UNIVERSIDAD NACIONAL DE SALTA

FACULTAD DE CIENCIAS EXACTAS

SEMINARIO DE COMPUTADOR UNIVERSITARIO

TRABAJO FINAL:

COMPONENTE GESTOR DE BASE DE DATOS PARA JAVA

"JAVA DATA BASE GENERIC MANAGER"

CRUZ, NELSON EFRAIN ABRAHAM

Director:
RAMIREZ, JORGE FEDERICO

Abril 2013

A mi madre Sara que siempre supo ser el pilar de mi familia, a mis hermanos Rafael y Silvia que siempre me alentaron y soportaron.

A mi fallecido padre que casi no conocí, pero se que fue de el que herede esa curiosidad que siempre me invade.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En cualquier sistema informático es común encontrarse con el uso de bases de datos relacionales para manejar, valga la redundancia, los datos que este debe procesar. Estos datos deben ser leídos, actualizados y registrados por el sistema en la base de datos, pero el sistema no es el que controla la base de datos si no que es el *Sistema de Gestión de Base de Datos* o *Motor de base de datos* o *DBMS*[1] el que controla la base de datos y cualquier acción que se quiera realizar sobre la base de datos se hace a través del motor mediante el lenguaje SQL, es decir el sistema informático debe comunicarse con un motor de base de datos.

En el mercado existen muchos *DBMS* relacionales[2] Microsoft SQL Server, MySQL y Oracle son solo algunos de ellos, esta amplia variedad crea algunos problema de compatibilidad puesto que cada uno de estos motores define un dialecto de SQL particular, muy a pesar de que SQL es un estándar de bastante antigüedad. El echo de por que se usan estos dialectos en vez del estándar corresponden a factores que no interesa discutir pero se puede nombrar como ejemplo la necesidad de mantener a los clientes atados a una plataforma en particular.

La comunicación con el motor se hace a través de un *driver* que por lo general es provisto por el desarrollador del motor y es especifico para este dependiendo de la plataforma sobre la que se esté trabajando. Entonces al desarrollar un sistema informático que utilice una base de datos, se debe tener en cuenta que exista el *driver* para el lenguaje de programación que se esta por utilizar y además que al desarrollar con un determinado motor el sistema queda atando en cierta medida al uso de esté. Esta dependencia con el motor puede ser minimizada dependiendo de como se diseñe el sistema, en este sentido se debe pensar que el sistema debe concentrarse en los datos y no en como obtener los datos, por ejemplo se puede pensar el caso de un sistema de ventas, desarrollado bajo POO, en el que existe una clase `Ventas` que se encarga de gestionar las ventas, las cuales es necesario que estén registradas en una base de datos, una primera solución sería que dicha clase se encargue de comunicarse con el motor y de grabar los datos, lo cual es posible pero se presentarían dos cuestiones: primero la clase perdería cohesión pues se estaría encargando de tareas extras y segundo la clase estaría interviniendo en el acceso al motor por lo que esta debería conocer más cosas de la que debe, es decir habría un mayor acoplamiento entre las diferentes clases, estas cuestiones hacen a la dificultad de mantenimiento del software que es una clave importante para la subsistencia del mismo. Una segunda solución seria que otra clase se encargue de realizar la persistencia de los datos, de este modo `Ventas` no tiene que conocer como es que se están persistiendo los datos solo necesita enviarle un mensaje a la otra clase y dejar que ella se encargue del trabajo. En el anterior ejemplo para la primer solución se estaría acentuando la dependencia con un motor en particular a la vez que se crear otros problemas, en cambio en la segunda solución se tiene una menor dependencia y se eliminan algunos problemas de diseño.

El presente trabajo pretende desarrollar en principio una capa de abstracción para manejar distintos motores de bases de datos desde el lenguaje Java ocultando los detalles correspondientes al acceso a cada motor en particular, inicialmente se dará soporte para 3 motores pero la estructura del componente permitirá agregar fácilmente soporte para más motores. El nombre

[1] Por sus siglas en ingles

[2] Se usara *DBMS* como sinónimo de *RDBMS* a lo largo de todo el texto pero hay que tener en cuenta que *DBMS* es un termino genérico que no se refiere solo a motores relacionales.

del proyecto es *Java Data Base Generic Manager* o *JDBGM* que es como sera referido de aquí en más.

1.1. Objetivos generales

El presente proyecto precisa que se realicen dos tareas básicas bien distintas, la primera es el estudio de los diferentes *DBMS* para aprender como es que se manejan esos motores y poder conocer tanto las características en común como aquellas en las que difieran, la segunda tarea tiene que ver con estudiar el modo de diseñar el proyecto, es decir de que modo estará estructurado *JDBGM* para poder permitir una fácil ampliación y un fácil mantenimiento. Así se pueden enumerar los siguientes objetivos generales del proyecto:

1. Familiarizarse con la lectura de documentación de proyectos.
2. Investigar sobre el uso de patrones de diseño.
3. Investigar como se diseña una API[3].

1.2. Tecnología utilizada

El proyecto sera desarrollado sobre el lenguaje de programación Java y los motores de base de datos *MySQL*, *PostgreSQL* y *SQLite* que son a los que inicialmente se les pretende dar soporte. Como herramientas para asistir el desarrollo se utilizara el entorno de desarrollo Eclipse que esta especializado en Java y que además agrega otras herramientas como por ejemplo la librería de pruebas unitarias JUnit. Por otro lado se usara el sistema de control de versiones Git que a pesar de que se puede integrar en Eclipse, se prefirió manejarlo de manera separada para poder aprender de manera más profunda su uso. Para el desarrollo de este informe se uso Latex junto con el "IDE" Texmaker. Todo el proyecto se desarrollo sobre el sistema operativo Ubuntu principalmente por la familiaridad y facilidad de uso de las herramientas que se están utilizando sobre este tipo de sistemas, por ejemplo Git fue inicialmente creado para ser usado en entornos *nix[4].

1.3. Organización del trabajo

El presente trabajo se organiza de la siguiente forma. El Capítulo 1 introduce a el objetivo de el trabajo y lo enmarca dentro de una tecnología determinada además de presentar los objetivos generales que se persiguen. El Capítulo 2 expone algunas de las herramientas existentes que resuelven desde distintos enfoques la problemática que ataca este trabajo, nombrando entre ellas ORM, generadores de código y otros. El Capítulo 3 introduce formalmente la problemática que se esta atacando e introduce el concepto de Patrones de Diseño para terminar introduciendo la solución que se propone. El Capítulo 4 es la especificación formal de la solución que se propone, para ello se divide el proyecto en dos módulos internos, un manejador de sentencias y un API para el acceso a motores de base de datos. El capitulo 5 documenta los aspectos importantes de el diseño sobre el que se construyo el proyecto, siguiendo para ello la división en dos módulos principales que se introdujo en el capitulo anterior. El Capítulo 6 documenta los pormenores, ajustes y correcciones que se tuvieron que hacer durante la implementación del proyecto y finalmente se comenta el modo en que se realizaron las pruebas sobre el código escrito. Para terminar con los capítulos se escribió una breve reseña sobre los resultados que se obtuvieron del proyecto y el trabajo a futuro que se espera realizar.
Como Apéndice de este informe se creo un manual de usuario que cubre los aspectos básicos de uso de la herramienta que se desarrollo y una descripción de una simple aplicación de ejemplo que muestra un uso practico de la herramienta.

[3] *Application Programming Interface*
[4] Aunque actualmente se lo puede usar en Windows/Linux/Mac

Capítulo 2

Antecedentes

En este capitulo se expondrán algunas de las soluciones ya existentes en el mercado que cubren la misma problemática que intenta resolver este proyecto, pero desde diferentes enfoques. Algunas de estas herramientas proveen soluciones mucho más completas que la que presenta este proyecto pero a su ves son también mucho más complejas de utilizar.

2.1. Generadores de código

Algunas de las soluciones existentes en el mercado resuelven el problema de la abstracción del uso de la fuente de datos trabajando sobre el modelo de datos para generar código de manera automática, en dichas herramientas el modelo debe ser pasado a un interprete que mediante reglas prefijadas es capaz de generar el código necesario para comunicarse e interactuar con el motor, luego este código puede ser usado como parte del software que se esta desarrollando.
Un ejemplo de estos generadores de código es MDAOG una herramienta libre que puede ser encontrado en la pagina web mdaog.sourceforge.net, el generador tiene una interfaz gráfica sencilla de utilizar. Como se puede ver en la figura 2.1, solo se necesitan configurar algunos parámetros y la herramienta generara el código que luego puede ser directamente usado en una aplicación web, pues esta herramienta esta enfocado en el ámbito de JEE[1] que es el SDK de Java enfocado en aplicaciones web. Inicialmente soporta únicamente el *DBMS PostgreSQL* pero según su pagina web no se descarta el soporte de múltiples motores en el futuro. Como base para la generación de código de esta herramienta se usa el patrón de diseño DAO el cual sera introducido más adelante pues es también base de inspiración para este proyecto.
Una **ventaja** del uso de generadores de código es que no se agrega procesamiento extra a el programa, por ejemplo no hay por debajo un componente que se este encargando de realizar tareas si no que es el código generado el que directamente realiza estas tareas, una **desventaja** de los generadores de código es que siempre se esta trabajando sobre una misma plantilla[2] a partir de la cual se genera el código, por lo que siempre se tendera a crear código de más por que no es posible o viable (pensando en un auto-generador) estar ajustando la plantilla de acuerdo a las necesidades especificas de cada tabla o base de datos pues se pierde la gracia de este tipo de herramientas.

2.2. Herramientas ORM

Las herramientas de mapeo objeto-relacional o *Object-Relational Mapping* o ORM que es como usualmente son conocidas, se encargan de eliminar de cierto modo la diferencia de paradigmas existente entre una aplicación que es orientada a objetos y un almacenamiento de datos que sigue un modelo relacional, creando virtualmente una base de datos orientada a objetos sobre la base relacional. De este modo se eliminan muchas tareas extras relacionadas con el mapeo de por ejemplo un atributo de un objeto a la fila y columna correspondiente a una tabla de una base de datos relacional, además se adquieren beneficios extras al poder utilizar el paradigma POO directamente sobre los datos.

[1] *Java Enterprise Edition*, más información en la web de Oracle
[2] EL conjunto de reglas prefijadas que sirven para crear el código.

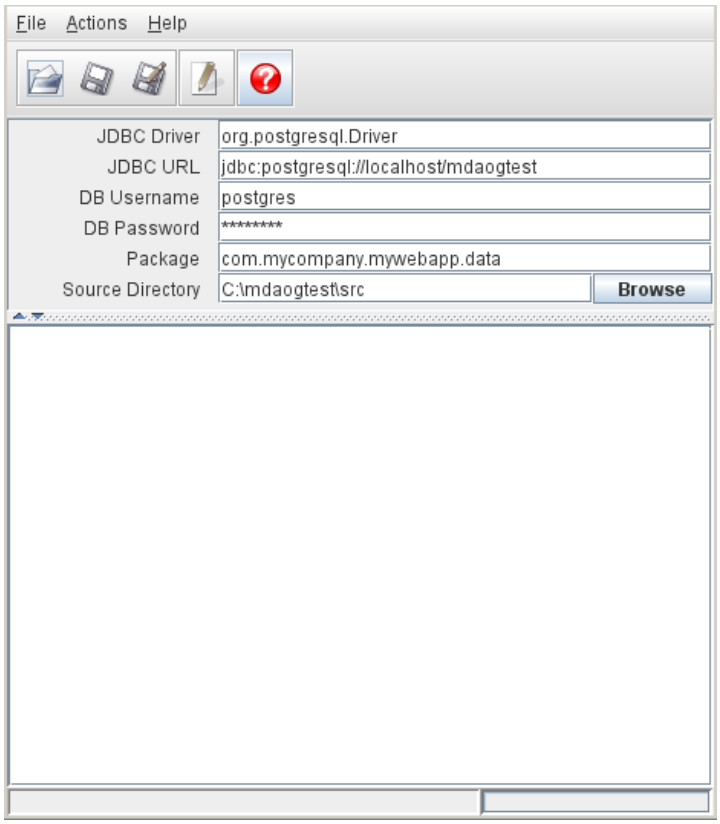


Figura 2.1: Captura de pantalla de la interfaz de MDAOG

Uno de los ejemplos más conocidos de estas herramientas es Hibernate, esta es una herramienta para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación mediante archivos declarativos (XML) o anotaciones en los *beans*[3] de las entidades que permiten establecer estas relaciones.

Hibernate es una herramienta bastante poderosa una vez que se aprende a utilizarla por completo, además decir que es solo un ORM es menospreciar las capacidades de la herramienta puesto que esta termino convirtiéndose en un conjunto de herramientas diferentes relacionadas entre si que brindan una solución mucho más completa. El código que se puede leer a continuación de una pequeña y simplificada muestra del uso de la herramienta:

Mínimo ejemplo de Hibernate: guardando datos en la DB

```
1 Session session = sessionFactory.openSession();
2 session.beginTransaction();
3 session.save( new Event( "Our␣very␣first␣event!", new Date() ) );
4 session.save( new Event( "A␣follow␣up␣event", new Date() ) );
5 session.getTransaction().commit();
6 session.close();
```

A diferencia del patrón DAO, ORM que es en realidad una técnica y no un patrón, trata específicamente el problema de persistencia de datos en bases de datos relacionales mientras que DAO en este sentido es más genérico pues considera otros tipos de soluciones para persistencia de datos. Y a diferencia de los generadores de código con los ORM no se genera código si no que se construye a medida con la herramienta. La única desventaja que se puede encontrar es que al tratarse de una herramienta tan completa, esta pueda terminar siendo demasiado para algunas situaciones, como se dice vulgarmente seria como matar moscas con un cañón.

[3]Los JavaBeans son un modelo de componentes para la construcción de aplicaciones en Java.

2.3. Otras Soluciones

Las diferentes soluciones existentes para lidiar con la abstracción de uso de las fuentes de datos se basan en técnicas o patrones de diseño preexistentes que brindan una base mejor fundamentada para el desarrollo de las mismas, como ultimo ejemplo de este tipo de soluciones podemos nombrar aquellas basadas en el patrón *Active Record* en que un Objeto esta relacionado a una fila de una tabla por lo que una creación de un objeto equivale a crear una nueva fila en la tabla mediante una acción de guardar, lo mismo con las actualizaciones de datos de un objeto se reflejan en actualizaciones en la tabla.
Un ejemplo de una herramienta que trabaja sobre *Active Record* es *activejdbc* que puede ser encontrado en code.google.com/p/activejdbc, es una herramienta relativamente nueva que se publico por primera vez en el 2010 bajo la licencia Apache 2.0. Como una característica interesante de *activejdbc* se puede resaltar que no es necesario especificar el modelo de datos, este es inferido directamente desde la base de datos. En el ejemplo siguiente es suficiente que exista la siguiente tabla en la base de datos:

Ejemplo de uso de activejdbc: tabla que debe existir en la BD

```
CREATE TABLE people (
  id  int(11) NOT NULL auto_increment PRIMARY KEY,
  name VARCHAR(56) NOT NULL,
  last_name VARCHAR(56),
  dob DATE,
  graduation_date DATE,
  created_at DATETIME,
  updated_at DATETIME
  );
```

para que se pueda crear una nueva fila correspondiente a una persona `person` en la tabla `people` (plural de person) mediante las siguientes lineas de código:

Ejemplo de uso de activejdbc: creando una nueva fila en una tabla

```
Person p = new Person();
p.set("name", "Marilyn");
p.set("last_name", "Monroe");
p.set("dob", "1935-12-06");
p.saveIt();
```

Como comentario final para cerrar este capitulo se quiere comentar que el espacio que quiere ocupar este proyecto es el de una herramienta que sea sencilla de utilizar, con mínimas dependencias de librerías externas y con la menor carga posible en el uso de recursos, que se usara el patrón DAO como base de diseño pero no de una manera completa puesto que se requeriría crear una suerte de generador de código para cubrir completamente los requerimientos de DAO, en vez de eso se brindaran las herramientas para que la implementación de DAO resulte sencilla.

Capítulo 3

JDBGM

En este capitulo se introducirá formalmente el proyecto propuesto por este trabajo, se empezara por recordar la problemática expuesta en el capitulo 1 pero de una manera más técnica, además se enmarcara el proyecto en un escenario más especifico definiendo el lenguaje sobre el cual se desarrollara y los *DBMS* a los que se les dará soporte, para luego finalmente exponer la solución propuesta.

3.1. El problema

Como se empezó a describir en la introducción del capitulo 1 una de las dificultades que se pueden encontrar al usar motores de bases de datos en el desarrollo de sistemas informáticos se traduce en problemas asociados a la dependencia existente con el uso de un motor de base de datos en particular, estos problemas se pueden ver genéricamente como problemas de mantenibilidad y portabilidad:

- Problemas en la portabilidad: Al utilizar un motor[1] en particular se esta atando al software en mayor o menor medida al uso de este según como haya sido diseñado el sistema, el mayor problema al que hay enfrentarse se traduce en las diferentes sintaxis para SQL que define cada *DBMS*. Culpa de ello es muy probable que una sentencia valida para un motor no lo sea para otro, por lo tanto a la hora de querer migrar desde un motor a otro e inclusive a una versión más nueva del mismo motor es necesario "actualizar" las sentencias para que se apeguen a la sintaxis del motor al que se pretende migrar. Además como parte de su sintaxis particular cada *DBMS* define sus propios tipos de datos lo que añade un poco más de dificultad al proceso de cambio o migración desde un motor a otro.

- Problemas en el mantenimiento: al utilizar bases de datos es importante como se diseña el acceso y manejo de los *DBMS*, hay que tener en cuenta las siguientes cuestiones: quien se tiene que hacer cargo de la persistencia de los datos? El programa tiene que conocer los datos o más bien como obtener los datos? Estas cuestiones no son nuevas y se solucionan en parte siguiendo patrones de diseño. Un ejemplo más conciso de esto lo podemos ver cuando se esta diseñando un modulo de un sistema que precisa persistir ciertos datos en una BD, la pregunta que hay que hacerse es quien debería hacerse cargo de realizar esta persistencia? Si lo hace el modulo en si se encontrarían accesos a la base de datos mezclados con la lógica de negocio del modulo, además estos accesos implican que el modulo debe conocer como conectarse con la base de datos y de que modo se deben enviar y recibir los datos, con lo que el modulo perdería cohesión y ganaría complejidad derivando todo esto en una mayor dificultad en el mantenimiento.

Estos dos puntos están fuertemente relacionados pues la mantenibilidad del software esta muy ligada a la portabilidad y viceversa, pero cuando se habla de que el software sea mantenible se esta hablando de muchos aspectos más.
Este trabajo se centrara en el uso de la base de datos y en como esto afecta a la mantenibilidad del software. Como ya se dijo es de buena practica el uso de patrones de diseño, así que en

[1] Hablando de motores de bases de datos

la sección siguiente se explicara brevemente que son los patrones y como estos van a ayudar a encontrar una base de diseño para el proyecto en el que se esta trabajando.

3.2. Patrones de diseño

Un patrón de diseño es una solución genérica y reusable a un problema que ocurre de manera frecuente en un contexto dado. Un patrón de diseño no es un diseño terminado que pueda ser transformado directamente en código es más bien una guía que indica como resolver un problema en determinados escenarios. Así los patrones de diseño son buenas costumbres que uno mismo debe implementar en la aplicación a desarrollar y en este mismo sentido son recomendaciones a tener en cuenta y no obligaciones[2]. Además es importante notar que el uso de patrones no garantiza éxito a la hora de diseñar, la descripción de un patrón indica cuando este puede ser aplicable, pero solo la experiencia indicara adecuadamente cuando el uso de un patrón de diseño en particular mejorara el diseño del software[12].
Entonces por que tener en cuenta a los patrones?

- Han sido probados. Los patrones reflejan la experiencia, conocimiento y perspectiva de desarrolladores quienes han aplicado satisfactoriamente estos patrones en su propio trabajo.
- Son reusables. Los patrones proveen soluciones ya descubiertas que pueden ser aplicadas a diferentes problemas.
- Son expresivos. Los patrones proveen un vocabulario común de soluciones que pueden expresar soluciones extensas de manera concisa.

Existe mucha teoría sobre los patrones pero en este trabajo no interesa ahondar sobre ellos, si no que interesa introducir el concepto al lector para poder presentar de mejor manera un patrón de diseño que servirá de base para el desarrollo de *JDBGM*.

