutará la sentencia SQL: **insert into recursosxidiomas(ididioma,id_material) select i.ididioma,s.id_material from sicode s,idiomas i where s.idioma=i.nombre**;

La Figura 4.5-14 muestra que antes de ejecutar la sentencia la tabla está vacía.

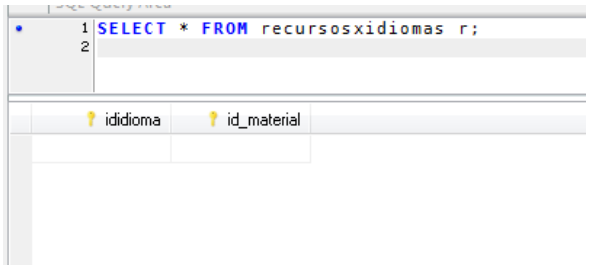


Fig.4.5-14 Resultado de la consulta que muestra el contenido de la tabla recursosxidiomas antes de la inserción de registros.

Después de la ejecución de la sentencia SQL se insertaron 2312 registros, como se muestra en la Figura 4.5-15.

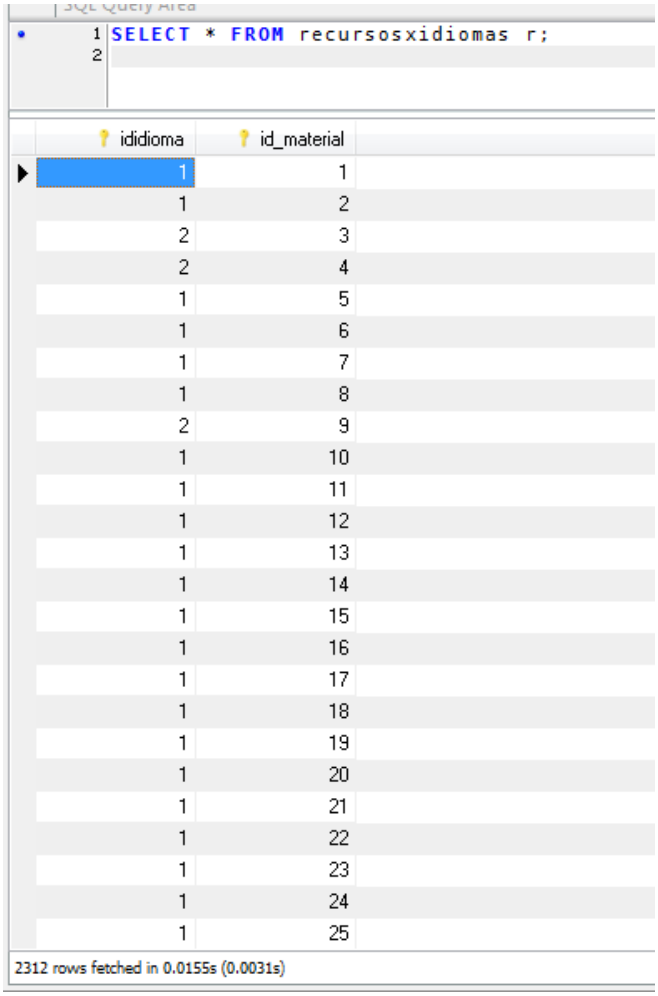


Fig.4.5-15 Resultado de la consulta que muestra el contenido de la tabla recursosxidiomas después de la inserción de registros.

En la Figura 4.5-16 se muestra el contenido de la tabla databasechangelog al término de las refactorizaciones para la tabla recursosxidiomas. Los registros numerados del 9 al 12 se insertaron como consecuencia de aplicar las refactorizaciones **Custom change** e **Insert row**.