3.2.1. Data Acces Object

Antes de describir este patrón es necesario ubicarse en un **contexto**, al acceso a los datos depende de la fuente de datos con la que se esté trabajando. El acceso a almacenamiento persistente de datos, tal como una base de datos, varia fuertemente dependiendo del tipo de almacenamiento (BD relacionales, BD orientadas a objetos, archivos planos, etc) y de la implementación de un proveedor en particular.
El **problema** se da por que en cierto punto las aplicaciones necesitan persistir sus datos. Para muchas aplicaciones, la persistencia de los datos es implementada mediante diferentes mecanismos y hay marcadas diferencias en las API's usadas para acceder a estos diferentes mecanismos. Otras aplicaciones quizás necesite acceder a datos almacenados en diferentes sistemas muy distintos del cual se esta trabajando los cuales exigen utilizar sus API's las cuales usualmente son propietarias (no se puede acceder a el código de la misma). Esta disparidad entre las diferentes fuentes de datos produce desafíos en el diseño y además crea una potencial dependencia directa entre el código de la aplicación y el código de acceso a los datos. Dicha dependencia en el código de los componentes de la aplicación vuelve tediosa y dificultosa la migración desde un tipo de persistencia de datos a otra pues cuando cambia la fuente de datos el componente debe ser modificado para poder manejar la nueva fuente de datos.
La **solución** a este problema viene por usar un DAO (Data Acces Object) para abstraer y encapsular todos los accesos a la fuente de datos. DAO maneja la conexión con la fuente de datos para obtener y almacenar los datos.
EL DAO implementa los mecanismos de acceso necesarios para trabajar con la fuente de datos, esta fuente de datos puede ser un almacén de persistencia de cualquier tipo como por ejemplo un RDBMS, un servicio externo, un repositorio o incluso archivos xml. El componente del negocio (aquel que trabaja con la lógica del negocio) que se apoya en el DAO accede a la interfaz simplificada que este brinda para sus clientes. El DAO oculta completamente los detalles de la implementación de la fuente de datos a sus clientes y como la interfaz expuesta a los clientes del DAO no cambia cuando cambia la implementación de la fuente de datos este patrón permite que el DAO se adapte a diferentes esquemas de almacenamiento sin afectar sus clientes o

componentes del negocio. Esencialmente el DAO actúa como un adaptador entre el componente y la estructura de la fuente de datos.
La figura 3.1 muestra un diagrama de clases que muestras las relaciones en el patrón DAO.

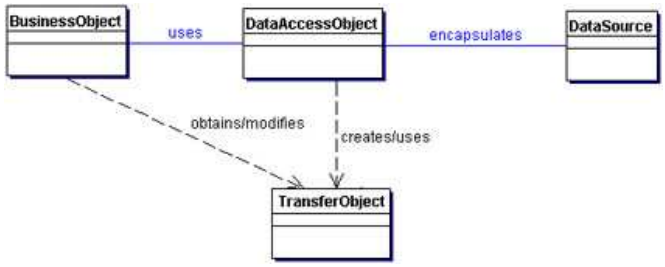


Figura 3.1: Data Acces Object

Para finalizar se listaran algunas de las consecuencias del uso de este patrón:

- Provee transparencia. El acceso a la fuente de datos es transparente pues los detalles de la implementación están ocultos en el DAO.
- Facilita la migración.
- Reduce la complejidad del código en los objetos que manejan la lógica del negocio.
- Centraliza todo el acceso a los datos en una capa separada.
- Agrega una capa extra, la cual debe ser diseñada e implementada para beneficiarse del uso de este patrón.
- Necesita diseño de jerarquía de clases, que implica otro esfuerzo extra.

3.3. La solución propuesta: *JDBGM*

Después de conocer el anterior patrón se puede inferir que el problema al que apunta este trabajo es común y a veces inevitable dependiendo de las necesidades especificas del sistema con el que se esté tratando, entonces por que desarrollar este proyecto? Pues para aquellos desarrollos donde el patrón sea aplicable donde, por ejemplo, los tiempos de respuesta no sean un factor critico recordando que DAO agrega una capa extra entre los datos y la lógica del negocio lo que en el fondo significa un tiempo de acceso un poco mayor comparado a un acceso directo, otro caso donde la solución puede ser aplicable es cuando se debe trabajar con diferentes fuentes de datos al mismo tiempo para por ejemplo persistir datos en diferentes motores al mismo tiempo. Tema aparte es que no se estará contemplando completamente la especificación de DAO si no que se lo estará limitando a bases de datos relacionales. Entonces con las aclaraciones dadas, a continuación se detallara de que trata el proyecto.
Para trabajar con *DBMS* Java provee JDBC[13] que según su documentación provee un medio para acceder virtualmente a cualquier tipo de datos tabulares. JDBC es usualmente usado para enviar comandos SQL directamente hacia el motor, aunque en realidad fue diseñado para ser la base sobre la cual se construyen herramientas e interfaces alternativas más "amigables con el programador" en el sentido que se puede construir un API más entendible o conveniente para un proyecto en particular que en el fondo es "traducida" a *JDBC*. Como en esté proyecto se pretende ocultar el acceso explicito a una base de datos no se puede usar directamente *JDBC* pues al hacerlo se necesita explicitar la conexión con el motor de la base de datos desde el componente en donde se esté realizando la persistencia de los datos, por otro lado las sentencias SQL deben ser pasadas como cadenas de texto lo que como ya fue expuesto presenta cierta dependencia con un motor en particular al estar tratando con tipos de datos inmutables (la sentencia SQL como *string*) que deben ser cambiados cuando se quiera alterar la sentencia.
Entonces haciendo uso de la idea de DAO se puede expresar lo siguiente sobre *JDBGM*: "Un adaptador entre la capa (o componentes) del negocio y la base de datos relacional subyacente que oculta los detalles específicos sobre el uso de un *DBMS* en particular", también se puede decir que *JDBGM* sera, en un diseño en capaz, parte esencial de la capa de acceso a datos. Se presentan las siguientes características básicas para el proyecto:

- Debe encargarse de gestionar la conexión hacia la base de datos.
- Debe Eliminar o limitar la dependencia del uso de los dialectos SQL.
- Debe proveer una estructura fácilmente ampliable pues inicialmente el proyecto tendrá un soporte limitado en cuanto motores de base de datos.

El objetivo primario de *JDBGM* es ofrecer un medio accesible para independizarse de el uso de un motor en particular y como consecuencia de ello se promueve el uso de una estructura ampliamente probada (basado en DAO). Hay algunos aspectos que no se pueden evitar, por ejemplo si queremos migrar de motor de base de datos es necesario que anteriormente la base se haya migrado al nuevo motor pues con lo que esta lidiando el proyecto es con el acceso a la base de datos no con la administración de la base de datos, otro aspecto es que para poder acceder a la base de datos es necesario conocer su URI[2] y nombre de usuario y contraseña usados para acceder al motor. En el capitulo siguiente se dará una descripción más detallada de los requisitos necesarios para cumplir con los objetivos que persigue el proyecto.

[2] *Uniform Resource Identifier*

Capítulo 4

Especificación

En este capitulo se establecerán los requisitos necesarios para cumplir con las necesidades que se presentaron anteriormente, para ello se establecerá que medidas se deben tener en cuenta a la hora de establecer el API que permitirá la comunicación con el motor de base de datos y además que se requiere para que el código escrito sea independiente del motor en particular que se este por utilizar.

4.1. Contexto

El proyecto como ya se comento se desarrollara sobre el lenguaje Java e inicialmente se dará soporte para tres motores de base de datos, las versiones usadas de cada una de ellas para el desarrollo del proyecto son las siguientes:

1. Java SE 6 o versión superior.
2. PostgreSQL 8.4
3. MySQL 5.1
4. Sqlite 3

En este capitulo se pretenden declarar los requerimientos generales del proyecto para cumplir con su objetivo y el por que de los mismos. Básicamente hay dos tópicos diferentes a tratar: primero se definirá como se ocultara el acceso a la base de datos o en otras palabras definir que funciones debe proveer la interfaz que ocultara el uso de JDBC; en segundo lugar esta definir como se trataran las sentencias SQL para que no sean "dialecto dependientes".

4.2. Ocultando el acceso a el motor

Cuando se habla de ocultar el acceso a el motor se esta haciendo referencia a que la API proporcionada por este proyecto debe en primer termino ocultar el uso de JDBC, en el sentido de que no se hace uso explicito de los métodos que este ofrece, en segundo lugar se refiere a que los métodos proporcionados por la API aquí desarrollada no deben ser "motor dependiente" para evitar que el código escrito con ella también lo sea. Para manejar el acceso al motor de base de datos *JDBGM* debe proveer las siguientes funcionalidades mínimas que a su vez puede que se desglosen en más de una función (o método):

1. Crear una conexión con el motor: es decir se debe encargar de obtener una conexión al motor y ponerla a disponibilidad de los métodos definidos, verificando que cuando se la necesite esta siga estando "viva".
2. Cerrar la conexión con el motor: poder cerrar la conexión con el motor para liberar recursos de la misma, una conexión no usada sin cerrar es un desperdicio de recursos.
3. Enviar sentencias SQL al motor: el modo de interactuar con el motor es mediante el envío de sentencias SQL, este punto se lo discutirá más ampliamente en la sección siguiente.

4. Poder enviar transacciones: una transacción consiste en un conjunto de sentencias que deben ejecutarse en conjunto, una vez finalizadas se guardan los cambios (*commit*) o se rechazan (*rollback*) si sucediere algún error en cualquiera de las sentencias que se pretenden ejecutar como un conjunto.

5. Manejar los errores que se puedan producir: se pueden dar errores en la sintaxis de las sentencias SQL como así también errores en la lógica de las sentencias que se formaron, como por ejemplo equivocarse en el nombre de una tabla o referirse a una columna inexistente, además pueden existir otros tipos de errores que no están directamente relacionados con el *DBMS*, por ejemplo que la conexión de red necesaria para comunicarse con un motor que se encuentra en un servidor externo falle. Estos errores deben ser correctamente atrapados para que el programador evite que el programa que esta desarrollando simplemente se cuelgue (deje de funcionar).

6. Devolver resultados de las consultas, por ejemplo para una consulta del tipo `SELECT` es necesario devolver el conjunto de datos que resultaron de la consulta, para las otras sentencias DML como `INSERT` puede resultar necesario conocer la cantidad de columnas que fueron afectadas.

7. Manejar la creación de los objetos de modo que se instancien desde la clase adecuada dependiendo de el motor que se esta por utilizar. En primer instancia esto puede parecer algo trivial pero es en realidad importante por que libera al programador de la necesidad de decidir que clase debe instanciar y hace a el código independiente de una implementación particular de las interfaces definidas.

4.3. Eliminando dialectos

En el capitulo anterior se comento que parte del desafío de este proyecto consiste en lidiar con los diferentes dialectos utilizados por cada motor en particular, por lo que a continuación se analizara como es que se pretende sortear esta dificultad.
JDBC provee la clase `Statement`[1] la cual posee los métodos `executeUpdate` y `executeQuery` los cuales permiten enviar sentencias SQL al motor de base de datos. El envío de estas sentencia se hace mediante cadenas de texto por lo que se presenta el problema de que estas cadenas "estáticas" dependen, según cuan complejas sean las sentencias, de el motor en particular que se esté usando. Es por ello que en este proyecto se opto por generar las sentencias bajo demanda, es decir se pretende que las sentencias se almacenen desglosadas de modo que su sintaxis quede expuesta y pueda luego ser volcada en una cadena de texto cuando sean requeridas por el programador. El almacenar de este modo las sentencias no asegura tampoco la independencia de los dialectos usados pero permite manejar el modo en que las sentencias serán convertidas en cadenas de textos, entonces si se define un nuevo dialecto que sea compatible con los dialectos usados por los otros motores y ajustando la estructura de datos que se va a utilizar para las sentencias a este dialecto se puede, mediante las reglas adecuadas, "traducir" la sentencia a su dialecto correspondiente.
Definir un nuevo dialecto puede parecer algo engorroso, pero en realidad este nuevo dialecto estará basado en los dialectos usados por los motores a los que se le esta dando soporte y además solo se trabajara con un subconjunto de las sentencias definidos en estos dialectos pues no es objetivo del proyecto brindar un total soporte sobre SQL si no que se pretende cubrir un subconjunto de estas que permitan realizar las operaciones usualmente conocidas como CRUD[2] que viene del ingles "*Create, Read, Update and Delete*" que puede traducirse en el uso de las sentencias SQL `INSERT, SELECT, UPDATE` y `DELETE`.
Las necesidades del proyecto quedarían satisfechas con el soporte para las sentencias listadas en el párrafo anterior pero como esta parte del proyecto se basara en una librería pre-existente que también da soporte para las sentencias `CREATE TABLE` y `ALTER TABLE` estas también serán incluidas como parte del proyecto. Un punto interesante de aclarar es que no hay que confundir la sentencia `CREATE TABLE` con la operación *create* a la que hace referencia el acrónimo CRUD

[1] Posteriormente se explicara como es que se trabaja con JDBC, por ahora interesa solo lo expuesto.
[2] Aunque a veces se cambian el significado de algunas de las siglas, usualmente la definición dada es la más usada.

pues este se refiere a crear un nuevo registro en el sistema de persistencia de datos que se esté usando, llevándolo al ámbito de los motores de base de datos relacionales se refiere a crear una fila en una tabla, mientras que `CREATE TABLE` sirve para crear tablas, las filas en las tablas se insertan mediante `INSERT`, visto esto y como no es parte del objetivo planteado para el proyecto el soporte para estas dos sentencias sera básico, dejando las puertas abiertas para que en futuras versiones el trabajo pueda ser refinado fácilmente. Por lo que las siguientes sentencias serán las reconocidas por *JDBGM*:

- `CREATE TABLE` - Permite crear tablas.
- `ALTER TABLE` - Modifica la estructura de una tablas.
- `UPDATE` - Modifica las filas de una tabla.
- `INSERT` - Inserta nuevas filas a una tabla.
- `DELETE` - Elimina filas de una tabla.
- `SELECT` - Realiza consulta sobre las tablas.

Las sentencias listadas anteriormente cubren las necesidades que se pueden presentar en la lógica de negocio del proyecto sobre el cual se esté trabajando, el uso más extensivo de SQL se da a la hora de la administración de la base de datos trabajo que usualmente es realizado por la persona que cumpla el rol de DBA (*Data Base Administrator*) el cual usa una interfaz diferente, propia a veces de el motor mismo, para trabajar directamente sobre el.
La estructura de datos que se quiere manejar, la cual quedara expresada en forma de una clase, aparte de contener los datos que hacen a la sentencia debe brindar métodos para convertir la sentencia en una cadena de texto, asignarles valores a los atributos y definir el comportamiento especifico que pueda tener la sentencia. Así se puede hacer un resumen de las características que se desea cumplan estas clases:

- Deben poder devolver una cadena de caracteres (*String*) que represente una sentencia SQL valida para el dialecto deseado.
- La sintaxis proveída para el manejo de las clases debe ser consistente entre todas ellas de modo que el aprendizaje de esta no presente mayor dificultad y debe ser lo suficientemente "expresiva" como para poder entender lo que se esta haciendo sin tener que recurrir necesariamente a la documentación, lo que tampoco quita que la documentación deba ser dejada de lado.
- Deben contemplar de manera básica que no se realicen acciones no permitidas con las sentencias, es decir respetar la sintaxis de los dialectos, aspecto que también ah de tenerse en cuenta cuando se esté definiendo el dialecto genérico para el proyecto. El manejo de estos errores debe ser mediante el uso de excepciones usando las clases hijas de `Exception` en Java.
- Deben contemplar todas las opciones disponibles para las sentencias que se están por incluir definidas en el dialecto genérico.
- Deben proveer los métodos necesarios para poblar de datos las clases.

Una vez definido el subconjunto de sentencias con las que se trabajaran es necesario definir el dialecto genérico sobre el cual se apoyara este proyecto, por lo que para cada una de las sentencias se definirá un comportamiento común, pero para que esto sea posible se deberán restringir los dialectos soportados al que menos posibilidades presente pues las diferencias entre los dialectos usados por los motores se da generalmente por que se agregan funcionalidades a las definidas en el estándar SQL y raramente se da el caso en que la diferencia se de por la falta de implementación de lo definido en el estándar. Como parte del análisis de las diferencias entre los dialectos es importante analizar también los tipos de datos que se usan en cada *DBMS*. También para ellos se debe encontrar un factor común, mas que nada por la inclusión de la sentencias `CREATE TABLE` y `ALTER TABLE` que precisan que se definan tipos de datos, por lo que a continuación se dedicara una sección entera a analizar los tipos de datos usados por cada motor.

4.3.1. Tipos de datos y sus diferencias

Al analizar los tipos de datos usados en los diferentes motores, quizá la diferencia más notoria se da en el modo en que *SQLite* maneja estos tipos. La mayoría de los motores de datos manejan un tipo de datos estático, en el que el contenedor del dato (la columna) define el tipo de dato que este podrá almacenar, o sea el tipo de dato declarado en la definición de la columna. En cambio en *SQLite* no existen los tipos de datos como tal, si no que se definen clases de almacenamiento[9] (*storage clases*) que son más genéricas que los tipos de datos usualmente definidos, por ejemplo la clase de almacenamiento `INTEGER` puede almacenar los 6 diferentes tipos de datos enteros que define *MySQL*, que en definitiva en lo único que difieren es en el espacio que ocupan para ser almacenados, por ejemplo `SMALLINT` ocupa menos bits para ser representado que el tipo `INTEGER`.
En *SQLite* el tipo de dato de un valor esta asociado a el dato en si y no a su contenedor, pero aun así existe compatibilidad con los otros motores en el sentido de que *SQLite* puede interpretar adecuadamente sentencias SQL validas que usen los clásicos tipos de datos estáticos. Para brindar esta compatibilidad se define el concepto de afinidad de tipo para las columnas que viene a ser el tipo de dato recomendado para los datos almacenados en esa columna, así existen una serie de reglas para definir la afinidad de la columna en base a el nombre del tipo de dato declarado para esa columna, pudiendo ser este nombre cualquiera. Aun así la afinidad de la columna **recomienda** un tipo de dato no lo exige para esa columna, por lo que en *SQLite* una columna puede almacenar cualquiera de los tipos de datos definidos (storage class para *SQLite*). Observado esto se puede decir que al ser *SQLite* tan flexible los tipos de datos que se usaran estarán definidos en base a *MySQL* y *PostgreSQL*.
JDBGM debe proveer una interfaz única en la que se establezcan los tipos de datos disponibles para usar, estos tipos de datos deben ser elegidos en base a las compatibilidades entre los diferentes motores que soporta para lograr un uso genérico sobre cualquiera de los motores. En principio se podrían elegir todos los tipos de datos definidos en los motores, uniendo a aquellos que sean equivalentes en un solo tipo y dejando que después *JDBGM* se encargue de mapear los datos a los soportados por un motor en concreto, recordando que algunos de los datos declarados no son directamente soportados por todos los *DBMS*. Pero esto podría acarrear confusiones pues el tipo de dato entero más chico (el que ocupa menos espacio en disco para representarse) en *MySQL* [3] es `TINYINT` que ocupa solo 1 byte, en cambio en *PostgreSQL* [6] el más chico es `SMALLINT` que ocupa 2 bytes entonces en el caso de que en los tipos de datos especificados para *JDBGM* se encuentren estos dos no habría problemas al usarlos pues si se llegara a especificar una columna con tipo de dato `TINYINT` mientras se trabaja sobre una base de datos con *PostgreSQL* no habría mas que mapear internamente el tipo de dato a `SMALLINT` que al ser más grande tranquilamente lo puede contener, pero en esta situación el programador podría no enterarse de que en realidad la columna que el esta usando ocupa 2 bytes por dato lo que en casos en que la capacidad de almacenamiento sea un tema sensible podría llevar a serias confusiones, entonces lo mejor es que en la especificación de *JDBGM* solo se incluyan los tipos que no lleven a este tipo de confusiones. A continuación se analizan los tipos de datos y sus congruencias, empezando por los datos numéricos del tipo entero mediante un cuadro comparativo en los que los tipos puestos en la misma linea son congruentes entre si, salvo que se indique lo contrario:

SQLite	*MySQL*	*PostgreSQL*
INTEGER	TINYINT	SMALLINT
	SMALLINT	
	MEDIUMINT	INTEGER
	INT	BIGINT
	BIGINT	

Tabla 4.1: Comparación de los tipos de datos enteros

Como se puede ver la clase de almacenamiento `INTEGER` de *SQLite* es capaz de contener todos los tipos de datos declarados en los otros dos motores, en *MySQL* se definen más tipos de datos enteros que difieren únicamente en el espacio que utiliza para almacenarse en disco, el más chico usa solo 1 Byte para almacenarse, mientras que el más grande usa 8 Bytes. En *SQLite* el dato es

almacenado en disco con la menor cantidad posible de Bytes pero tan pronto como es cargado en memoria es convertido a un entero de 8 bytes con signo[3]. Lo que interesa para el proyecto es elegir un subconjunto de estos tipos que sean aplicables a los tres motores, en este caso es *PostgreSQL* el que define menos tipos de datos, teniendo en cuenta que *SQLite* presenta un tipo de dato que envuelve a todos los demás, por lo que se tomaran esos tipos como los elegidos para *JDBGM*.

SQLite	*MySQL*	*PostgreSQL*
REAL	FLOAT REAL DOUBLE PRECISION	 REAL DOUBLE PRECISION

Tabla 4.2: Comparación de los tipos de datos reales aproximados

En el caso de los tipos de números reales se tienen básicamente dos clases de tipos, los que guardan los datos de forma exacta y otros no exactos que se guardan en el formato punto flotante. Para el caso de los tipos de datos con punto flotante se tomaran los tipos definidos para *PostgreSQL* como los elegidos para *JDBGM*, el tipo `FLOAT` definido en *MySQL* es similar a `REAL`, y nuevamente el tipo `REAL` de *SQLite* es capaz de contener todos los otros tipos.

SQLite	*MySQL*	*PostgreSQL*
NUMERIC	NUMERIC DECIMAL DATETIME TIMESTAMP DATE TIME	NUMERIC DECIMAL TIMESTAMP DATE TIME

Tabla 4.3: Comparación de los tipos de datos reales exactos y fechas

Para analizar el caso de los tipos de datos reales con almacenamiento exacto, se deberán analizar los tipos de datos de fecha y hora, pues aunque parezca un poco confuso *SQLite* no tiene un tipo de dato especifico para estos tipos, pero si posee una afinidad de columnas para ellos que se denomina `NUMERIC`, este afinidad lo que hace es decidir cual es el mejor modo de almacenar el dato, pudiendo ser cualquier de las clases `NULL, INTEGER, REAL, TEXT` o `BLOB`. Para los tipos `NUMERIC` Y `DECIMAL` que son para almacenamiento exacto de números reales no se tiene una clase especifica, pero lo que si se puede hacer es asignarle el tipo (afinidad de columna) `NUMERIC` que se encargara de que el motor guarde el dato en el formato que le parezca más adecuado. Para los datos del tipo fecha y hora se tiene bastante similitud en *MySQL* y *PostgreSQL* salvo que en *MySQL* esta `DATETIME` que en lo único que difiere de `TIMESTAMP` es en el rango de fechas que soporta. En *SQLite* no se tienen tipos de datos para fecha y hora, pero se pueden guardar las fechas como cadenas de texto o numero, por lo que si se le da la afinidad de columna `NUMERIC` el motor resolverá cual es el mejor tipo de datos para almacenar la facha u hora según el formato que se esté utilizando.

SQLite	*MySQL*	*PostgreSQL*
TEXT	CHAR(n) VARCHAR(n) TINYTEXT TEXT MEDIUMTEXT LONGTEXT	CHAR(n) VARCHAR(n) TEXT

Tabla 4.4: Comparación de los tipos de datos cadenas de texto

[3]Cuando al representarse un numero se usa el signo (±) se achica el rango de numero que se puede representar

Para el caso de los tipos de datos cadenas de texto se tiene como siempre *SQLite* con una clase de almacenamiento bastante genérica que cubre todos los tipos de datos definidos en los demás motores. La diferencia que se encuentra con los otros dos motores es que el tipo `TEXT` que en *MySQL* tiene 4 variantes que varían en el limite de almacenamiento, siendo el ultimo sin limite (`LONGTEXT`) en cambio en *PostgreSQL* existe solo `TEXT` y que es para almacenamiento de texto sin limite, así que como antes se tomaran los tipos de datos que define *PostgreSQL* como los elegidos para *crossdb*.

SQLite	*MySQL*	*PostgreSQL*
BLOB	BINARY VARBINARY TINYBLOB BLOB MEDIUMBLOB LONGBLOB	bytea

Tabla 4.5: Comparación de los tipos de datos binarios

En el caso de los tipos de datos binarios se tienen en *MySQL* una gran cantidad de variantes de `BLOB` que varían en tamaño máximo permitido y otros que son de tamaño variable. En cambio para *SQLite* como siempre se tiene una única clase "genérica" bastante flexible y en *PostgreSQL* se tiene un solo tipo que puede almacenar datos binarios que difiere en el tamaño máximo que es capaz de contener. Como la principal diferencia entre los motores es el tamaño máximo se optara por elegir un único tipo, el `BLOB`.

SQLite	*MySQL*	*PostgreSQL*
1 o 0	1 o 0	BOOLEAN
NUMERIC	NUMERIC	MONETARY

Tabla 4.6: Comparación de tipos de datos varios

Los últimos tipos que quedan por revisar son los booleanos que en lo que difieren es en el modo en el que se almacenan, por ejemplo en *MySQL* mediante un valor entero y en *PostgreSQL* mediante un carácter. Pero no se generan mayores complicaciones si se utiliza un único tipo para representarlos a todos. Para los valores monetarios por lo general se exigen valores exactos para no generar errores en los cálculos que se pudieran hacer sobre estos numero, para ellos *PostgreSQL* define un tipo especial, `MONETARY`, pero que puede ser fácilmente reemplazado en su función especifica por `NUMERIC` así que se descartara el tipo especifico de *PostgreSQL*, decantándose por el uso de `NUMERIC`.

4.3.2. Funciones

Las funciones que presentan los diferentes motores son variadas, en la documentación de *PostgreSQL* se puede leer[7] la siguiente aclaración:

> *If you are concerned about portability then note that most of the functions and operators described in this chapter, with the exception of the most trivial arithmetic and comparison operators and some explicitly marked functions, are not specified by the SQL standard. Some of this extended functionality is present in other SQL database management systems, and in many cases this functionality is compatible and consistent between the various implementations.*

Que en resumen explica que la mayoría de las funciones ofrecidas por *PostgreSQL* con excepción de algunas pocas no forman parte de ninguno de los estándares SQL por lo que no hay que esperar compatibilidad entre los diferentes motores aunque existen algunas funciones comunes que se pueden encontrar entre los diferentes motores. Así que a continuación se buscaran funciones similares entre los diferentes motores a los que se les esta dando soporte en este proyecto,

para lo cual se empezara con una tabla comparativa en la que las funciones equivalentes de la misma linea tiene la palabra `igual` para denotar tal situación:

SQLite	*MySQL*	*PostgreSQL*
abs(X)	igual	igual
changes()	ROW_COUNT()	–
coalesce(X,Y,...)	igual	igual
glob(X,Y)	–	–
ifnull(X,Y)	igual	mediante coalesce(x,y)
hex(X)	igual	–
length(X)	LENGTH()	char_length(string)
like(X,Y,Z)	x LIKE y [ESCAPE z]	igual mysql
lower(X)	igual	igual
ltrim(X)	igual)	igual
max(X,Y,...)	–	–
min(X,Y,...)	–	–
nullif(X,Y)	igual	igual
quote(X)	igual	quote_literal(string text)
random()integer	RAND()float	random() float
randomblob(N)	–	–
replace(X,Y,Z)	igual	igual
round(X), round(X,Y)	igual	igual
rtrim(X)	igual	igual
substr(X,Y,Z), substr(X,Y)	igual	igual
trim(X), trim(X,Y)	igual	igual
upper(X)	igual	igual

Tabla 4.7: Comparación de las funciones core de *SQLite*

En la tabla 4.7 están listadas las funciones que son parte de las funciones básicas[11] o *core functions* de *SQLite* que como se puede ver no son muchas, igual se obviaron algunas funciones especificas de *SQLite* aunque las aquí presentadas son una gran parte de las mismas. Lo que se pretende mostrar con esta tabla es la equivalencia de estas funciones con las que presentan los otros motores para lo cual se estudio la documentación proveída por *PostgreSQL* [7] y por *MySQL* [4] y se encontraron las equivalencias mostradas.
Tal como fue señalado existe cierta universalidad entre las funciones que ofrecen los motores como por ejemplo ocurre con la función `abs()` que se usa de igual manera en todos los motores y diferencias como con la función `ifnull()` que en *PostgreSQL* no existe pero se tiene un resultado igual con la función `coalesce()`[4]. Otras diferencias también se pueden ver en funciones que no tienen equivalente en los otros motores como ocurre en el caso de la función `randomblob(N)` de *SQLite*, de todos modos es posible definir nuevas funciones en cualquiera de los tres motores con lo que se podría llegar a una total compatibilidad, pero esa no es la intención de este proyecto, lo que se pretende es buscar las compatibilidades existentes entre los motores por lo que *JDBGM* tendrá disponible un conjunto de funciones que son reconocidas por los tres motores de manera predeterminada.
De las funciones ofrecidas por este proyecto solo se ofrecerá el nombre de la función quedando a cargo del usuario la correcta utilización de las mismas teniendo en cuenta que a pesar de tener los mismos nombres la implementación de una función en un motor puede ser más permisiva que en otro en cuanto a los parámetros aceptados y las valores que devuelven, así que de existir un error este sera reportado cuando se intente usar la sentencia que se formo haciendo uso de cualquiera de estas funciones. *SQLite* posee otras funciones disponibles por defecto que están documentadas en dos capítulos aparte, por lo que a continuación se presentaran dos tablas que muestran las equivalencias de las mismas en los otros motores.
Entre las funciones agregadas de *SQLite* se encuentra que casi todas son compatibles excepto por la función `group_concat` que presenta una sintaxis algo diferente pero funciona igual en

[4] *SQLite* aclara en su documentación que `ifnull(x,y)` es equivalente a `coalesce(x,y)`.

SQLite	*MySQL*	*PostgreSQL*
avg(X)	igual	igual
count(X)	igual	igual
count(*)	igual	igual
group_concat (X[,Y])	group_concat (X [separator Y])	array_to_string (array_agg(X), Y)
sum(), total()	igual	igual
max()	igual	igual
min()	igual	igual

Tabla 4.8: Comparación de las funciones agregadas de *SQLite*

MySQL y se puede obtener el mismo resultado en *PostgreSQL* mediante el uso de dos funciones distintas. La otra función que no existe en los demás motores es `total()` que es similar a `sum()` pero trabaja de manera diferente con los valores `null` y además a diferencia de `sum` devuelve siempre un `float` como resultado. Cabe aclarar que `max()` y `min()` aparecen también en el *core* de *SQLite* pero estas son diferentes pues toman distintos parámetros y se comportan de manera diferente.

SQLite	*MySQL*	*PostgreSQL*
date()	igual con now()	current_date or date()
datetime()	timestamp() , now()	to_timestamp, current_timestamp
time()	time() con now()	"time"() or now()::time
julianday()	-	-
strftime()	-	varios

Tabla 4.9: Comparación de las funciones Fecha y Hora de *SQLite*

La tabla 4.9 muestra las funciones que tiene *SQLite* para el manejo de fechas y horas y sus equivalentes de existir en los otros dos motores, las funciones que define *SQLite* tienden a estar sobrecargadas (o admiten valores por defecto para los parámetros como en PHP) por ejemplo la función `date()` de usársela sin ningún parámetro devuelve la fecha (DATE) actual en el formato `'YYYY-MM-DD'` en cambio si se le pasa una cadena de texto equivalente a un `TIMESTAMP` la misma función devuelve la porción de la fecha de la cadena de texto, para lograr esto en *MySQL* y *PostgreSQL* se usan dos funciones para obtener la misma funcionalidad por ejemplo si en *SQLite* se quiere obtener la fecha (DATE) actual se debe usar `date(now())` y `date(D)` funcionaria como en *SQLite* donde `D` es una fecha completa (TIMESTAMP). Para `juliandays()` y `strftime()` se pueden obtener resultados parecidos mediante el uso de más de una función o realizando algunos cálculos, por lo que no existe una función equivalente para las mismas a menos que se la defina o se usen más de una para obtener el mismo resultado.

Como se dijo previamente el uso de las funciones puede llegar a comprometer la portabilidad y salvo por aquellas más comunes se pueden obtener resultados diferentes ante las mismas entradas por lo que *JDBGM* debe ofrecer las funciones que tengan su equivalente en los otros motores y no debe chequear sintaxis pues esto ya lo hará el propio motor, el realizar esta comprobación seria una redundancia y un aumento de procesamiento con su correspondiente sobrecarga totalmente evitable. Por lo que el soporte para funciones se limitara a ofrecer una lista de funciones permitidas pero dejando la puerta abierta a el uso de otras a riesgo propio del programador.

4.3.3. Sentencias SQL

En las siguientes secciones se pasa a analizar por separado cada una de las sentencias que se están por incluir en el dialecto genérico, para ello se compararan las implementaciones que se hicieron sobre cada uno de los motores sobre los que se da soporte.

4.3.3.1. Especificación de CREATE TABLE

Esta sentencia es usada para crear tablas en una base de datos relacional, un resumen de su sintaxis como se la define en el estándar SQL es la siguiente:

```
CREATE TABLE <table name> (
{ <column name> [ <column type> ][ PRYMARY KEY ][ REFERENCES <foreign table> ] }...
)
[ PRIMARY KEY <indexed columns> ]
[ FOREIGN KEY <columns and referenced table> ]
```

Se obviaron las definiciones de algunas de las opciones de `CREATE TABLE` puesto que escapan al alcance del proyecto el cual esta limitado por los DBMS a los que tiene que soportar, además hay que tener en cuenta algunas salvedades, como que si se especifica una columna como `PRIMARY KEY` dentro de la definición de la columna no se puede especificar la clausula `PRIMARY KEY` fuera de los paréntesis que encierran a las definiciones de las columnas. más específicamente el análisis puede empezar con *SQLite* el cual, sin restarle importancia, es el que menos características implementa debido a su "pequeñez" por lo cual se pasa a estudiar `CREATE TABLE` tal como la entiende *SQLite* [10][5] y comparando las diferencias con *MySQL* [5] y *PostgreSQL* [8]

```
CREATE [ TEMP | TEMPORARY ] TABLE <database name> <dot> <table name>
[ IF NOT EXISTS ] <table contents source>

<table contents source> ::=
  <left paren> <table element> [ { <comma> <table element> }... ] <right paren>
| AS <select stmt>
```

Los parámetros `TEMPORARY` y `TEMP` se usan indistintamente para crear una tabla temporaria en *SQLite*, lo cual es soportados por los 3 DBMS con la salvedad que en *MySQL* solo se acepta la palabra reservada `TEMPORARY`. El parámetro `IF NOT EXISTS` indica que de existir una tabla del mismo nombre la sentencia `CREATE TABLE` no tiene efecto sobre la base de datos, este parámetro no es soportado por *PostgreSQL* por lo que sera omitido. Por ultimo hay que tener en cuenta que al ser *SQLite* un motor "serverless" siempre se esta trabajando o sobre la base de dato principal o sobre una temporaria por lo que este para el parámetro `<database name>` solo acepta `"main"` ó `"temp"` de modo que en un principio el nombre de la base de dato no se tomara en cuenta y cualquier tabla que se cree sera en la base de dato actual. La restricción `AS <select stmt>` funciona de manera similar para los tres motores de base de datos por lo que también sera incluida, en lo único que varían es en como son reconocidos los tipos de datos y además que debido a *SQLite* los nombres de las columnas agregadas deben ser los mismo que los de la sentencia `SELECT`, por ultimo vale aclarar que `<select stmt>` debe ser una sentencia `SELECT` valida y correspondiente a el mismo motor.

```
<table element> ::=
  <column definition>
| <table constraint definition>

<column definition> ::=
<column name> [ <data type> | <domain name> ]
[ DEFAULT <default option> ]
[ <column constraint definition>... ]
[ COLLATE <colation name> ]

<default option>::=
  <signed number>
| <literal value>
| <left paren> <expr> <left paren>

<column constraint definition> ::=
[ CONSTRAINT <name> ] <column constraint>
```

[5] para ver la sintaxis usada ir a el apéndice A

```
<column constraint> ::=
  NOT NULL <conflict cause>
| UNIQUE <conflict cause>
| PRIMARY KEY [ ASC | DESC ] <conflict cause> [ AUTOINCREMENT ]
| <references specification>
| CHECK <left paren> <expr> <right paren>
```

La siguiente sección a estudiar es la definición de columnas y restricciones de columna que se diferencian de las restricciones de tabla en que las primeras se definen junta a la definición de columna y solo se pueden hacer sobre una única columna, en cambio las restricciones de tabla se hacen después de las definiciones de columnas y estas pueden referirse a un grupo de columnas. Como primer diferencia se observan los tipos de datos que tienen diferentes nombres y significados, pero esto ya fue tratado en la sección 4.3.1 de la página 14. La clausula `DEFAULT` en este caso esta limitada por *MySQL* que solo acepta valores constantes, es decir no se podría usar funciones como `NOW()` con la excepción de que se puede poner `CURRENT_TIMESTAMP` como `DEFAULT` para una columna del tipo `TIMESTAMP`.
El parámetro `COLLATE` también presenta diferencias, pero es en este caso es *PostgreSQL* el que limita ya que no acepta `COLLATE` en la definición de columna por lo que no sera adoptada en la especificación de *JDBGM*. En *PostgreSQL* se puede especificar el "collate" a usar sobre todas las tablas en la creación de la base de datos. `UNIQUE` y `PRIMARY KEY` como restricción de columna no aceptan ningún modificador en *MySQL*, por lo tanto se obviara el modificador `<conflict cause>` de *SQLite* y otros que acepta *PostgreSQL*. El parametro `NOT NULL` tampoco acepta ningún modificador en *MySQL*, su contra parte el parámetro `NULL` puede ser omitido ya que de no agregarse se toma por defecto que la columna puede aceptar valores nulos. Por ultimo hay que señalar que *MySQL* si bien implementa `CHEK <expre>` este no es reconocido en la definición de columna.
En *SQLite* y *PostgreSQL* a las restricciones de columna se les puede especificar un nombre con `CONSTRAINT <name>` pero esto no es soportado en *MySQL*. Una ultima diferencia se puede observar en el parámetro `AUTOINCREMENT` (`AUTO_INCREMENT` en *MySQL*) que solo se puede especificar cuando una columna es `PK` y del tipo `INTEGER`, en *MySQL* se puede hacer incluso sobre columnas que no sean `PK`, en *PostgreSQL* este parámetro no existe, al menos no en la version sobre la que se esta trabajando. Aun así es posible sortear este problema en *PostgreSQL* pues es posible usar la palabra clave `SERIAL` la cual es un atajo para crear una secuencia y asignársela a la columna, en este caso se usara para emular el funcionamiento de la clausula anterior y con *MySQL* lo único que hace falta es limitar esta clausula como lo hace *SQLite*.

```
<table constraint definition> ::=
[ CONSTRAINT <name> ] <table constraint>

<table constraint> ::=
  [ UNIQUE | PRIMARY KEY ] <left paren> <unique column list> <right paren> <conflict cause>
| <referential constraint definition>
| CHECK <left paren> <expr> <right paren>

<unique column list> ::= <column name list>

<referential constraint definition> ::=
FOREIGN KEY <left paren> <referencing columns> <right paren>
<references specification>
```