Las refactorizaciones Custom change se utilizaron para insertar en la tabla recursosxidiomas algunos registros que no se habían tomando en cuenta en las inserciones anteriores. La

refactorización Insert row se utilizó para insertar en la tabla idiomas el valor correspondiente al idioma Francés.

```
SQL Query Area
1  SELECT * FROM databasechangelog d;
```

ID	AUTHOR	FILENAME	DATEEX...	...	EXECTYPE	MD5SUM	DESCRIPTION
2	roman	scripts/bibliotecaimpee/2creatablaidiomas.xml	2013-05-...	1	EXECUT...	3:a0fe22bf0a10986c1c3a3d17894ef187	Create Table
3	roman	scripts/bibliotecaimpee/3poblartablaidiomas.xml	2013-05-...	2	EXECUT...	3:68bdc71496f7b5ac3493260940cce623	Custom SQL
4	roman	scripts/bibliotecaimpee/4creartablarecursosxidiomas.xml	2013-05-...	3	EXECUT...	3:29352c7f1bc0c190884579c1012f0f1e	Create Table
5	roman	scripts/bibliotecaimpee/5creallaveforaneaenrecursosxidiomasparaididioma.xml	2013-05-...	4	EXECUT...	3:3b9714829a9e0168a7d382424dda881d	Add Foreign Key Constraint
6	roman	scripts/bibliotecaimpee/6creallaveforaneaenrecursosxidiomasparaid_material.xml	2013-05-...	5	EXECUT...	3:04b414cb73ba11861ee18d6731572650	Add Foreign Key Constraint
7	roman	scripts/bibliotecaimpee/7creallaveprimariaenrecursosxidiomas.xml	2013-05-...	6	EXECUT...	3:fefd80a7c8df215e175ba77ca5523894	Add Primary Key
8	roman	scripts/bibliotecaimpee/8poblartablarecursosxidiomas.xml	2013-05-...	7	EXECUT...	3:8cd5306e1f4c186016e4caf7c948225f	Custom SQL
9	roman	scripts/bibliotecaimpee/9poblartablarecursosxidiomas2.xml	2013-05-...	8	EXECUT...	3:ede57bd42f5eb833aa56c8fcd113c9b0	Custom SQL
10	roman	scripts/bibliotecaimpee/10poblartablarecursosxidiomas3.xml	2013-05-...	9	EXECUT...	3:9498c4af1674776308635c979b8fb305	Custom SQL
11	roman	scripts/bibliotecaimpee/11introduciridiomafrancesenidiomas.xml	2013-05-...	10	EXECUT...	3:1bc7df21284283fc21cb58451706bdf6	Insert Row
12	roman	scripts/bibliotecaimpee/12introduciridiomasparaelmaterial1404.xml	2013-05-...	11	EXECUT...	3:3bbf966c5efbaac7b78c1c5e4892bdcb	Custom SQL

Fig.4.5-16. Contenido de la tabla databasechangelog al finalizar las refactorizaciones para la tabla recursosxidiomas

Como prueba de que las refactorizaciones aplicadas mejoran el diseño de la base de datos, se realizó una comparación de dos consultas, en las cuales se extrae el atributo id_material tanto de la tabla sicode como de la tabla creada recursosxidiomas, para los diferentes idiomas(Español, Inglés, etc)

El resultado de las comparaciones fue el siguiente:

- Para el idioma Español(Figura 4.5-17).
  Del lado izquierdo se encuentra el resultado de la consulta sobre la tabla original(sicode). De lado derecho se encuentra el resultado de la consulta sobre la tabla recursosxidiomas.

```
1  select id_material from recursosxidiomas where ididioma=1;
```

id_material
1
2
5
6
7
8
10
11
12
13
14
15
16
17
18
19
20
21

1111 rows fetched in 0.0107s (0.0097s)

id_material
1
2
5
6
7
8
10
11
12
13
14
15
16
17
18
19
20
21

1111 rows fetched in 0.0085s (0.0043s)

1: 4

Fig.4.5-17. Resultado de las comparaciones para el idioma Español

De los resultados obtenidos se observa que el tiempo que tardó la consulta que corresponde a la tabla recursosxidiomas fue menor que la consulta que corresponde a la tabla sicode.

Considerando las mejoras al diseño de la base de datos y los resultados de la prueba anterior, se puede constatar que las refactorizaciones aplicadas hasta el momento aportaron muchos beneficios al esquema.

Caso: Manejo de los diferentes tipos de recursos dentro de la biblioteca.