De las restricciones de tabla la única a la que no se le podrá especificar un nombre de restricción con `CONSTRAINT <name>` es la restricción `CHECK (expr)`. Los parámetros `UNIQUE` y `PRIMARY KEY` que se especifican como restricción de tabla solo comparten en común entre la especificación de los 3 DBMS que encierran entre paréntesis a las columnas que se indican con dicha restricción. Resta por analizar la especificación de clave foránea (`FK`):

```
<references specification> ::=
REFERENCES <referenced table and columns>
[ MATCH { FULL | PARTIAL | SIMPLE } ] [ <referential triggered action> ]
[ <constraint characteristics> ]
```

```
<referenced table and columns> ::=
<table name> [ <left paren> <reference column list> <right paren> ]

<reference column list> ::= <column name list>

<referential triggered action> ::=
  <update rule> [ <delete rule> ]
| <delete rule> [ <update rule> ]

<update rule> ::= ON UPDATE <referential action>

<delete rule> ::= ON DELETE <referential action>

<referential action> ::=
  CASCADE
| SET NULL
| SET DEFAULT
| RESTRICT
| NO ACTION

 <constraint characteristics> ::=
[ NOT ] DEFERRABLE [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Los parámetros `MATCH { FULL | PARTIAL | SIMPLE }` son reconocidos en los 3 DBMS, pero en *SQLite* no tienen ningún efecto, es decir que estos se pueden especificar pero siempre se tendrá que la `FK` se procesara con `MATCH SIMPLE`, con lo que de momento sera parte de la especificación pero hay que tener siempre en cuenta que *SQLite* lo procesara de manera diferente. Por ultimo quedan dos diferencias, la primera es que la opción `SET DEFAULT` de `<referential action>` no es soportada por *MySQL*; segundo que `<constraint characteristics>` no es soportada por *MySQL*. Así que finalmente se tiene la siguiente sintaxis de `CREATE TABLE` para el proyecto:

sintaxis para CREATE TABLE

```
CREATE [ TEMPORARY ] TABLE <database name> <dot> <table name>
<table contents source>

<table contents source> ::=
  <left paren> <table element> [ { <comma> <table element> }... ] <right paren>
| AS <select stmt>

<table element> ::=
  <column definition>
| <table constraint definition>

<column definition> ::=
<column name> [ <data type> | <domain name> ]
[ DEFAULT <constant> ]
[ <column constraint>... ]

<column constraint> ::=
  NOT NULL
| UNIQUE
| PRIMARY KEY [AUTO_INCREMENT]
| <references specification>

<table constraint definition> ::=
[ CONSTRAINT <name> ] <table constraint>

<table constraint> ::=
  [ UNIQUE | PRIMARY KEY ] <left paren> <unique column list> <right paren>
| <referential constraint definition>
| CHECK <left paren> <expr> <right paren>
```

```
<unique column list> ::= <column name list>

<referential constraint definition> ::=
FOREIGN KEY <left paren> <referencing columns> <right paren>
<references specification>

<references specification> ::=
REFERENCES <referenced table and columns>
[ MATCH { FULL | PARTIAL | SIMPLE } ] [ <referential triggered action> ]

<referenced table and columns> ::=
<table name> [ <left paren> <reference column list> <right paren> ]

<reference column list> ::= <column name list>

<referential triggered action> ::=
  <update rule> [ <delete rule> ]
| <delete rule> [ <update rule> ]

<update rule> ::= ON UPDATE <referential action>

<delete rule> ::= ON DELETE <referential action>

<referential action> ::=
  CASCADE
| SET NULL
| RESTRICT
| NO ACTION
```

4.3.3.2. Especificación de UPDATE

Esta sentencia es usada para actualizar los datos de una tabla, es decir para modificar valores existentes de filas. El análisis de la siguiente sintaxis corresponde a como es entendida la sentencia `UPDATE` por *SQLite* [10].

```
<update statement> ::=
UPDATE [ OR <or option> ] <target table>
SET <set clause list>
[ WHERE <search condition> ]

<or option> ::=
  ROLLBACK
| ABORT
| REPLACE
| FAIL
| IGNORE
```

En este punto se encuentra la primer diferencia con los demás DBMS, el elemento `<or option>` que permite especificar que acción tomar si no es posible realizar la acción de `UPDATE` sobre la tabla, es un añadido de *SQLite* que no esta presente en las especificaciones de los otros dos DBMS y tampoco aparece en la especificación de SQL.

```
<target table> ::=
[ <database name> <dot> ] <table name>
[ INDEXED BY <index name> ]
[ NOT INDEXED ]

<set clause list> ::= <set clause> [ { <comma> <set clause> }... ]

<set clause> ::=
```

```
<column name> <equals operator> <expr>

-----Solo disponible si se habilito durante la compilación---------

[ ORDER BY <ordering-term> ]
[ LIMIT <expr> ]
```

Las elementos `[ INDEXED BY <index name> ]` y `[ NOT INDEXED ]` no son reconocidos por *MySQL* y *PostgreSQL* por lo que serán ignorados en la especificación del proyecto. Lo demás es de idéntica sintaxis salvo por las opciones `ORDER BY` y `LIMIT` que solo están disponibles en *SQLite* si durante la compilación del motor se habilito la opción `SQLITE_ENABLE_UPDATE_DELETE_LIMIT` la cual por defecto esta desactivada y además *PostgreSQL* no las reconoce por lo que quedan descartadas. La especificación final de *JDBGM* queda como sigue.

sintaxis para UPDATE

```
<update statement> ::=
UPDATE <target table>
SET <set clause list>
[ WHERE <search condition> ]

<target table> ::=
[ <database name> <dot> ] <table name>

<set clause list> ::= <set clause> [ { <comma> <set clause> }... ]

<set clause> ::=
<column name> <equals operator> <expr>
```

4.3.3.3. Especificación de INSERT

La sintaxis que se analiza a continuación corresponde a la sentencia `INSERT` que sirve para insertar filas nuevas en una tabla dada, felizmente la sentencia de `INSERT` entendida por *SQLite* solo difiere en dos aspecto de las otras, primero es que esta presenta el elemento `<or option>` que no es soportado por los otros DBMS, segundo el elemento `REPLACE` que tampoco es reconocido por los demás motores. Además es bueno aclarar que *SQLite* esta restringiendo las capacidades de los otros motores, como por ejemplo en *MySQL* existe el elemento `ON DUPLICATE KEY UPDATE` el cual sirve para resolver conflictos cuando una inserción en una tabla provoca que alguna columna que deba ser única (`PK` ó `UNIQUE`) presente un valor duplicado. Así que sin más la siguiente es la sintaxis elegida y compatible con los tres motores:

sintaxis para INSERT

```
<insert statement> ::=
INSERT INTO <insertion target> <insert columns and source>

<insertion target> ::= [ <database name> <dot> ] <table name>

<insert columns and source> ::=
  <from subquery>
| <from constructor>
| DEFAULT VALUES

<from subquery> ::=
[ <left paren> <insert column list> <right paren> ]
<query expression>

<from constructor> ::=
[ <left paren> <insert column list> <right paren> ]
VALUES <left paren> <exp> [ { <comma> <exp> }... ] <right paren>

<insert column list> ::= <column name list>
```

4.3.3.4. Especificación de ALTER TABLE

`ALTER TABLE` sirve en SQL para alterar la estructura de las tablas, en este caso *SQLite* es el motor que esta limitando a los demás motores pues la sintaxis que soporta esta muy limitada, las únicas acciones que soportan son renombrar las tablas y agregar nuevas columnas a una tabla. Por lo que la sintaxis final elegida es la misma que soporta *SQLite*:

sintaxis para ALTER TABLE

```
<alter table statement> ::=
ALTER TABLE [ <database name><dot> ] <table name> <alter table action>

<alter table action> ::=
  RENAME TO <new table name>
| ADD [ COLUMN ] <column def>
```

Además de estas limitaciones hay algunas más, al agregar nuevas columnas la definición de estas puede tomar cualquiera de las opciones disponibles en la definición de `CREATE TABLE` excepto por las siguientes:

1. La columna no puede ser clave primaria o `UNIQUE`.
2. La columna no puede tener como valor por defecto `CURRENT_TIME`, `CURRENT_DATE`, `CURRENT_TIMESTAMP`, o una expresión entre paréntesis.
3. Si la columna no puede tener valor nulo se ha de especificar un valor por defecto diferente de `NULL`
4. If foreign key constraints are enabled and a column with a REFERENCES clause is added, the column must have a default value of NULL.

4.3.3.5. Especificación de DELETE

La sintaxis de `DELETE` entendida por *SQLite* es la siguiente:

```
<delete statement> ::=
DELETE FROM <target table> [ WHERE <search condition> ]

<target table> ::=
[ <data base name><dot> ] <table name>
[ INDEXED BY <index name> ]
[ NOT INDEXED ]
```

En ella se puede encontrar unas pocas diferencias con las demás motores de bases de datos. Esta diferencia es que no es posible usar los elementos `INDEXED BY <index name>` y `NOT INDEXED` en *PostgreSQL* y *MySQL*, además cabe aclarar que no se están agregando elementos que si son soportados por los otros motores. Así que finalmente queda la siguiente sintaxis:

sintaxis para DELETE

```
<delete statement> ::=
DELETE FROM <target table> [ WHERE <search condition> ]

<target table> ::=
[ <data base name><dot> ] <table name>
```

4.3.3.6. Especificación de SELECT

Esta sentencia es quizá la más compleja a analizar debido a la cantidad de posibilidades que presenta y a que es una de las sentencias que más se utilizara dentro de un sistema por lo que es necesario poner énfasis en ella, así sin más se analiza a continuación `SELECT` tal como es conocido por *SQLite*:

```
<query specification> ::=
<select core> [ { <compound operator> <select core> }... ]
[ ORDER BY <ordering term> [ { <comma> <ordering term> }... ]
[ LIMIT <expr> [ { OFFSET | <comma> } <expr> ] ]
```

Hasta este punto la sintaxis usada es compatible con los otros dos motores, por lo que sera aceptada tal cual aunque es necesario aclarar que hay una diferencia con respecto a *MySQL* el cual permite que las consultas intervinientes en una unión puedan tener cada una una restricción `ORDER BY` y/o `LIMIT` mientras que en los otros dos motores esto solo es posible de aplicarse a la consulta resultado de la `UNION` de consultas, siendo `UNION` uno de los posibles valores para `<compound operator>`. Se siguen buscando diferencias:

```
<select core> ::=
SELECT [ DISTINCT | ALL ] <result column> [ { <comma><result column> }... ]
[ FROM <join source> ]
[ WHERE <exp> ]
[ GROUP BY <group list> ]

<group list> ::=
<ordering term> [ { <comma><ordering term> }... ]
[ HAVING <expr> ]

<result column> ::=
  <asterisk>
| <table name><dot><asterisk>
| <expr> [ [ AS ] <column alias> ]

<join source> ::=
<single source> [ { <join op> <single source> <join constraint> }... ]

<single source> ::=
  [ <data base name> ] <table name> [ [ AS ] <table alias> ] [ <index option> ]
| <left paren> <select stmt> <right paren> [ [ AS ] <table alias> ]
| <left paren> <join source> <right paren>

<index option> ::=
  INDEXED BY <index name>
| NOT INDEXED
%not in postgre
<join op> ::=
  <comma>
| [ NATURAL ] [ LEFT [OUTER] | INNER | CROSS ] JOIN

<join constraint> ::=
  [ ON <expr> ]
| USING <left paren> <comma><column name> <right paren>

<ordering term> ::=
<expr> [ COLLATE <collation name> ] [ ASC | DESC ]
%no COLLATE en mysql
<compound operator> ::=
  UNION [ ALL ]
| INTERSECT
| EXCEPT
%solo union para Mysql
```

Son pocas las incompatibilidades que se pueden encontrar, pero existen, la primera incompatibilidad es que el elemento `<index option>` dentro de `<single source>` no es aceptado por *PostgreSQL*; segundo en el elemento `<ordering item>` *MySQL* y *PostgreSQL* no soportan establecer la opción `COLLATE` que son las restricciones para `ORDER BY`; por ultimo *MySQL* solo acepta `UNION [ ALL ]` para el elemento `<compound operator>`. Así que la sintaxis final quedaría de la siguiente manera:

sintaxis para SELECT

```
<query specification> ::=
<select core> [ { UNION [ ALL ] <select core> }... ]
[ ORDER BY <ordering term> [ { <comma> <ordering term> }... ]
[ LIMIT <expr> [ { OFFSET | <comma> } <expr> ] ]

<select core> ::=
SELECT [ DISTINCT | ALL ] <result column> [ { <comma><result column> }... ]
[ FROM <join source> ]
[ WHERE <exp> ]
[ GROUP BY <group list> ]

<group list> ::=
<expr> [ { <comma><expr> }... ] [ HAVING <expr> ]

<result column> ::=
  <asterisk>
| <table name><dot><asterisk>
| <expr> [ [ AS ] <column alias> ]

<join source> ::=
<single source> [ { <join op> <single source> <join constraint> }... ]

<single source> ::=
  [ <data base name> ] <table name> [ [ AS ] <table alias> ]
| <left paren> <select stmt> <right paren> [ [ AS ] <table alias> ]
| <left paren> <join source> <right paren>

<join op> ::=
  <comma>
| [ NATURAL ] [ LEFT [OUTER] | INNER | CROSS ] JOIN

<join constraint> ::=
  [ ON <expr> ]
| USING <left paren> <comma><column name> <right paren>

<ordering term> ::=
<expr> [ASC | DESC]
```

Habiendo definido un dialecto genérico que sea soportado por todos los motores a los que se le esta dando soporte y teniendo en cuenta además de la sintaxis de las sentencias a los tipos de datos definidos y las funciones que presentan cada uno de los motores es necesario a continuación diseñar la arquitectura del proyecto atendiendo a las necesidades expuestas en este capitulo, actividad que se desarrollara en el siguiente capitulo.

Capítulo 5

Diseño

En este capitulo se documentara como se desarrollo el diseño de la arquitectura del proyecto, para ello se expondrá el modo en que trabaja *JDBC* y además se mostrara la estructura de clases que se eligió para el sistema para que este pueda cumplir con los requisitos expuestos en el Capitulo 4.

5.1. Visión general

Para diseñar *JDBGM*, como ya se menciono, se tomo como base de diseño a el patrón de desarrollo DAO[1], por lo que siguiendo la idea de este patrón *JDBGM* debe manejar todas las consultas echas hacia el motor obligando de este modo a que la aplicación solo se pueda comunicar con *JDBGM* para acceder a la base de datos, se puede observar esquemáticamente esta idea en la figura 5.1. La idea es que la aplicación utilice el formato que se esta proponiendo para manejar con las sentencias SQL, las cuales serán interpretadas por *JDBGM* y convertidas en cadenas de texto para que puedan ser entendibles por *JDBC*, este ultimo se comunicara directamente con el DBMS y pondrá a disposición de *JDBGM* el resultado de la operación.

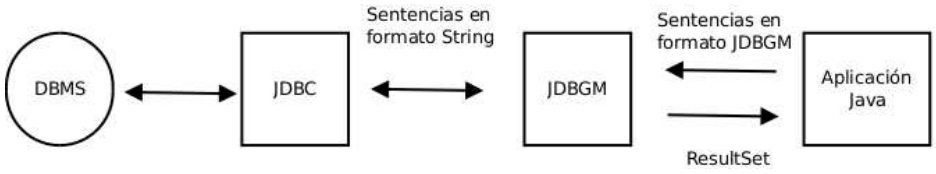


Figura 5.1: JDBGM dentro de una aplicación

Así *JDBGM* debe ser capaz de hacer dos tareas principales que están relacionadas entre si, en primer lugar debe proveer el medio de transporte/soporte para las sentencias de modo que puedan ser traducidas luego a el dialecto que corresponda según el motor con el que se esté trabajando actualmente, y en segundo lugar proveer los métodos necesarios para comunicarse con el motor mediante *JDBC*, se puede observar esquemáticamente la situación en la Figura 5.2 en la pagina 28 que muestra la idea de independencia existente entre los dos módulos principales que se están exponiendo. El modulo manejador de sentencias es totalmente independiente de la capa intermedia pues de lo que esta encargado es de proveer los métodos necesarios para "armar las sentencias" y traducirlas cuando sea requerido a el dialecto correspondiente en un formato que sea entendible por *JDBC* que es en el fondo quien se encargara de comunicarse con el motor, de este modo este modulo podría usarse independientemente como una herramienta para independizarse de los dialectos de SQL. En cambio la capa intermedia dependerá del manejador de sentencias para poder proveer la independencia de dialectos que se quiere lograr y además este solo entenderá las sentencias que utilicen el formato propuesto por *JDBGM* pues de otro modo lo único que ofrecería la capa intermedia seria una interfaz simplificada de *JDBC*. A continuación se procederá a explicar la estructura propuesta para este manejador para que luego se pueda explicar correctamente el funcionamiento de la capa intermedia.

[1]Se explico en la sección 3.2.1 de la página 8

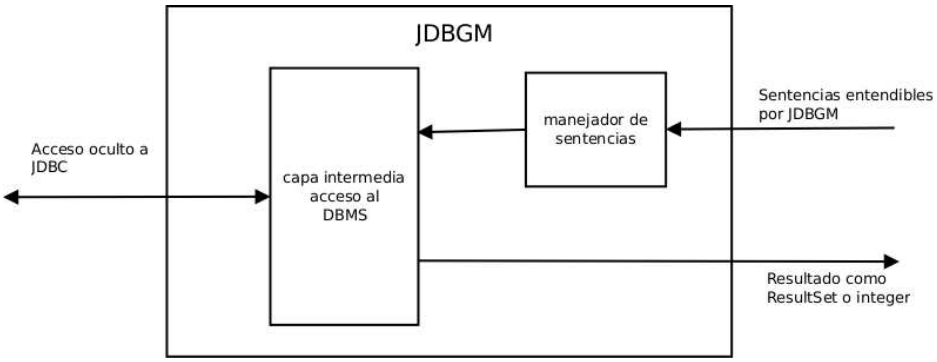


Figura 5.2: Vista abstracta del funcionamiento de JDBGM

5.2. Manejador de sentencias

Como ya se dijo el manejador de sentencias debe ser capaz de almacenar las sentencias de forma "desglosada" así que a continuación se analizara lo que se quiere lograr. Como una primer solución se puede pensar que una sentencia como la siguiente "`nombre_comando parametro opcion1 parametro1 opcion2 parametro2`" que se podría corresponder con la siguiente consulta "`SELECT * FROM table WHERE table.id=1`" puede ser almacenada en una clase que se corresponda con la siguiente estructura:

Pseudocódigo de la estructura de dato que contiene la sentencia

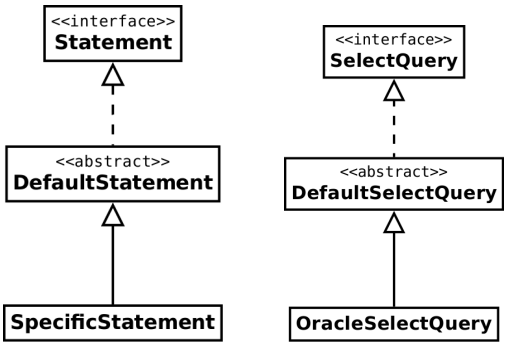
```
class Sentencia{
        nombre_comando;
        parametro;
        set_opcion1(parametro1);
        set_opcion2(parametro2);
        devolver_sentencia();
}
```

Como se ve es una idea bastante sencilla en la que la clase identificara la sentencia y sus opciones correspondientes serán nombres de atributos pues en definitiva lo que interesa de estas opciones son los valores que toman o si están presentes o no en la sentencia que se esta armando. Como clase de POO debe ofrecer métodos para almacenar los valores que tomaran las opciones y es aquí donde empiezan a jugar las reglas dispuestas en el dialecto genérico que fue especificado en el capitulo 4, puesto que las opciones ofrecidas por cada sentencia no pueden ser libremente usadas es necesario que estos métodos restrinjan el modo en pueden usarse, ya sea limitando las posibilidades de el/los métodos o bien usando excepciones para detener la ejecución del programa[2] cuando alguna acción ilegal sobre las sentencias ocurra.

Con esto ya se tiene una idea genérica de las clases que se quiere que representen a las sentencias y teniendo en cuenta que cada una de estas clases representa a una sentencia diferente con comportamiento similar al de las demás es necesario recalcar que estas diferentes clases deben presentar las mismas similitudes, por ejemplo supóngase que los objetos `select` y `create` representan las sentencias SQL del mismo nombre, las cuales precisan que se les asigne el nombre de la tabla sobre la que se esta trabajando, entonces lo correcto sera que ambos objetos tengan un método de igual nombre y comportamiento similar como por ejemplo `select.set_table_name("name_1")` y `create.set_table_name("name_2")`. Es muy importante tener en cuenta esto puesto que facilitara el aprendizaje del uso de la API del manejador de sentencias que se esta proponiendo y además hará el uso de la misma mucho más natural con respecto a SQL.

Antes de empezar a hablar de lleno de la arquitectura propuesta para el manejador de sentencia es necesario nombrar al paquete *crossdb* del cual se tomaron algunas ideas.

[2]Una explicación más detallada del manejo de excepciones se vera en las secciones siguientes.



a: Idea base b: Ejemplo con Select

Figura 5.3: Vista abstracta de la arquitectura de crossdb

5.2.1. El paquete *crossdb*

Este paquete que puede ser encontrado en SourceForge en la siguiente dirección `http://sourceforge.net/projects/crossdb/` intenta solucionar el mismo problema que el manejador de sentencias aquí presentado, tal como se puede observar en la pagina del proyecto cuyo objetivo se resume en la siguiente oración:

> "To provide cross database tools for manipulating all major databases without having to worry about each vendors specific implementation."

Esta librería usa la misma idea de ir generando bajo demanda las sentencias especificas para un dialecto en particular, por lo que se la estudio y se llego a la conclusión de que podía usarse como base para el manejador que se esta por implementar en este proyecto. Algunas consideraciones antes de avanzar:

- El paquete se distribuye bajo Apache Software License 1.1 por lo que no hay inconvenientes en reusar el código escrito, siempre y cuando se respeten las condiciones impuestas. Además esta licencia es compatible con GPL que es la licencia que usaría el proyecto cuando sea liberado.
- La ultima actualización del proyecto data el 23-08-2005 (la ultima modificación registrada en el repositorio Subversion) por lo que aparentemente el proyecto quedo en un punto muerto y en una versión beta básica según se puede ver al revisar el código.
- Carece de una buena documentación pero el código es bastante entendible y más aun después de venir estudiando una solución similar a la propuesta por *crossdb*

Se puede resumir la estructura usada por esta librería en el gráfico 5.3 de la página 29 que muestra que por cada sentencia que se desea implementar existe una interfaz (hablando siempre en POO) `Statement` que define el comportamiento de la misma, luego para esta interfaz base existe una implementación por defecto `DefaultStatement` que implementa las funciones que son comunes a toda implementación que se pueda realizar de dicha interfaz, recordando que al trabajar con dialectos se esta trabajando con variaciones de un lenguaje base, y por cada una de estas implementaciones por defecto existe una implementación especifica `SpecificStatement` que contempla las particularidades valga la redundancia de un motor en particular sobre dicha sentencia. En la misma figura se puede observar como se conforma dicha arquitectura para la sentencia `SELECT` con la clases `SelectQuery` siendo la interfaz base, `DefaultSelectQuery` siendo la implementación por defecto de dicha interfaz y `OracleSelectQuery` una implementación especifica para el dialecto utilizado por Oracle.
Además de las clases a las que se hizo referencia *crossdb* usa otras auxiliares, como `Column` que contiene los datos necesarios para especificar una columna o `WhereClause` que representa una clausula `WHERE`, clases que son necesarias para poder almacenar adecuadamente los datos para las sentencias que se están armando. También se implementa la idea del patrón *Factory* pero de una manera muy básica. Terminado de presentar ligeramente este paquete sobre el cual se baso el diseño propuesto en este trabajo se pasara a exponer la arquitectura usada por el manejador de sentencias y su relación con las ideas tomadas de *crossdb*.

5.2.2. Diseño básico del Manejador

Recordando lo expuesto en la sección anterior sobre las clases que representan las sentencias se puede resumir su contenido de manera muy genérica de la siguiente manera:

- Variables que almacenaran los valores para las opciones o parámetros de las sentencias necesarios para la reconstrucción de las mismas.
- Un método para generar la sentencia y como todo objeto métodos para inicializar la clase y poblarla con los datos necesarios para su construcción.

El comportamiento de cada una de estas clases puede ser implementado directamente desde las restricciones impuestas en la sección 4.3 de la página 12 pero siguiendo ciertos lineamientos que se verán a continuación.
En primer lugar se tiene que tener en cuenta la arquitectura general del manejador para ello hay que observar lo siguiente: se puede hacer una distinción en como es que responde el motor de base de datos a las sentencias, o más bien como responde JDBC, que es el "interlocutor" cuando procesa determinada sentencia SQL. Por un lado están aquellas que solo necesitan reportar la cantidad de filas afectadas por la sentencia, después están aquellas que necesitan devolver datos después de ejecutada la sentencia, estas ultimas sentencias son precisamente las que realizan consultas sobre la base de datos, siendo las sentencias `SELECT` las únicas que hacen esto. Por otro lado las sentencias `UPDATE`, `INSERT` y `DELETE` solo precisan reportar la cantidad de filas que fueron afectadas por la consulta, en cambio las sentencias `ALTER TABLE` y `CREATE TABLE` que sirven para modificar las tablas (son parte del DDL de SQL) no afectan directamente a las filas de la misma pero por convención se tomo que *JDBC* devuelva 0 como numero de columnas afectadas, por lo que en este caso se las agrupara en el mismo conjunto. Entonces para que *JDBGM* pueda hacer distinción de estos dos tipos de consultas se utilizaron dos interfaces las cuales pueden ser tomadas como tipos de datos genéricos, la Figura 5.4 en la pagina 30 muestra estas interfaces y sus hijas.

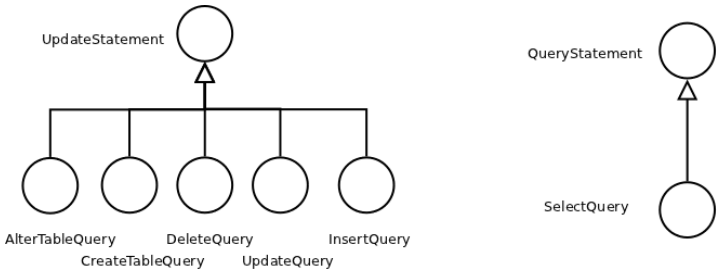


Figura 5.4: Interfaces base para los tipos de sentencia

De este modo se utilizan las interfaces `UpdateStatement` y `QueryStatement` como un tipo genérico para identificar las sentencias que no realizan consultas de aquellas que si, es decir si en la firma de cualquier método se declara algo similar a lo siguiente `metodo_a(UpdateStatement statement)` el parámetro `statement` podrá ser cualquier clase que sea heredera directa o indirecta de la interfase `UpdateStatement` y no podrá tomar ninguna heredera de la clase `QueryStatement`, cabe aclarar que este es el primer cambio que se agrego a la estructura que proponía *crossdb*.
En la Figura 5.5 en la pagina 31 se tiene una vista completa de como se debe componer la estructura de clases para el tipo base `QueryStatement` que es la más simple pues solo tiene una clase hija, la única que sirve para hacer consultas sobre la base de datos, la sentencia `SELECT`. Como ya se dijo la primer interfaz sirve para distinguir a nivel genérico el tipo de sentencia SQL con que se esta tratando, la interfaz hija directa de `QueryStatement` si ya define un comportamiento especifico para las funciones de las sentencias `SELECT`. La clase abstracta `DefaultSelectQuery` brinda una implementación base para los métodos definidos en `SelectQuery` que presentan un comportamiento común, para comprender por que se debe hacer esto es necesario recordar que la clase que represente una sentencia contiene toda la información necesaria para generar una sentencia valida y esta información es la misma que se precisa para cualquier sentencia SQL de un dialecto en concreto (además ya se definió una sintaxis que es totalmente soportada) la

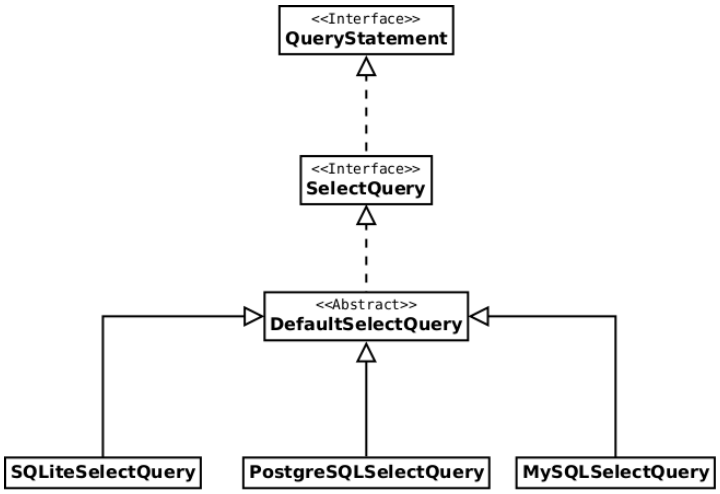


Figura 5.5: Diagrama de clases para la sentencia SELECT

diferencia esta en el modo "en que se escriben" las sentencias, la sintaxis, por ello el único método que deberá ser especifico a un *DBMS* será aquel que esta encargado de armar la sentencia. Después de la clase abstracta si por fin se tienen implementaciones especificas para cada uno de los motores, estas son la clase `SpecificSelectQuery` de las que se habla en la Figura 5.3 en la pagina 29, la idea es que de estas clases hayan tantas como motores estén soportados por el proyecto.

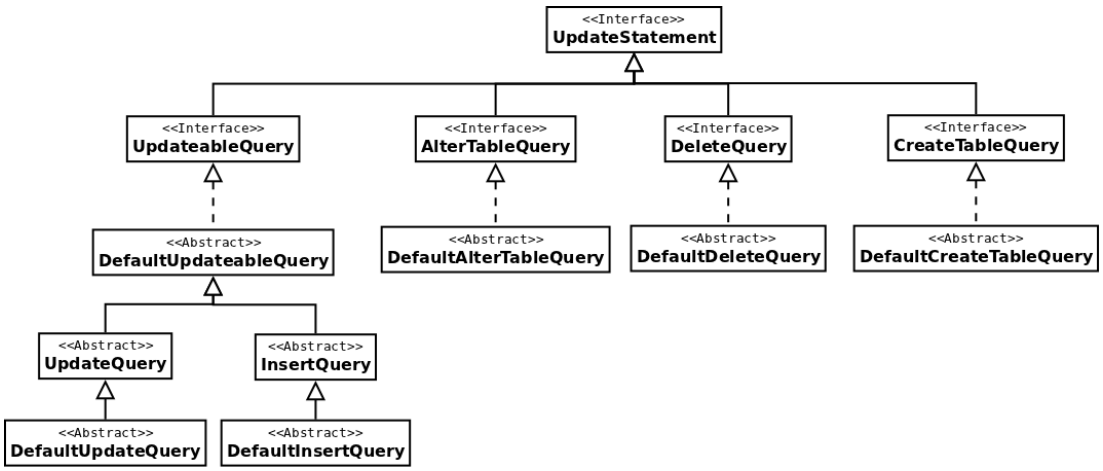


Figura 5.6: Estructura de las otras sentencias

En la figura 5.6 se tiene una descripción de como se deben estructurar los tipos de la interfaz base `UpdateStatement`, esta es algo más elaborada pero como antes la primer interfaz, funciona como tipo genérico, en las clases hijas de esta se ve algo parecido al caso de `QueryStatement` salvo por la interfaz `UpdateableQuery` que agrupa comportamiento común a `Updatequery` e `InsertQuery`, también se brinda un comportamiento predeterminado para esta mediante `DefaultUpdateStatement` pero el cual al brindar una implementación base tiene que ser a la fuerza una clase abstracta[3] así que las clases que definen el comportamiento especifico de `UpdateQuery` e `InsertQuery` son clases abstractas y no interfaces, pero el resultado es el mismo. Y finalmente para cada una de las interfaces que definen(o representan) a las sentencias se tiene una implementación por defecto en las clases cuyo nombre empieza por `Default` de las cuales deberán heredar cada una de las implementaciones especificas, en la figura 5.6 no se las representa puesto que seria demasiado extenso.

[3]Una clase abstracta se puede ver como una interfaz que fuerza la implementación de parte de su comportamiento

Después de analizar el dialecto genérico que se definió en la sección 4.3 de la página 12 y de definir la estructura de las clases para el manejador se buscaron aquellos atributos de las clases (extraídos de las sentencias que representan) que por su complejidad ameritaban convertirse en una clase por si mismas, tal es el caso de una columna que en si es una entidad distinguible y lo suficientemente compleja, como ejemplo se puede notar que una columna posee nombre, tipo de dato que almacena y muchas otras restricciones que pueden ser encontradas en el capitulo 5 en la sección que se refiere a la sentencia `CREATE TABLE`. A continuación se listan y explican las clases auxiliares que utiliza el manejador.

5.2.3. Clases Auxiliares

Como ya se denoto para un adecuado y consistente funcionamiento del manejador de sentencias es necesario crear algunas clases extras que brindan funcionalidades comunes a varias clases o bien funcionalidades extras no propias de dichas clases que ayudaran a formar las similitudes entre las diferentes clases que representan a las sentencias, a continuación un listado de estas seguida de una descripción de cada una de ellas:

- `Column`
- `TableConstraint`
- `DataTypes`
- `Formater`
- `Functions`
- `Join`
- `SQLFactory`
- `WhereClause`

5.2.3.1. La clase Column

Representa una columna de una tabla, para ello posee atributos tales como nombre de columna, banderas que indican si es clave foranea y todos los demás atributos que sirven para definir la columna en una sentencia `CREATE TABLE`. Entre sus métodos no se incluye ninguno que sirva para convertir la columna en una cadena de texto que sirva para definirla en una sentencia SQL pues a pesar de que la sintaxis de la definición de columna no varia demasiado, salvo por los tipos de datos, si varia el uso sobre ella. Por ejemplo la sentencia `ALTER TABLE` también puede definir columnas en su sintaxis pero con ciertas restricciones como se ve en la sección 4.3.3.4 y para evitar confusiones cada clase (clase que represente una sentencia) que haga uso de un objeto columna deberá implementar las restricciones sobre la construcción de la sentencia ignorando los atributos que no use.
Además de esto la columna también posee un atributo para almacenar un valor para la columna, de modo que la clase no solo servirá para definir una columna si no también como contenedor para el valor, el tipo y el nombre de la columna para poder usarla luego con otras sentencias que no necesiten definir columnas si no ingresarles datos como es el caso de `INSERT`. Haciendo esto se evita crear otra clase que solo sirva de contenedor para el valor de una columna y por lo tanto hay menos clases para recordar. Además se deja abierta la posibilidad de establecer a que tipo de datos pertenece el valor que se esta ingresando para poder manejar el modo en que este valor es convertido a `String`, para aclarar esto supóngase que se quiere ingresar un valor booleano a una columna y como los constructores de esta clase aceptan para el parámetro que establece el valor de la columna un objeto de la clase `Object` se le puede pasar cualquier objeto por lo que intuitivamente se le podría pasar una cadena por ejemplo `"TRUE"` a menos que se indique el tipo de columna a la clase la clase puede inferir que se quiere enviar una cadena de texto y convertirla a por ejemplo `.. SET boolean_column = 'TRUE'` que puede llevar a un error cuando se intente enviar la sentencia a el motor, en cambio si el tipo de dato es establecido a boolean la clase lo convertirá en un `1` o `0` según corresponda.

5.2.3.2. Las restricciones de tabla en TableConstraint

Esta clase representa lo que en una sentencia `CREATE TABLE` vendría a ser una restricción de tabla, las cuales pueden ser `PRIMARY KEY, FOREIGN KEY o UNIQUE` estas restricciones a su vez pueden hacerse sobre una (clave simple) columna o bien sobre varias (clave compuesta). La razón por la que se eligió crear una clase para representar las restricciones de tabla es que de no existir se complicaba mucho la "traducción" a SQL de la clase que representa a `CREATE TABLE`, esta clase representa un medio más sencillo para la traducción de la restricción, como contra parte se puede observar que la sintaxis se vuelve un poco más compleja pero aun así es mayor el beneficio pues se hace más sencilla la traducción a cadena de caracteres.

5.2.3.3. La clase DataTypes

Esta es una clase abstracta que servirá para mapear los tipos de datos a los correspondientes a los de un motor especifico, esto se logra mediante un método que toma como entrada los tipos de datos definidos por *JDBC* y los mapea a los tipos de datos encontrados en el capitulo anterior.

5.2.3.4. La interfaz Formatter

La Interfaz `Formatter` sirve para definir el modo en que algunos tipos de datos han de ser convertidos a su equivalente como cadena de caracteres, se puede decir que define un formateador. Se provee una implementación por defecto de esta interfaz en `DefaultFormatter` la cual puede llegar a servir sin modificaciones para el motor sobre el que se esté trabajando, aun así esta implementación por defecto obligara a que se cree una clase que herede de la misma puesto que se tratara de una clase abstracta y como tal no puede ser instanciada directamente. La clase que herede de dicha implementación puede no definir nada y ser creada solamente para poder instanciar la implementación base a través de esta, pero se exige su implementación por si llegase a ser necesario sobrescribir (*override*) algunos de los métodos ya implementados para manejar adecuadamente algún tipo de dato según lo requiera el motor de bases de datos.

5.2.3.5. Las funciones en Functions

Como se detalla en la sección 4.3.2 de la página 16 el soporte para las funciones es básico por lo que la clase `Functions` contendrá una lista de constantes con los nombres de las funciones reconocidas por *JDBGM* como cadenas de texto `String`, como algunas de estas funciones tienen distintos nombres en los otros motores se precisara que por cada motor en particular exista una clase que herede de `Functions` y "sobre escriba"los valores asociados a las constantes cuyos nombres son diferentes. Como los nombres y valores de las constantes que define la clase `Functions` se toman de las funciones que define *SQLite* la clase en particular que se implemente para este motor puede ser una clase que no defina nada. Además en ninguna parte se exigirá el uso de esta clase solo se sugerirá su uso pues lo único que brinda es nombres de funciones quedando el paso de los parámetros resumido en otra cadena de texto que deberá suplir el programador.

5.2.3.6. Join