En la base de datos ***sicode***, se pueden encontrar recursos de diferentes tipos, por ejemplo: Libro, Carpeta, Tesis, etc.
De acuerdo con la estructura de la tabla, se puede observar que se utilizan 8 campos para realizar una consulta y obtener el tipo de recurso(Figura 4.5-18). Los problemas que se tendrían con este esquema son:
1.- Tener información repetida o información que se refiere a lo mismo pero está escrita de manera diferente.
2.- Al realizar búsquedas sobre el atributo tipo habría que considerar estas 8 columnas, lo cual podría afectar el desempeño.
3.- En caso de querer introducir un nuevo tipo de recurso, habría que insertar otra columna.
4.- Si se necesitara clasificar a un recurso con un nuevo tipo, el diseño no podría modelarlo.

```
1  select tipo,libro,encuadernado_rigido,encuadernado_flexible,carpeta,tesis,documento,video from sicode
```

tipo	libro	enc...	enc...	car...	tesis	doc...	video
Libro	1	0	0	0	0	0	0
Libro	1	0	0	0	0	0	0
Carpeta	0	0	0	1	0	0	0
Carpeta	0	0	0	1	0	0	0
Carpeta	0	0	0	1	0	0	0
Libro encuadernado verde	1	1	0	0	0	0	0
Libro	1	0	0	0	0	0	0
Manual	1	0	0	0	0	0	0
Diccionario	1	0	0	0	0	0	0
Diccionario	1	0	0	0	0	0	0
Documento encuadernado	0	0	1	0	0	1	0
Documento encuadernado	0	0	1	0	0	1	0
Documento encuadernado	0	0	1	0	0	1	0
Documento encuadernado	0	0	1	0	0	1	0
Documento encuadernado	0	0	1	0	0	1	0
Libro	1	0	0	0	0	0	0
Libro	1	0	0	0	0	0	0
Tesis	0	0	0	0	1	0	0

2312 rows fetched in 0.0177s (0.0019s) Edit Apply Changes

Fig.4.5-18 Resultado de la consulta que muestra los tipos de recurso.

En la Figura 4.5-19 se muestra el modelo relacional correspondiente para los diferentes tipos de recurso.

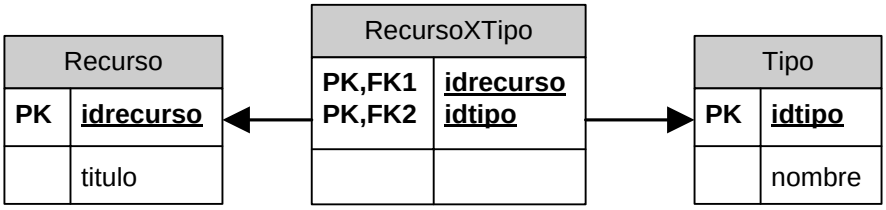


Fig.4.5-19 Modelo relacional de los tipos de recurso.

Los problemas que presenta la tabla para modelar el atributo "tipo" son los mismos que se presentaban en el caso anterior para el atributo "idioma". Por lo que sólo se describirán de manera general los pasos a seguir para refactorizar la tabla ***sicode*** siguiendo el mismo procedimiento utilizado en el caso del atributo idioma.

1.- Obtener los diferentes tipos de recurso que se manejan en la tabla.
2.- Si existen diferentes valores para el atributo tipo, aplicar la refactorización Apply Standard Codes.
3.- Crear la tabla Tipo mediante la refactorización Create Table. Esta servirá para manejar los diferentes tipos de recurso.
4.- Poblar la tabla Tipo mediante la refactorización Custom Change.
5.- Crear la tabla RecursoXTipo asignándole como atributos las llaves primarias de la tabla recurso y de la tabla Tipo.
6.- Utilizar la refactorización Create Foreign Key para crear integridad referencial con las tablas recurso y tipo.
7.- Utilizar la refactorización Create Primary Key para generar la llave primaria compuesta en la tabla RecursoXTipo.
8.- Poblar la tabla RecursoXTipo por medio de la refactorización Custom Change.

Una vez que se ha completado el proceso de refactorización para esta tabla, se ejecutará el paso 3 del proceso de refactorización, el cual consiste en actualizar los programas de acceso externo, debido a que el atributo Tipo si se toma en cuenta para realizar consultas desde la aplicación (mencionada en la sección 4.1).