La clase `Join` se encargara de dar soporte para la creación de las restricciones `JOIN` que se le pueden agregar a `SELECT`, esta clase no se debe encargar de verificar la existencia de errores en la restricción que se esta construyendo pues como ya se dijo seria una redundancia de trabajo. Como las restricciones de este tipo tienen una sintaxis bastante sencilla, que se puede ver en la sección 4.3.3.6, esta puede ser construida a partir de una única función con los parámetros adecuados pero esto significaría un uso excesivo de parámetros y por lo tanto más código por escribir por lo que la clase deberá proveer métodos que sean atajos para construir las diferentes restricciones posibles, además si se le ponen nombres lo suficientemente claros a las funciones estas pueden ayudar a una mejor lectura del código escrito.

5.2.3.7. Fabrica de objetos SQLFactory

Clase que implementa el patrón *Abstract Factory* para manejar los diferentes tipos de implementaciones especificas de las sentencias, de modo que a la hora de instanciar las sentencias "clases" el programador quede liberado de realizar la elección de la clase correcta de la cual debe instanciar. Una explicación más profunda sobre los patrones *factories* se vera más adelante en este capitulo.

5.2.3.8. La restricción WHERE como WhereClause

Como la clausula `WHERE` no es más que una lista de condiciones en la que intervienen los mismos operadores lógicos, comparativos y específicos de SQL como por ejemplo el operador `IN` se decidió crear una clase que lo contenga, en resumen no hay implementación especifica pues es la misma para todos, además el uso de una misma clase para generar las restricciones en las distintas sentencias facilita y "unifica" el uso de las mismas.
En definitiva esta clase debe proveer un amplio abanico de funciones para facilitar el ingreso de los diferentes tipos de condiciones que intervienen en la clausula `WHERE`. Entre las sentencias que utilizan esta clase tenemos a: `SELECT`, `UPDATE` y `DELETE` estas deberán proveer un método que devuelva una referencia a un objeto `WhereClause` que sera miembro de la misma e igualmente llamado, el método. Este método debe ser especificado en las diferentes interfaces que especifiquen el comportamiento de las sentencias que recién se nombraron.

5.2.4. Manejo de errores

El manejo de errores o excepciones en el manejador de sentencias estará limitado en dos aspectos: primero solo se lanzaran errores de tiempo de ejecución y segundo los errores se producirán cuando se quieran realizar acciones ilegales que no precisen de muchos cálculos, por ejemplo una sentencia `SELECT` no se puede "construir"[4] si es que no se estableció una fuente para la misma, ya sea desde una tabla u otra consulta embebida, para lo cual resulta sencillo verificar que se haya establecido o no algunas de estas fuentes, en cambio chequear la sintaxis de toda la sentencia requiere de una mayor complejidad tanto a nivel estructural como de cálculos, además esta tarea ya la realizara el DBMS cuando reciba dicha sentencia.

5.2.4.1. Tipos de excepciones

En Java como en muchos lenguajes existen dos tipos de excepciones de las que se pueden hacer uso[14] para comunicar errores o comportamientos inesperados que hayan sido contemplados en el código:

1. ***checked exceptions***: aquellas excepciones de las que el programa se puede recuperar y que pueden ser capturadas para que el programa tome acciones con respecto de dicho error.

2. ***unchecked exceptions***: aquellos errores de los que el programa no se puede recuperar y que por lo general se corresponden a "bugs" existentes en el código. Este tipo de excepciones es también conocido como excepciones de tiempo de ejecución.

En el manejador de sentencias de *JDBGM* los errores que pueden ocurrir se resumen primariamente en un incorrecto uso de la API entregada y el único modo en que se pueden corregir estos errores es corrigiendo el código que fue escrito, por lo que no es posible tomar medidas en tiempo de ejecución para intentar corregir dichos errores. Este precisamente es el tipo de errores irrecuperables que cubren las *unchecked exceptions* por lo que es el tipo de errores que lanzara *JDBGM* para su manejador de sentencias. La clase `RuntimeException` de Java es una de las clases que entra entre las excepciones de dicho tipo y es la que sera utilizada por el manejador de sentencias para lanzar sus excepciones.

[4] En realidad si se puede crear una sentencia SELECT sin una tabla definida pero para el propósito de *JDBGM* esto no sirve.

5.2.5. Diseño de cada una de las sentencias

Una vez que se presento el esqueleto del proyecto ya se puede empezar a ver las particularidades de cada una de las ramas finales que en este caso son las interfaces, aunque es bueno aclarar que estas particularidades son las que precisamente delinearon a la estructura que se expuso anteriormente, por lo que a continuación se explicara en más detalle la relación de las interfaces que definen a las sentencias y las clases auxiliares como así también detalles que se creen importantes, recordando que las interfaces quedan casi completamente definidas por la especificación que se dio en la sección 4.3.

5.2.5.1. La interfaz para CREATE TABLE

La sentencia `CREATE TABLE` representada en la interfaz `CreateTableQuery` básicamente sirve para definir columnas y restricciones de tabla, aunque en las columnas también es posible definir algunas de las restricciones estas están limitadas ya que solo se pueden definir sobre dicha columna. En el dialecto genérico que se definió anteriormente se encontraron compatibilidades con las definiciones de restricciones de columna pero al analizar la estructura necesaria para una implementación de la misma a nivel clases se termino por excluirlas por que complicaban la conversión del objeto a `String` al ser necesario estar comprobando si una columna pertenece a una restricción de columna o a una restricción de tabla (lo que involucraba iteraciones extras), además restringir el tipo de restricciones a las de tabla únicamente evidencio la necesidad de una clase que almacene dicha restricción con la consiguiente simplificación de la interfaz principal. La Figura 5.7 en la pagina 35 muestra como se componen las relaciones de `CreateTableQuery`, sus clases auxiliares y la implementación por defecto de la misma.

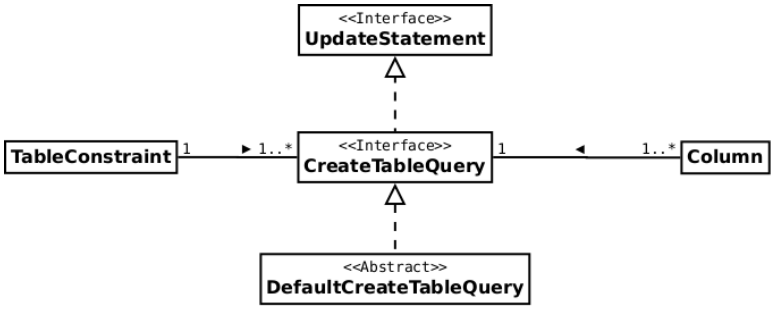


Figura 5.7: Diagrama de clases para la interfaz de CreateTableQuery

La clase `TableConstraint` representa a las restricciones de tabla las cuales necesitan referencias (punteros a los objetos) de las columnas que la componen, la tabla sobre la que se están creando las restricciones y además otros atributos que tienen las restricciones de tabla que se pueden ver en la sección 4.3.3.1 de la página 19. Lo que no se contemplo en esta interfaz y tampoco es posible definir mediante la misma es el control de la integridad de la sentencia, en el sentido de que si es correcta o no pues de hacerlo aumentaría el procesamiento necesario para poder convertir esta sentencia a cadena de caracteres complicando y agregando trabajo extra (*overhead*) a la capa de software que accede a el motor, trabajo que en definitiva el motor terminara realizando de nuevo produciéndose una redundancia innecesaria. Lo único que cambiara al obviar estas comprobaciones es quien terminara por reportar el error, por lo que no se realizan este tipo de comprobaciones en el manejador de sentencias.
Por ultimo hay que señalar el uso de la clase `Column` para almacenar la definición de las columnas, que además después sera reutilizada como contenedora de valores. Esta clase originalmente contenía las definiciones de las restricciones de columnas pero desde la creación de la clase `TableConstraint` estas fueron eliminadas con la única excepción de la restricción combinada de clave primaria con columna auto-incrementada del tipo entera.

5.2.5.2. La interfaz para UPDATE

La sentencia `UPDATE` al igual que la sentencia `INSERT` precisa que se le indiquen las columnas sobre las cuales se quiere trabajar, pudiendo ser todas o solo algunas de las columnas de una tabla, es por ello que se decidió crear una interfaz intermedia que defina este comportamiento

para ambas sentencias. Dicha interfaz es `UpdateableQuery` la cual precisamente define los métodos necesarios para poder establecer cuales serán las columnas sobre las que se trabajara. Una vista completa de como se disponen las clases necesarias para esta sentencia se puede ver en la figura 5.8 en ella se puede apreciar que esta interfaz común hace uso de la clase `Column` para almacenar los nombres y valores para las columnas además de los tipos de datos que se precisan para esa columna, aunque por como quedo definida la interfaz se deja abierta la posibilidad de que *JDBGM* decida a que tipo de dato pertenece el valor que se le acaba de establecer, esto se hace mediante las implementaciones de la interfaz `Formatter` que decide como se debe interpretar el valor de la columna. Y por ultimo antes de entrar directamente a la interfaz especifica para esta sentencia se puede ver que existe una implementación base para que el comportamiento no solo quede definido sino que también implementado.

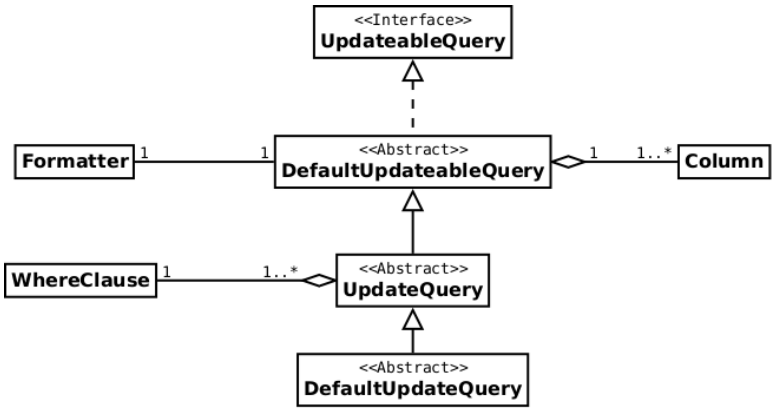


Figura 5.8: Diagrama de clases para la interfaz de UpdateQuery

Poco le queda por definir a la interfaz `UpdateQuery`, como se puede apreciar en la sección 4.3.3.2, falta por definir la restricción `WHERE` lo cual se hace mediante la clase auxiliar `WhereClause`, por lo que esta interfaz debe poder devolver un objeto de dicha clase para crear la restricción, este objeto debe ser atributo de la clase que implemente esta interfaz para que luego la clase pueda interactuar con ella y obtener la restricción.

5.2.5.3. La interfaz para INSERT

La sentencia `INSERT` como ya se menciono hace uso de la interfaz común `UpdateableQuery` para definir su comportamiento base, luego como puede observarse en la figura 5.9 sera la interfaz `InsertQuery` la que defina el comportamiento especifico para dicha sentencia. En esta interfaz se define el comportamiento especificado en la sección 4.3.3.3, no hay más para aclarar con respecto al diseño salvo que esta sentencia puede ser armada a partir de tres fuentes distintas para las cuales existen sendas funciones, como el uso de estas fuentes debe ser excluyente una de la otra, es decir solo se puede usar una fuente a la vez, se decidió que la interfaz optara como fuente para la sentencia a aquella que haya sido establecida en ultima instancia mediante la llamada a el método correspondiente.

5.2.5.4. La interfaz para ALTER TABLE

La sentencia `ALTER TABLE` expresada en la interfaz `AlterTAbleQuery` presenta una estructura bastante simple como se puede ver en la 5.10. Dicha simplicidad ya había sido denotada en la sección 4.3.3.4 del capitulo de especificación por lo que no hay mucho para aclarar, además esta sentencia no esta contemplado dentro de lo que serian los métodos CRUD (al igual que `CREATE TABLE`) pero como *crossdb* presentaba un soporte para esta se la termino por agregar como una funcionalidad extra.

5.2.5.5. La interfaz para DELETE

La sentencia `DELETE` definida en la interfaz `DeleteQuery` es otra de las que es bastante sencilla según la especificación dada en la sección 4.3.3.5 de la página 24 en la que se ve que aparte del

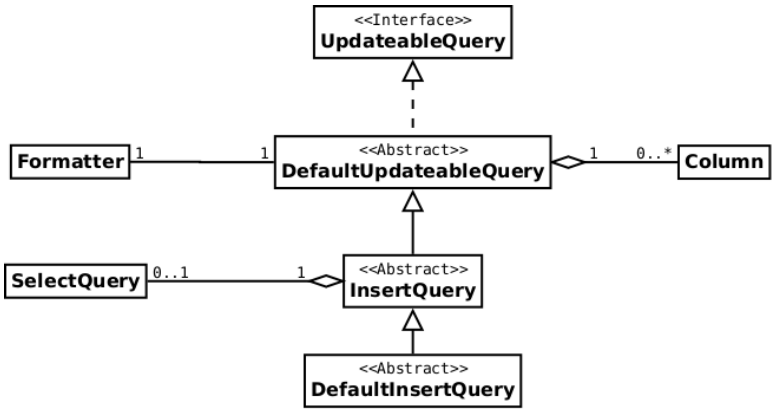


Figura 5.9: Diagrama de clases para la interfaz de InsertQuery

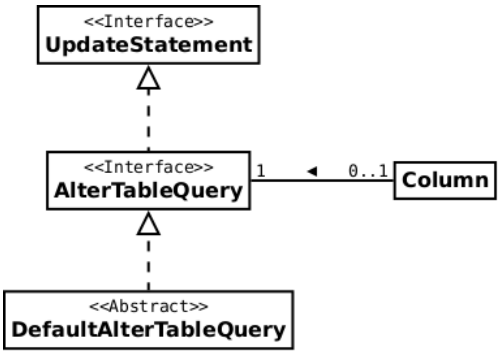


Figura 5.10: Diagrama de clases para la interfaz de AlterTableQuery

nombre de la tabla sobre la que se esta trabajando necesita de una clausula `WHERE` la cual es proveída por la clase `WhereClause`, véase la Figura 5.11 en la pagina 37.
Esta sentencia es un buen ejemplo de como un apropiado reconocimiento de objetos puede simplificar a los propios objetos, en este caso `WhereClause` esta liberando de mucho trabajo a las implementaciones de `DeleteQuery` y de paso ayudando a la reutilización de código al estar disponible como objeto para que otras interfaces que lo puedan utilizar.

5.2.5.6. La interfaz para SELECT

La sentencia `SELECT` definida en la interfaz `SelectQuery` es la más complicada de todas con las que se trabajo, para no sobrecargarla muchas de las tareas se relegaron a clases auxiliares, la primera de ellas es la clase `WhereClause` que ya se había utilizado previamente para crear restricciones del tipo `WHERE`, la segunda clase auxiliar que fue creada exclusivamente para esta sentencia es la clase `JOIN` que se encarga de ofrecer los métodos necesarios para establecer las

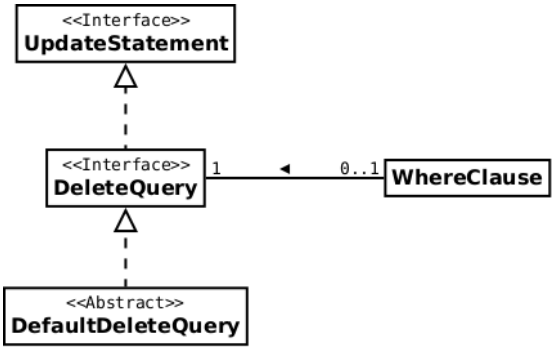


Figura 5.11: Diagrama de clases para la interfaz de DeleteQuery

restricciones de origen para `FROM`. La clase que se podría haber utilizado pero que no se utilizo es `Column` puesto que para `SELECT` lo que realmente importa es el nombre de la columna y quizá un alias y nombre de la tabla de la misma, pero estos datos no justifican el uso de `Column` por lo que se decidió no usar la misma.

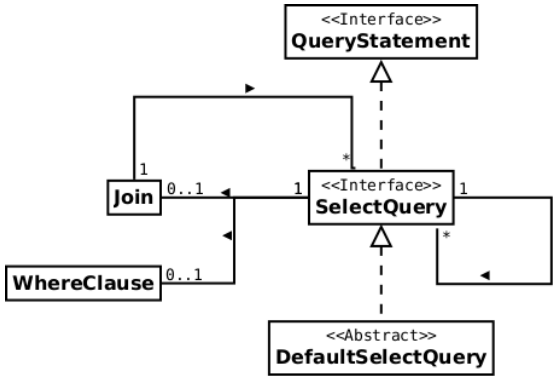


Figura 5.12: Diagrama de clases para la interfaz de SelectQuery

En la figura 5.12 esta representado el diagrama de clases para `SELECT`, en este se puede ver que la clase hace uso de si misma pues es posible hacer una operación `UNION` con otra/s sentencia/s `SELECT`, la clase `Join` también hace uso de esta interfaz por que `SELECT` se puede usar como sentencia anidada dentro de una restricción `JOIN`. Otro punto importante dentro de esta clase es el soporte de las funciones agregadas de las bases de datos, estas como ya se dijo no son, en su mayoría, parte del estándar y por lo tanto se encuentra bastante disparidad ente ellas. Por ello para el manejador de sentencias se opto que las funciones se podrán elegir desde un catalogo de constantes disponibles en la clase `Functions` que proveerá nombres de funciones que tienen equivalentes en los tres motores, quedando a cargo del programador el correcto uso de la misma.

5.2.6. Controlando la creación de objetos

Observando la estructura que se fue proponiendo hasta ahora se tienen una tres implementaciones concretas por cada una de las interfaces que definen a las diferentes sentencias, una por cada motor al que se le da soporte. Así que con lo expuesto hasta ahora el programador debe decidir cual es la implementación que necesita usar y en base a ello instanciar siempre las del mismo tipo teniendo en cuenta el motor que se quiere usar. Siguiendo esta lógica el código resultante puede resultar difícil de mantener, obstruyendo el objetivo primordial de *JDBGM*, pues en el momento que se desea migrar de motor sera necesario buscar una por una las variables que se fueron declarando para las sentencias SQL para corregir el tipo con el que fueron declaradas y además cambiar a el constructor adecuado o también puede ocurrir que el programador haya usado correctamente como tipo de dato para las variables a su correspondiente interfaz en cuyo caso aun restaría por corregir el uso del constructor adecuado. Para sortear este problema se hará uso del patrón *Abstract Factory*.

5.2.6.1. El patrón Abstract Factory

Se puede decir en forma resumida que el patrón *Abstract Factory* le permite a un cliente crear objetos que son parte de una familia de objetos relacionados o dependientes. El tema o rasgo en común de esta familia de objetos puede llegar a ser pertinente a muchas más clases por lo que en estos casos se suele crear paquetes paralelos que mantienen separadas estas familias de objetos dependiendo de como las afecte el rasgo en común. Una clase que implemente este patrón le provee a una clase cliente una fabrica que le permite crear objetos que están relacionados mediante un rasgo en común, es decir creara solo aquellas que compartan el rasgo en común. Una descripción mucho más detallada y sencilla de comprender puede ser encontrada en el siguiente libro[2].
Bajando un poco el nivel de abstracción se puede ver que para que las diferentes familias de clases estén relacionadas estas deben implementar u heredar de una interfaz o clase abstracta

en común, entonces una implementación posible de este patrón consiste en crear una interfaz Fabrica que defina métodos para obtener las implementaciones de cada una de las interfaces de la familia, de este modo cada una de las familias implementara esta interfaz Fabrica para devolver instancias de objetos que se correspondan con su familia, utilizando siempre como tipo a las interfaces que definen a la familia. Con esto se obtendrán tantas Fabricas especificas como familias existan por lo que resta obtener la fabrica adecuada dependiendo de la variable externa que hace de tema común para la familia. el concepto quedara más claro cuando se exponga el modo en que fue implementado el patrón en este caso.

5.2.6.2. Implementación del patrón

En el manejador de sentencias las familias vendrían a estar formadas por el conjunto de implementaciones de las interfaces que definen las sentencias SQL soportadas y habría una familia por cada motor al que se le esté dando soporte, para estas el tema en común seria precisamente el motor en especifico que se esté usando. Viéndolo desde otro punto de vista dependiendo de el motor en particular que se esté usando se deberá instanciar objetos de una familia determinada. Algo que queda un poco implícito con el uso de este patrón es que una vez que se tiene la "fabrica" adecuada ya no es necesario preocuparse por que tipo de objetos se esta instanciando y si se usan como tipo de datos a las interfaces, para cambiar de motor lo único que se tiene que hacer es cambiar de "fabrica".
En este proyecto la clase que se encargara de implementar el patrón Abstract Factory sera `SQLFactory`, que tiene que realizar dos tareas principales: primero se encargara de definir métodos abstractos para devolver instancias de las diferentes clases que implementen las interfaces base; segundo se encargara de decidir de cual familia de clases se deben instanciar los objetos. Para la segunda tarea la clase deberá verificar si es que ya se estableció con que motor se desea trabajar y en base a esto decidir que implementación especifica de la clase *factory* debe trabajar, pues la razón de definir como clase abstracta a `SQLFactory` es que los métodos abstractos de la misma sean implementados en clases especificas para que estas devuelvan el tipo adecuado de objetos, esto se puede apreciar mejor en la figura 5.13 que sintetiza lo que se vino explicando.

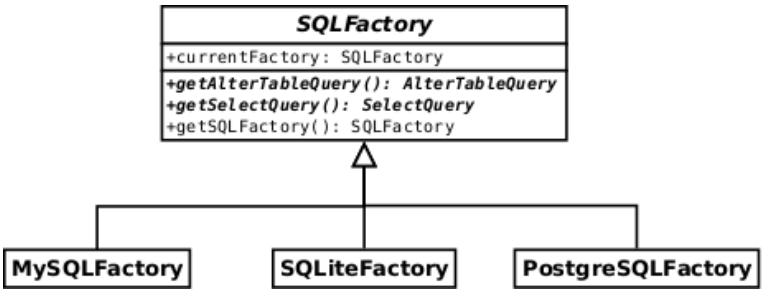


Figura 5.13: Diagrama de clases para las clases "Fabrica"

En dicha figura no se muestran todos los métodos abstractos que debe tener `SQLFactory` pero debe existir uno por cada interfaz base y siguiendo el mismo molde `get[Interfaz](): Interfaz`, el método de clase `getSQLFactory()` debe verificar que se haya establecido el motor con el cual se quiere trabajar para poder decidir de que implementación se debe instanciar. Las clases que implemente a la fabrica lo único que deben hacer es implementar de forma adecuada los métodos abstractos.

5.3. Capa de acceso a el motor - JDBC

Una vez diseñado el manejador de sentencias es necesario diseñar la capa que se encargara de abstraer el uso de *JDBC*, esta capa como ya se menciono trabajara las sentencias SQL con el formato propuesto en el manejador de sentencias, de modo que obligara el uso del mismo en pos de mantener la compatibilidad. La principal tarea de esta capa será la de ocultar el uso de *JDBC* a la vez que se ofrece una interfaz mucho más simple y fácil de usar junto con pre-configuraciones para facilitar aun más su uso. En orden de establecer el diseño para esta

capa es necesario entender algunos puntos importantes en el funcionamiento de la API que se esta ocultando.

5.3.1. Como funciona JDBC

JDBC es un API bastante completo para poder acceder, según su documentación, virtualmente a cualquier tipo de datos tabulares contándose entre estos a las bases de datos relacionales, el flujo de trabajo minimo cuando se usa *JDBC* se puede resumir en los siguientes pasos:

1. Establecer una conexión.
2. Crear un `statement`.
3. Ejecutar una consulta.
4. Procesar el objeto. `ResultSet`.
5. Cerrar la conexión.

Para **establecer la conexión** con la fuente de los datos, una BD relacional en este caso aunque puede tratarse de otros medio, es necesario contar con el *driver* jdbc adecuado para poder obtener un objeto de la clase `Connection` que es la que representara la conexión, después de esto mediante la conexión ya se puede **crear un objeto de** `Statement` que es una interfaz que representa sentencias SQL "vacías" y brinda diferentes métodos para poder **ejecutar** dichas sentencias sobre el motor, existen tres tipos `Statement, Preparedstatement` y `CallableStatement` siendo estas dos ultimas extensiones de la primera. Una vez ejecutada la sentencia, dependiendo del tipo de sentencia que se haya ejecutado, se obtienen resultados como `int` o `ResultSet`, esta ultima clase es la que interesa pues da acceso a los datos que son resultado de la consulta mediante un cursor para ir obteniendo fila por fila los datos obtenidos. Por ultimo una vez que se obtuvieron o modificaron los datos necesarios es necesario liberar los recursos mediante los métodos `close()` que ofrecen `Connection` y `Statement`, que sirven en el primero para terminar la conexión con la base de datos lo que cierra completamente la comunicación con el motor y liberar los recursos usados por la consulta en el segundo[1].
Para entender un poco mejor el funcionamiento se analizara una porción de código mínima que ilustra el uso de esta API y muestra que al hacer uso de el es necesario hacer algunas configuraciones especificas que pueden ser "automatizadas".

Porción de código java para la conexión a una base de datos

```
class conectDB(){
...

        Class.forName("com.mysql.jdbc.Driver");
        Connection conexion = DriverManager.getConnection(
                "jdbc:mysql://localhost/AsistenciaAlumnos", "tester",
                "tester");
        Statement st = (Statement) conexion.createStatement();
        String query = "SELECT␣*␣FROM␣tabla";
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
                String coffeeName = rs.getString("COF_NAME");
                System.out.printline(coffeeName);
        }
        rs.close();

...
}
```

En esta porción de código lo que se hace inicialmente es instanciar el driver el cual es necesario que exista para poder obtener una conexión a la BD, es decir crear el objeto driver, luego el método estático `DriverManager.getConnection()` devuelve un objeto que representa la conexión al motor con el que se esta trabajando. Este objeto del tipo `Connection` provee métodos para generar los objetos `Statement` que permiten hacer consultas a la base de datos mediante las funciones que dispone como por ejemplo el método `executeQuery()` que permite hacer consultas del tipo `SELECT` y que como resultado devuelve un objeto `ResultSet` del que se

pueden ir extrayendo fila por fila los datos obtenidos. Los objetos `ResultSet` y `Statement` están sumamente relacionados por ejemplo si se cierra una sentencia mediante `Statement.close()` el `ResultSet` obtenido de la misma también sera cerrado, además solo pueden existir en una relación uno a uno. Una descripción mucho más extensa de JDBC puede ser encontrada en la documentación disponible en la web de Oracle[13].

5.3.2. El diseño

La API que ofrecerá *JDBGM* sera sencilla para simplificar el uso de *JDBC*, tal como se comento en el capitulo 4 este debe ofrecer métodos para realizar consultas y manejar la conexión con el motor, así que para generalizar el API se creara una interfaz para definirla, la cual tendrá una única implementación base, al menos para este caso, la cual deberá tener implementaciones especificas para cada uno de los motores.

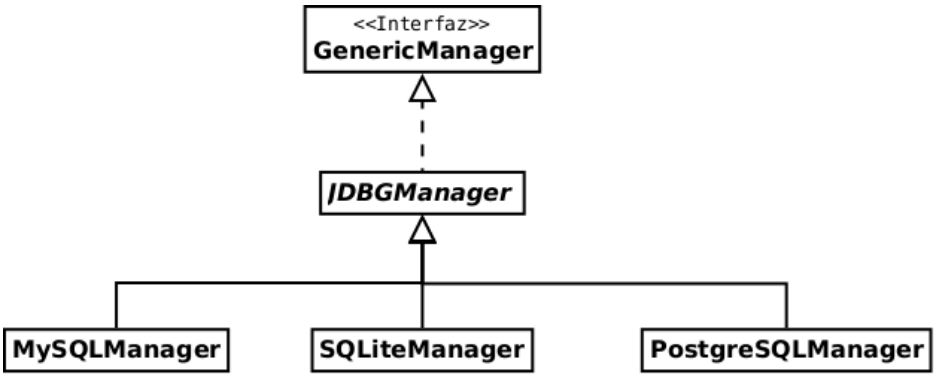


Figura 5.14: Diagrama de clases para la capa de abstracción de JDBC

En la figura 5.14 se puede ver claramente la estructura elegida para la capa de abstracción, en esta se puede ver la interfaz que se encarga de definir la API en `GenericManager` la cual puede parecer que es innecesaria pero si se tiene en cuenta que en el futuro se puede usar una interfaz diferente de *JDBC* para acceder a los motores toma algo de sentido la existencia del mismo aparte la interfaz hace mucho más clara la lectura de los métodos disponibles. La clase abstracta `JDBCManager` es la implementación de la interfaz anterior haciendo uso de *JDBC* en la que se deben implementar todos los métodos definidos puesto que en lo que diferenciaran las implementaciones especificas es valga la redundancia en configuraciones especificas para cada uno de los motores, estas diferentes implementaciones entonces únicamente deberán darle valores específicos para los atributos no configurables como por ejemplo el nombre del driver. Resta por definir los métodos que serán necesarios para poder cumplir con las expectativas actuales del proyecto, estos se los puede ver en el siguiente pseudocódigo:

interfaz GenericManager

```
interface GenericManager{
        Connection getConnection() throws JDException;
        void beginConnection() throws JDException;
        void endConnection() throws JDException;
        int update(UpdateStatement update) throws JDException;
        ResultSet query(QueryStatement query) throws JDException;
        int updateAndClose(UpdateStatement update) throws JDException;
        CachedRowSet queryAndClose(QueryStatement query) throws JDException;
        void beginTransaction() throws JDException;
        void commit() throws JDException;
        void endTransaction() throws JDException;
}
```

El método `beginConnection()` inicializa la conexión con la base de datos y revisa que los parámetros pasados al constructor de la clase que implemente esta interfaz sean los adecuados, `endConnection()` es utilizado para liberar de forma explicita los recursos del DBMS, `update()` es usado para realizar cualquiera de las acciones `UPDATE, DELETE` u `INSERT` definidos en SQL, `query()` es para realizar solamente consultas, la necesidad de diferenciarlas esta en

que mientras que una acción de `update` solo necesita informar a cuantas columnas a afectado, una acción de query necesita devolver los datos que son resultado de la consulta. Los métodos `updateAndClose()` y `queryAndClose()` realizan acciones similares que las dos descriptas anteriormente con la diferencia de que estas dos ultimas obtienen una conexión propia para realizar su acción mientras que las primeras lo hacen sobre una única conexión la cual esta activa mientras el objeto `GenericManager` que la contiene no sea eliminado o hasta que esta sea explícitamente cerrada, `beginTransaction()` y `endTransaction()` son utilizados para demarcar el inicio y fin de una secuencias de acciones que serán tomadas como una única transacción por la base de datos, es decir todos los métodos `update()` que sean llamados dentro de la sección demarcada se harán permanentes en la base de datos solamente una vez que se salga de dicha sección (esta característica puede no estar disponible en algunos *DBMS*) Las clases `QueryStatement` y `UpdateStatement` son las contenedoras de las sentencias SQL como ya se explico anteriormente. Otra cosa importante de aclarar es que cualquiera de estos métodos puede fallar ya sea por problemas de conexión, falta de driver, intentar leer un dato inexistente, etc. es por ello que estos métodos lanzan excepciones.

5.3.3. Manejo de errores y excepciones

Un importante tema a tener en cuenta en esta parte del proyecto es el manejo de errores, en *JDBC* casi todos los métodos son capaces de lanzar excepciones del tipo *checked exceptions* los cuales deben ser capturados o bien lanzados nuevamente por el método que lo recibió caso en cual el método deberá ser marcado con la palabra reservada `throw` del modo siguiente `metodo() throws Exception`. Como *JDBGM* esta echo para escribir más código sobre el, este no puede decidir concretamente que hacer cuando ocurren las excepciones a diferencia de lo que ocurre con el manejador de sentencias donde todos los errores que se pueden producir son del tipo *unchecked exceptions* que interrumpen la ejecución del programa. En esta sección los errores deben ser capturados y tratados pero no por *JDBGM* si no por el que esté haciendo uso de el mismo y es por eso que las excepciones que debe lanzar *JDBGM* deben ser también del tipo *checked exceptions*.
Como la fuente de las excepciones en este caso es *JDBC* todas las excepciones que lancen los métodos del API de la capa de acceso a el motor serán básicamente errores que envuelven a los errores que devuelve *JDBC* que son del tipo `SQLException` o alguna subclase de la misma. Para ellos se creara la clase `JDException` la cual sera una subclase de `Exception` que es la clase base para este tipo de excepciones, como la principal tarea de esta excepción es la de servir de envoltorio para las excepciones que puedan ser lanzadas por los métodos subyacentes, la misma debe contar con un atributo del tipo `SQLException` para contenerlas, además como los métodos de este API ocultan el uso de más de una función cada uno de los cuales puede lanzar un error por diferentes razones `JDException` ah de tener información extra con respecto al método donde se produjo el error. Una vista abstracta de estas clases se puede ver en el diagrama 5.15.

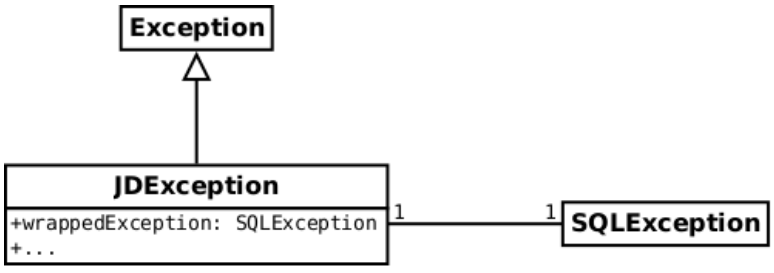


Figura 5.15: Diagrama de clases para JDException

Como la información extra que tiene `JDException` concierne más a la capa de abstracción de *JDBGM*, la información que le interesa al programador estará muchas veces en la clase `SLQException` (suponiendo que no haya surgido ningún problema en *JDBGM*), por lo que este, el programador, deberá conocer la información que esta contiene. Una descripción de `SLQException` puede ser encontrada en la web de Oracle[14] donde se explica también como se obtiene información de la excepción.

5.3.4. Que se devuelve como resultado

Al ejecutar las sentencias SQL sobre el motor es posible obtener uno de dos tipos de resultados posibles dependiendo de si son consultas (`SELECT`) las cuales devuelven un tipo de dato complejo representado por la clase `ResultSet` que representa la tabla resultado de la consulta o bien un tipo de dato primitivo `int` para las otras sentencias que usualmente indica la cantidad de filas que fueron afectadas por la sentencia salvo para el caso de las sentencias del tipo DDL que no afectan a las filas de datos.
Cuando el tipo de datos devuelto es `int` no hay mucho para decir salvo que ese mismo valor sera el devuelto, para el caso de `ResultSet` también sera devuelto el mismo objeto por que como *JDBGM* no esta implementando un `ORM` es necesario un modo de extraer los datos de la BD y esto esta brindado por `ResultSet` de una manera bastante completa por lo que proveer un método propio para extraer los datos seria como envolver los métodos de dicha clase sin ningún añadido, por lo que se estaría frente a una clase que lo único que hace es renombrar los métodos de `ResultSet`, por otro lado esta clase es implementada por el proveedor del driver por lo que es ïndependiente"de un motor en especifico, salvo que existan implementaciones que mal interpretaron las interfaces de *JDBC* pero dicho caso es bastante improbable.

5.3.5. *Factories* en la capa de acceso

Igual que con el manejador de sentencias también acá se debe controlar la creación de objetos. pero en este caso es mucho más sencillo puesto que solo se poseen instancias de una única familia, si es que se piensa como en *abstract factory*, por lo que en este caso se usara la idea de un patrón similar pero diferente como lo es *Factory method* que es una simplificación del anterior en la que existe un método que es el que decide la instancia correcta que se a de crear. En este caso el método no elegirá directamente que instancia a de crear si no que se lo especificara mediante un parámetro, esto hace que no se esté estrictamente hablando de el patrón *factory method* pero es muy similar, una mayor información sobre este patrón puede ser encontrada en el libro sobre Patrones[2].
En el caso de la fabrica para la capa también se hará otra excepción pues se brindara otro método para obtener la fabrica de sentencias a modo de atajo, evitando la necesidad de tener que importar explícitamente otra clase más.

Capítulo 6

Implementación

Una vez que la arquitectura del proyecto fue definida es necesario bajar el nivel de abstracción para obtener una vista más detallada de las diferentes clases y sus herencias pues es necesario empezar a lidiar con los detalles propios del lenguaje ya que si bien se estuvo pensando desde un API propio del lenguaje se lo estuvo analizando de forma muy abstracta. En este capitulo se analizaran los detalles que fueron surgiendo a la hora de escribir el código de *JDBGM* y como se los encaro.

6.1. Implementación de la capa de acceso a el motor

A primera vista la implementación de esta interfaz aparenta ser bastante sencilla, en especial por que el conjunto de funciones que define esta pensado para que sea sencillo de utilizar y de recordar, pero el problema radica precisamente en mantener esta simplicidad. Dos son lo principales problemas: el manejo de las excepciones y el correcto manejo de los recursos, en este caso estos problemas principalmente se presentan en la clase abstracta `JDBCManager` que es la que implementara la interfaz definida en base a *JDBC*.

6.1.1. Manejo de las excepciones

Como ya se explico prácticamente todas las funciones definidas en *JDBC* pueden lanzar excepciones del tipo `SQLException`, que cuando ocurren deben ser desviadas de nuevo por el API que se esta definiendo para que estas sean manejadas por el programador, entonces el problema no es volver a lanzar la misma excepción que se puede hacer de manera muy sencilla marcando la función donde estén ocurriendo las excepciones con la palabra reservada `throw` que indica que la función puede lanzar errores, tantos como se declare después de dicha palabra reservada. El mayor problema surge más bien a la hora de brindar información extra sobre donde y por que ocurrieron las excepciones, puesto que existe más de un motivo para que una función lance una excepción, por ejemplo cuando se llama a el método `Statementet.executeQuery()` este puede lanzar excepciones por lo menos por dos causas: por que la sentencia SQL que se le envío es invalida o bien por que no es posible conectarse con la base de datos, lo que puede ocurrir por ejemplo por un error en la red sobre la que se comunicaba con la base de datos. Para clarificar un poco este asunto se analizara un ejemplo real extraído de `JDBCManager`.

función extraída de JDBCManager

```
1  public abstract class JDBCManager implements GenericManager{
2
3          String message1 = "la conexion no fue inicializada o fue cerrada";
4          String message2 = "problema mientras se ejecutaba la sentencia: ";
5
6          public int update(String sql) throws JDException{
7                  if ( !connectionStarted )
8                          throw new JDException(message1, null);
9                  Statement stat = null;
10                 int result = -1;
11                 try {
12                         stat = connection.createStatement();
```

```
13                          result = stat.executeUpdate(sql);
14                  } catch (SQLException e) {
15                          rollbackIfTransaction(message2 + sql);
16                          throw new JDException(message2 + sql, e);
17                  } finally {
18                          try {
19                                  stat.close();
20                          } catch (SQLException e) {
21                                  throw new ConnectionIssueException(e);
22                          }
23
24                  }
25          return result;
26          }
27  }
```

La función `update(String)` que se muestra en este extracto de código se corresponde con una función privada que se usa para implementar la función `update(UpdateStatement)` que se define en la interfaz `GenericManager`, esta función esta encargada de brindar un modo para ejecutar las sentencias SQL que no son del tipo `SELECT`, lo primero para notar son las variables `message1` y `message2` cuya única función es la de almacenar mensajes de información para cuando ocurran las excepciones, después ya dentro de la función se puede notar que el primer control es independiente de los métodos de *JDBC* y lanza una excepción propia cuyo único contenido es el mensaje de error que corresponde cuando no se a inicializado la conexión con el DBMS mediante el método adecuado, después si ya se usa explícitamente los métodos de *JDBC* donde se intenta crear un objeto `Statement` y a partir del mismo ejecutar la sentencia que fue pasada como parámetro. Ambas acciones pueden terminar con una excepción por lo cual deben ser encerradas en un bloque `try/catch` el cual sirve para manejar excepciones pero lo único que se hace en este caso es envolver la excepción `SQLException` dentro de la excepción propia de *JDBC* la cual como se ve en su constructor `JDException(String, SQLException)` recibe una excepción de dicho tipo esto es así por que la única información coherente que se puede dar sobre la excepción en la función es que ocurrió un error mientras se estaba intentando ejecutar la sentencia pero por debajo puede estar ocurriendo un error de conexión con la BD o el uso de una sentencia mal formada en cuyo caso el programador deberá indagar sobre esto en la excepción que se esta envolviendo. El método `rollBackIfTransaction(String)` cuya declaración también es privada es usado cuando el método `update(String)` es llamado durante una transacción y puesto que de haber entrado en `catch{}` quiere decir que ocurrió una excepción entonces se deben volver a el momento en el que se marco el inicio de la transacción mediante `Connection.rollback()` método que es usado dentro de `rollBackIfTransaction()`, y por supuesto este método también puede terminar con una excepción que de ocurrir sera lanzada por `rollback()` la cual sera envuelta en un objeto `JDException` y sera lanzado por la función deteniendo su ejecución ya que `catch{}` no captura errores.
Un manejo similar ocurre con los otros métodos implementados, siempre se trata de capturar las excepciones producidas en los métodos subyacentes para brindar información sobre el momento en que se producen estas excepciones en *JDBGM*, a veces para ello anidando bloques `try/catch` para poder discernir sobre el motivo y elegir el mensaje adecuado puesto que una vez que se declara que la función lanza excepciones de un dado tipo se pueden obviar los bloques `try/catch` para ese tipo de excepciones, pudiendo volverse muy confuso en ese caso el origen de la excepción.

6.1.2. Manejo de los recursos

Como *JDBC* se trata de acceder a recursos externos, no es suficiente con manejar las excepciones que puedan ocurrir en el uso de el API subyacente puesto que se a de tener cuidado con la disponibilidad de los recursos externos los cuales como siempre son limitados y a veces costosos de obtener, entonces para simplificar el API que se esta desarrollando es necesario que esta administración de recursos sea lo más transparente posible para el programador. Para analizar como se encaro este manejo de recursos se utilizara el mismo extracto de código que recién se utilizo:

función extraída de JDBCManager

```
...
        public int update(String sql) throws JDException{
                if ( !connectionStarted )
                        throw new JDException(message1, null);
                Statement stat = null;
                int result = -1;
                try {
                        stat = connection.createStatement();
                        result = stat.executeUpdate(sql);
                } catch (SQLException e) {
                        rollbackIfTransaction(message2 + sql);
                        throw new JDException(message2 + sql, e);
                } finally {
                        try {
                                stat.close();
                        } catch (SQLException e) {
                                throw new ConnectionIssueException(e);
                        }