Una de las estrategias que se utilizan para refactorizar una base de datos es encapsular el acceso a la base de datos (Ambler & Sadalage, 2006). Con esto se logra que sea más fácil de refactorizar la base de datos. Encapsular el acceso a la base de datos se puede lograr implementando objetos de acceso a los datos(DAOs), los cuales son clases que implementan la lógica de acceso a los datos de manera separada de las clases del negocio. Para esta aplicación se utilizarán Objetos de acceso a datos PHP(PDOs) debido a que es una aplicación implementada con el lenguaje de programación PHP.

Actualmente el acceso a la base de datos desde la aplicación se realiza a través de sentencias SQL incrustadas en cada página PHP. Para cambiar esta forma de acceso y evitar la repetición de código SQL se implementará una clase PDO llamada database_handler.php. En esta clase se desarrollarán métodos los cuales proveerán funcionalidad genérica para la conexión a la base de datos y para la manera de realizar las consultas.

El siguiente paso es crear la clase de negocio recursoPDO.php, la cual se encargará de realizar las consultas a la base de datos de la biblioteca. En ésta clase se hace uso de procedimientos almacenados, los cuáles servirán para agilizar los tiempos de consulta.

Estos procedimientos almacenados se crearán en la base de datos a través de la refactorización Create Procedure.

El último paso para terminar de refactorizar el código de la aplicación es hacer uso de las clases de acceso a datos creadas anteriormente. Esto dará como beneficio la reutilización de código y evitará la duplicación de código SQL, con lo cual será más sencillo realizar refactorizaciones posteriores.

En parte del proceso de refactorización se utilizó la refactorización Drop Column para eliminar algunas columnas que no tenían relación con la aplicación o que no tenían valores asignados para ningún registro. Como ejemplo de estas columnas estaban: documento_prestado, prestado_a, fecha_de_prestamo y fecha_de_devolucion.

Nota: Debido a que el proceso a seguir para los demás atributos de la tabla es similar, sólo se muestran estos dos casos que son los más representativos.

De acuerdo con la información contenida en la base de datos y después de realizar un análisis de la misma y aplicar las distintas refactorizaciones, se llegó al diseño de base de datos mostrado en la Figura 4.5-20.

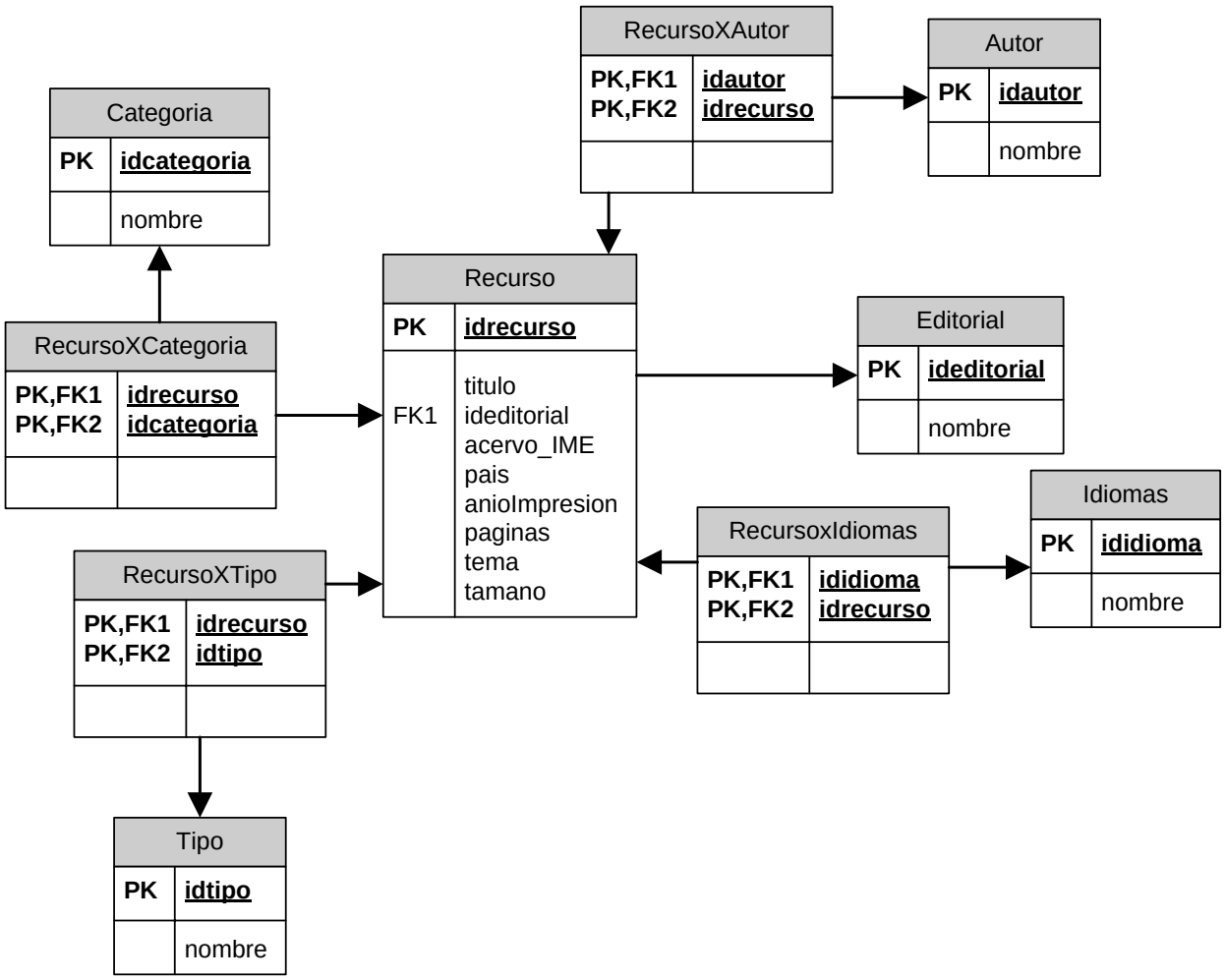


Fig.4.5-20 Diseño final después aplicar las refactorizaciones.

Las ventajas que presenta este diseño con respecto al diseño original de la base de datos son:

1.- Se tiene una mejor organización de la información que compone la base de datos, cada entidad se separó en una tabla distinta para una mejor claridad y mantenimiento.

2.- La creación de índices ayuda en el desempeño de las consultas, obteniendo los resultados de la consulta de manera más rápida.

3.- Permite agregar nuevos valores para un atributo sin tener que agregar más columnas a la tabla, como se hacía originalmente.

4.- Se evita dentro de lo posible la redundancia de información.

5.- Se mantiene la integridad referencial.

CONCLUSIONES.

En este trabajo de tesis:

1.- Se construyó una aplicación web para el proceso de refactorización de una base de datos. Esta aplicación se construyó siguiendo el Proceso Unificado como metodología de desarrollo de software. Además, se utilizó el patrón de arquitectura modelo-vista-controlador para una mejor separación de la funcionalidad de cada una de las capas y se integró AJAX para lograr una mejor interactividad entre el usuario y la aplicación.

2.- Se aplicó el proceso de refactorización a una base de datos en operación, empleando para ello la aplicación web desarrollada anteriormente y la biblioteca Java Liquibase.

Las siguientes conclusiones están dadas en términos de estos dos puntos anteriores.

Para el punto 1:

- Se diseñó un sistema web para generar archivos XML que enriqueció el uso de la biblioteca Java Liquibase, aportando una mejora a los programas o sistemas actuales que se utilizan para aplicar el proceso de refactorización de una manera disciplinada.

- La metodología de desarrollo utilizada permitió administrar y controlar mejor el proyecto. La integración de ciclos, fases, flujos de trabajo, mitigación de riesgo, control de calidad y control de configuración hace que se puedan afrontar los cambios que se presenten y abordarlos de manera iterativa e incremental. Además de producir software de mejor calidad.

- El patrón de arquitectura MVC sirvió para separar de manera lógica la funcionalidad de cada una de las capas, aportando al proyecto una mejor claridad al código dando como consecuencia que los errores fueran más fáciles de detectar y corregir.

- La tecnología web AJAX fue muy importante en este proyecto. Por medio de la funcionalidad que ofrece, se pudo lograr la creación dinámica o "al vuelo" de los formularios que se presentan al usuario en la pantalla principal de la aplicación. Esto permitió una mejor interactividad entre el usuario y la aplicación web construida.

- El uso de las bibliotecas Java: XSOM y JDOM, permitieron profundizar en las características de XML y DOM. Incorporar estas bibliotecas al proyecto también fue parte fundamental en la creación de la aplicación web, ya que XSOM sirvió para analizar un esquema XSD, obtener las partes o elementos que lo formaban y posteriormente, por medio de estos elementos construir dinámicamente los formularios. Por otra parte, JDOM sirvió para construir el documento XML de salida, basándose en la información recabada con los formularios mencionados anteriormente.

Para el punto 2:

- El desarrollo de la aplicación web permitió agilizar la creación de los archivos XML con los cuales se alimenta a Liquibase. Estos archivos XML generados contienen las sentencias SQL necesarias para llevar a cabo una refactorización en la base de datos. La aplicación fue muy útil debido a que no hace necesario que se tengan conocimientos tanto de XML como de esquemas XML para poder realizar una refactorización. Al mismo tiempo, la aplicación web ayudó a disminuir la cantidad de errores que se presentaban cuando se construían los archivos XML de manera manual.

- El proceso de refactorización de una base de datos, no es una tarea sencilla. Cada refactorización que se aplique requiere realizar una serie de pasos para mantener el estado de la información en la base de datos y al mismo tiempo mejorar el diseño de su esquema. Entre esos pasos están:

    - Cambiar el esquema
    - Migrar datos: para llevar a cabo esta tarea se deben de tener conocimientos de SQL y del manejador de bases de datos utilizado.
    - Actualizar programas de acceso externo: para llevar a cabo esta tarea de deben tener conocimientos tanto del lenguaje de programación en el que se encuentre desarrollada la aplicación que accesa a la base de datos, así como de SQL.

- El uso de Liquibase permitió realizar los cambios a la base de datos, de forma ordenada y al mismo tiempo llevar un control de los cambios aplicados.

- Después de aplicar el proceso de refactorización sobre la base de datos, se pudo verificar que el diseño del esquema mejoró notablemente. Conservó la semántica de la información y la semántica del comportamiento. Esto dió como resultado consultas más rápidas, redundancia de información mínima, asegurar la consistencia de la información y mantener la integridad referencial.

TRABAJO FUTURO

Para un trabajo futuro se propone realizar el proceso de refactorización en un escenario multiaplicación, empleando para ello las herramientas de software utilizadas en este trabajo de tesis.

También se propone realizar el proceso de refactorización sobre otro manejador de bases de datos.

Ambler, S. W., & Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley.

Darie Cristian, B. E. (2008). *Beginning PHP and MySQL E-Commerce: From Novice to Professional, Second Edition.* Apress.

de Miguel Adoración, M. P. (2001). *Diseño de bases de datos, Problemas resueltos.* Alfaomega.

Deitel, D. y. (1998). *Como programar en Java.* Prentice Hall.

Fain, Y. (2011). *Java® Programming 24-Hour Trainer.* Wiley Publishing Inc.

Larman, C. (2002). *UML y Patrones.* Pearson.

Pratt Philip J., L. M. (2009). *SQL.* Anaya.

Ramez Elmasri, S. B. (2007). *Fundamentos de sistemas de bases de datos.* Pearson.

Rosa, K. E. (2005). *SQL SERVER Bases de Datos Robustas y Confiables.* MP Ediciones.

Sierra, A. J. (2008). *Ajax en J2EE.* Alfaomega.

Silberschatz, A. (2002). *Fundamentos de bases de datos.* McGraw-Hill.

Sitio web de JDOM. (s.f.). Recuperado el 2013, de www.jdom.org

Sitio web de Liquibase. (s.f.). Recuperado el 2012, de www.liquibase.org

Sitio web de XSOM. (s.f.). Recuperado el 2012, de https://xsom.java.net/

Viera, R. (2007). *Programación con SQL Server 2005. Fundamentos.* Anaya.

Villalobos Jorge A., C. R. (2006). *Fundamentos de programación.* Pearson.

Wang, P. S. (2000). *Java con programación orientada a objetos y aplicaciones en la WWW.* Thomson.

Weitzenfeld, A. (2005). *Ingeniería de Software orientada a objetos con UML, Java e Internet.* Thomson.

XML Schema Part 1: Structures Second Edition. (s.f.). Recuperado el 2012, de http://www.w3.org/TR/xmlschema-1/