                }
        return result;
        }
...
```

El primer y principal recurso es la conexión con la base de datos representado con un objeto `Connection` el cual es el más costoso de obtener en cuanto a tiempo, *JDBC* intentara mantener una única conexión disponible la que a pesar de haberse obtenido exitosamente puede llegar a morir si es que ocurre un determinado tiempo de inactividad superior al *timeout* establecido en el DBMS, transcurrido este tiempo limite es el propio motor el que cierra la conexión pero *JDBC* no puede saber cuando sucede esto por lo que el objeto `Connection` queda en un estado inconsistente ya que para el objeto la conexión sigue viva, para solucionar esto se estableció un contador interno que controla un propio *timeout* transcurrido el cual al intentarse usar la conexión se revisa si es que la conexión sigue viva, si la conexión ya no sirve se deberá crear una nueva para poder continuar operando normalmente.
Una vez obtenida la conexión el otro objeto que representa uso de recursos del motor son las instancias de `Statement`, en este caso se debe analizar dos casos separados. El primer caso se da cuando se usa el método `Statement.executeUpdate()` el cual devuelve como resultado un valor del tipo `int` que indica la cantidad de filas que fueron afectadas por la sentencia, en este caso resulta sencillo administrar los recursos usados ya que una vez ejecutada la sentencia el valor devuelto no significa uso de recurso alguno del motor por lo que se pueden liberar los recursos consumidos por `Statement` mediante `close()` lo que en el método `update()` se hace en la sección `finally{}` del bloque `try/catch` que se ejecuta si o si al final de la ejecución del bloque, esto implica que cada llamada a `update()` crea su propio `Statement` el cual después de haber sido usado es eliminado. En el segundo caso la situación es un poco más compleja por que no se pueden disponer de los recursos usados por `Statement`, esto ocurre cuando se llama a `Statement.executeQuery()` el cual devuelve un objeto del tipo `ResultSet` que sirve para obtener los datos de la consulta que se acaba de realizar, sucede que dicho objeto esta fuertemente relacionado con el objeto `Statement` del cual fue creado y si se lo cierra mediante `close()` el objeto `ResultSet` también perderá conexión con el motor.

función extraída de JDBCManager

```
...
        public ResultSet query(String sql) throws JDException{
                if ( !connectionStarted )
                        throw new JDException(message1, null);
                Statement stat = null;
                ResultSet resultset = null;
                try {
                        stat = getConnection().createStatement();
                        resultset = stat.executeQuery(sql);
                } catch (SQLException e) {
                        rollbackIfTransaction(message2 + sql);
                        throw new JDException(message2 + sql, e);

```

```
14                  }
15                  return resultset;
16          }
17  ...
```

La porción de código anterior se corresponde con el método `query(String)` que análogamente a `update(String)` sirve para ejecutar únicamente sentencias del tipo `SELECT` y como se puede ver existen dos diferencias claves: se usa `Statement.executeQuey()` en vez de `Statement.executeUpdate()` y no existe el elemento `finally{}` en el correspondiente bloque `try/catch` puesto que no se pueden disponer de los recursos consumidos por el correspondiente objeto `Statement` sin indirectamente disponer también de el objeto `ResultSet` que resulta de la ejecución de la consulta. Dicho esto, es este el único caso en el que el programador deberá explícitamente hacerse cargo de disponer de los recursos del objeto producto de la consulta mediante `ResultSet.close()`. Al igual que con `update()` por cada llamada al mismo se creara un `Statement` diferente lo cual es un comportamiento esperado puesto que si se quiere realizar más de una consulta a la vez es necesario crear un nuevo `Statement` ya que dada la estrecha relación de este con `ResultSet` solo es posible que exista una relación de uno a uno entre ellos. Como aclaración final sobre el manejo de los recursos cabe aclarar la importancia de ir "cerrando" los diferentes objetos puesto que si bien *JDBC* en su especificación aclara que al llamar al método `Connection.close()` se cerrara cualquier objeto `Statement` y `ResultSet` que se haya creado sobre el mismo liberando así los recursos consumidos por los mismos, esto no siempre es cierto puesto que los proveedores de los drivers *JDBC* a veces no manejan correctamente los recursos al cerrar la conexión, lo que puede llevar a problemas con conexiones futuras a el mismo DBMS o incluso a uso excesivos de memoria al no cerrarse adecuadamente dichos objetos.

6.2. El patrón *factory method*

La implementación de este patrón es bastante sencilla, con la salvedad de que no se esta frente a un caso puro de este patrón como ya se explico en el capitulo 5 de diseño, de todos modos para implementar este patrón se creo la clase `ManagerFactory` ya que se precisara al menos un método más para obtener la `fabrica` de el manejador de sentencias, aunque según el patrón se debería haber puesto en la clase que implemente `GenericManager` que en este caso viene a ser `JDBCManager` puesto que es la clase base para las subclases que precisan del patrón. El patrón en si como indica su nombre se implementa en un método y es por esa razón que usualmente se lo pone en la clase "Padre", en este caso el método es `ManagerFactory.getManager()` que si se estuviera frente a un caso puro para este patrón no debiera recibir ningún parámetro pues el mismo seria el encargado de decidir que instancia a de crear, pero en este caso si los recibe (por eso no es puro) y son los siguientes:

- `vendor:` establece el DBMS con el que se esta por trabajar.
- `user:` el nombre de usuario para conectarse a la base de datos.
- `password:` la contraseña del usuario para conectarse a la base de datos.
- `location:` la URI para conectarse a la base de datos.

Como se ve de estos parámetros el primero establece el tipo de `Manager` que se a de instanciar y los demás establecen los parámetros necesarios para realizar la conexión a la base de datos, este método se estableció como método de clase (`static`) para que se lo pueda usar sin tener que instanciar la clase, algo que es innecesario puesto que solo se precisa llamar una vez a este método para registrar el DBMS que se esta por usar, y luego si por alguna razón se necesitara obtener nuevamente el `Manager` se definió el método sobrecargado `ManagerFactory.getManager()` que no toma ningún parámetro y devuelve el mismo objeto que se obtuvo con la llamada a el anterior método. Para que sea única la instancia de `GenericManager` esta es guardada como atributo de clase de `ManagerFactory` al igual que las constantes que definen a los diferentes proveedores de DBMS que soporta *JDBGM* de este modo además se hace accesible para cualquier clase del programa que importe esta clase.

6.3. Implementación de el manejador de sentencias

La implementación de el manejador de sentencias fue más complicada que en el caso de la capa de acceso, principalmente por la existencia de funcionalidades en común, algunas de ellas comunes para todas las sentencias y otras solo para algunas. La arquitectura base que se mostró en el capitulo de 5 de diseño declara la existencia de clases auxiliares, las cuales aparecieron recién después de varias versiones preliminares. Inicialmente las funcionalidades en común eran copiadas y pegadas en las diferentes interfaces que compartían dicho comportamiento puesto que se quería ofrecer una sintaxis lo más sencilla posible, por ejemplo las sentencias `SELECT (SelectStatement)` y `UPDATE (UpdateStatemeny)` tienen en común el uso de la restricción `WHERE` para lo cual se declaraban, entre otros muy similares pero con diferentes parámetros, los siguientes métodos en sus correspondientes interfaces:

1. `SelectStatement.addWhereCondition(String x, int comparison, int y): void`
   `SelectStatement.addWhereCondition(String x, int comparison, long y): void`
   `SelectStatement.addWhereCondition(String x, int comparison, Date y): void`

2. `UpdateStatement.addWhereCondition(String x, int comparison, int y): void`
   `UpdateStatement.addWhereCondition(String x, int comparison, long y): void`
   `UpdateStatement.addWhereCondition(String x, int comparison, Date y): void`

El método `addWhereCondition(String,int,int)` al igual que los otros estaba duplicado tanto en las declaraciones de las interfaces como en las implementaciones de las mismas en la correspondientes clases por defecto, con esto surgían varios problemas para empezar si se quería corregir algo se lo debía hacer en ambas clases por separado recordando que el comportamiento para `WHERE` es siempre el mismo, además se debía estar chequeando con los eventuales cambios que este fuera replicado en todas las implementaciones de las mismas.
Implementado así, el código escrito usando *JDBGM* resultaba bastante reducido pero a costo de tener un código no mantenible, por lo que se decidió relegar dicho trabajo a clases auxiliares, cabe aclarar que en la primera version que recién se expuso ya existía una clase `WhereClause` heredada de *crossdb* pero cuya única función a fin de cuentas era la de representar un tipo de datos complejo para ser usado a la hora de traducir a SQL. Con la nueva implementación lo que se hizo es que `whereClause` se encargue completamente del armado de la restricción, de este modo las clases que la usen no necesita saber como es que se arma una restricción `WHERE` solo necesitan saber que esta clase proveerá la restricción traducida a SQL, quedando de la siguiente manera las interfaces anteriores:

1. `SelectStatement.addWhere(): WhereClause`

2. `UpdateStatement.addWhere(): whereClause`

De este modo las funciones que se declaraban al principio pasan a ser parte de `WhereClause` y ya no se las debe declarar dentro de las interfaces, por lo tanto a la hora de armar las restricciones `WHERE` se debe trabajar sobre un objeto `WhereClause` y no directamente sobre un objeto `SelectStatement` por ejemplo, ahora para armar la restricción primero se tiene que obtener un objeto `WhereClause` lo que se hace mediante el método `addWhere()` y recién sobre el llamar al método que se desee por lo que el código resultante quedaría de la siguiente manera:

Nueva implementación para el uso de las clases auxiliares

```
1 ...
2 SelectStatement select = factory.getSelectStatement();
3 select.addWhere().andEquals(value, key);
4
5 //o bien
6
7 WhereClause where = select.addWhere();
8 where.andEquals(value, key);
9 ...
```

Como se ve en el ejemplo es necesario usar el método extra `addWhere()` para armar la restricción, ahora dependiendo de como se lo use al método se pierde o no facilidad de lectura: si se lo usa como `select.addWhere().andEquals(value, key)` la lectura del código se hace bastante sencilla en cambio si se usa una variable intermedia, `where` en el ejemplo, se puede perder fácilmente la claridad del código aunque obviamente las dos formas son totalmente validas lo único que cambia es cuan explicito es el uso de variables intermedias.
Este problema puede ser visto como una mal interpretación de la programación orientada a objetos (POO) pues es el caso de un objeto que hace uso de otro objeto "auxiliar" para poder llevar a cabo su trabajo y cuyo objeto auxiliar a su ves sirve a varios otros objetos "principales", el problema en realidad era que tan sucinto resultaría el código que se escribiría con *JDBGM*.
Lo que se pretendía evitar era el uso de funciones extras por decirlo de alguna manera, en la implementación que se termino eligiendo la función extra es `addWhere()` pues como se ve es necesario llamar a esta función para poder agregar una restricción `WHERE`, en cambio en la primera version esto se podía hacer directamente sobre el objeto principal con lo que se ganaba simplicidad pero se perdía mantenibilidad y poniendo ambos aspectos en una balanza claramente se tenia que decantar por la mantenibilidad.
Otro modo de encarar el problema era que mediante herencia múltiple, pensando siempre en que se quería evitar el uso de funciones extras, pero lamentablemente Java no admite herencia múltiple[1] y dada la arquitectura propuesta sin herencia múltiple se hacia inviable agregar las funcionalidades mediante herencia, la otra opción que si era posible pero no correcta era usar implementación de múltiples interfaces para declarar en una interfaz el comportamiento de la restricción pero como para todas las clases la implementación del comportamiento era el mismo la implementación de dichas interfaces también resultaría en una duplicación de código.

6.3.1. Facilitando la escritura de código

Como ya se comento la indecisión a la hora de implementar el uso de las clases auxiliares se debía más que nada a la necesidad de ofrecer una escritura de código lo más sencilla posible, ahora la cuestión es cuantos métodos son necesarios para alcanzar dicho objetivo? Para responder esta pregunta primero es necesario hacer una aclaración, la pregunta no se refiere a la cantidad de funciones para cubrir las funcionalidades necesarias llamémosle funciones básicas si no a aquellas que proveen un uso más sencillo de las funciones básicas las cuales serán llamadas funciones auxiliares, para ilustrar esto se puede ver el siguiente ejemplo sobre la misma clase auxiliar `WhereClause` que se venia usando, cuyo razonamiento luego puede ser extrapolado.
Debido a la naturaleza de la restricción `WHERE` una única, en realidad tres, función básica seria la necesaria para poder realizar la función principal de la restricción por ejemplo con el método `addCondition(String and_or, String column, String operator, Object value)` se cubren prácticamente todos los aspectos necesarios, pero su uso fuerza a pasarle cuatro parámetros dos de los cuales son para configurar el tipo de restricción que se esta agregando (`and_or` y `operator`) pues bien el uso de los mismo no solo aumenta la cantidad de parámetros a ser pasados a la función si no que también obligan a recordar valores validos para dichos parámetros con la correspondiente comprobación de los mismos. Ahora es cuando entran en juego las funciones auxiliares que trabajando sobre la función básica proveen un uso más acotado de la misma por ejemplo los siguientes métodos son un extracto de los que se declararon en `WhereClause`:

Extracto de WhereClause

```
1 ...
2 public void andLike(String key, String value);
3 public void orLike(String key, String value);
4 public void andNotLike(String key, String value);
5 public void orNotLike(String key, String value);
6 ...
```

Todos ellos son atajos para `addCondition()` en los que los parámetros `and_not` y `operator` son valores fijos dependiendo del método que se llame por ejemplo para el caso de `andLike()` se tiene `and_not="AND"` y `operator="LIKE"`. Como se ve estos métodos auxiliares sirven para escribir menos código y evitar el tener que recordar valores validos para algunos parámetros.
Ahora es posible retomar la pregunta cuantos métodos son necesarios? La respuesta es que depende de que tan extenso sea el abanico de posibilidades para una función, en el caso de

`WhereClause` se considera que se consiguió cubrir una importante cantidad de posibilidades pero aun así se dejo la puerta abierta para agregar las restricciones mediante otros métodos básicos puesto que siempre pueden quedar posibilidades que no se tuvieron en cuenta, por lo que en este caso más es mejor.
Aun así hay que tener en cuenta que demasiados métodos pueden representar menos código para escribir pero también muchos más para recordar por lo que hay que poner en la balanza sintético contra curva de aprendizaje, así que lo mejor es cubrir únicamente las funcionalidades más usadas con las funciones auxiliares dejando el resto para las funciones básicas.

6.3.2. Interfaces y clases abstractas

Las interfaces están echas para definir comportamiento, esto se hace cuando diferentes clases comparten las mismas funciones pero internamente estas funciones se implementan de diferente manera dependiendo de la clase a la que pertenezcan las mismas. En el caso del manejador de sentencias se esta frente a una situación similar a la expuesta pero en este caso solo alguna de las funciones deben ser implementadas de diferente manera, en este caso lo correcto o más sencillo hubiera sido crear una clase abstracta en la que los métodos que son iguales para todas las subclases que puedan existir estén implementados y aquellos que se deban implementar de diferente manera se declaren como abstractos. Esta ultima clase abstracta si existe en el manejador de sentencias pero no es explícitamente la que define el comportamiento para las sentencias si no es una interfaz la que define esto, interfaz que es implementada en parte por la clase abstracta. En el manejador de sentencias tal como indica el capitulo 5 de diseño la interfaz por ejemplo para `SELECT` es `SelectQuery` y la clase abstracta correspondiente es `DefaultSeletQuery`, la cuestión es que la interfaz termina no siendo estrictamente necesaria por lo que en un momento de la etapa de implementación se pensó en la posibilidad de dejar de lado la interfaz y quedarse únicamente con la clase abstracta pues esta podía completamente definir el comportamiento para una de las sentencias cualquiera, pero de todos modos se conservo la interfaz puesto que usando como el tipo de dato a esta y no la clase abstracta se deja abierta la posibilidad a que en un futuro se cree otra clase abstracta que implemente de diferente manera los métodos correspondientes, de una manera más eficiente quizás o directamente con controles de seguridad extra, y las subclases correspondientes podrán ser instanciadas directamente por la clase *factory* dependiendo de lo que se desee.

6.3.3. Fabricas de objetos, *abstract factory*

Al igual que en la capa de abstracción para un correcto uso del manejador de sentencia es necesario disponer de una clase *factory* que maneje adecuadamente las instancias que se deben crear. Para el manejador de sentencias la clase que implementa el patrón es `SQLFactory` que sirve tanto de base para las *fabricas* especificas como de selector de las instancias que se han de crear, el primer problema que se encontró fue como saber el tipo adecuado de clase que se debía instanciar sin tener que estar usando parámetros en `SQLFactory.getFactory()` por que obligaría a que el programador revisase cada vez que llamase a este método de pasarle el valor adecuado para dicho parámetro el cual en definitiva siempre debe ser el mismo pues se esta trabajando con un único DBMS, la solución es simple basto con agregar algunos atributos extras a `ManagerFactory` que es quien registra el tipo de DBMS con el que se quiere trabajar, dichos atributos son:

- `ManagerFactory.currentVendor` que indica el tipo de DBMS (el proveedor) con el que se esta trabajando.
- `ManagerFactory.isRegistered` que indica si algún motor fue o no registrado, se podría haber usado el parámetro anterior para revisar esto pero no resultaba tan claro y sencillo como usar un parámetro especifico para esto.

De este modo con los parámetros nuevos resulta sencillo comprobar si se registro algún DBMS y además también saber cual DBMS se esta usando sin tener que recurrir a parámetros en la función `SQLFactory.getFactory()`, además con los nuevos atributos resulta sencillo ver si ya se instancio alguna implementación de `SQLFactory` para usar siempre una única instancia.
Para las subclases de `SQLFactory` no hay ninguna apreciación puesto que lo único que deben hacer es implementar los métodos abstractos de su superclase. Pero si es necesario revisar los

constructores de los que hará uso la fabrica, bien con el patrón *abstract factory* lo que se quiere hacer es controlar la creación de los objetos, lo que se puede traducir en restringir el acceso a los constructores de las clases que maneja la fabrica. Dicha restricción se puede llevar a cabo de diferentes maneras incluso existe un patrón, *singleton*, que sirve para evitar que se pueda instanciar más de una vez una clase de modo que a lo largo del programa solo puede existir una única instancia del mismo[2], esto se puede lograr haciendo que el constructor de la clase sea privado (`private`) y reemplazando su papel por un método de clase que se encargara de instanciar una única vez a su clase y siempre devolver esa única instancia cada vez que este sea requerido, esto no es aplicable a este caso pero da la pauta para restringir el acceso a los constructores de las sentencias. Una primer idea es convertir en `private` el constructor y proveer un método estático que haga las veces de constructor pero en este caso el método, dependiendo de la visibilidad de la clase, puede ser libremente accedido por lo que a fin de cuentas no se soluciona nada, en cambio si se opta por visibilidad `default` o lo que es lo mismo no poner ningún modificador de visibilidad para el constructor, se limita el acceso a el constructor a clases del mismo paquete y como la subclase de `SQLFActory` se encuentra en el mismo paquete que las clases especificas de las sentencias este puede acceder libremente a los constructores pero ninguna clase externa a dicho paquete podrá acceder, con lo que se logra el cometido de que sea `SQLFactory` el único medio a través del cual se puedan obtener instancias de las sentencias.

6.3.4. Implementación de cada una de las sentencias

Entrando a lo que es la implementación en especifico de cada una de las sentencias no hay mucho que remarcar más allá de lo señalado en la sección 4.3 del capitulo de Diseño que indica cuales son las clases auxiliares que usa cada una de las sentencias, salvo por el caso de algunas sentencias que no precisan de redefinir el método `toString()` que traduce la sentencia a SQL puesto que la sintaxis soportada es tan sencilla que es perfectamente entendible por los tres motores sin tener que realizar ninguna modificación, aun así es forzosa la existencia de una subclase especifica puesto que las clases base (las superclases) son abstractas para precisamente forzar este comportamiento, un ejemplo de sentencia que no redefine el dicho método se puede ver en el caso de `DefaultDeleteQuery` que quedo de la siguiente manera:

Extracto de la clase DefaultDeleteQuery

```
1  public abstract class DefaultDeleteQuery implements DeleteQuery {
2  ...
3          public String toString() {
4                  String ret = "DELETE␣FROM␣" + table;
5                  if (wclause != null) {
6                  ret += wclause.toString();//" WHERE ";
7                  }
8                  return ret;
9          }
10 }
```

Como se ve en el extracto de código el método `toSring()` se implementa dejando la clase abstracta sin ningún método abstracto, aunque esto no se ve es así, pero no hay problema la clase puede seguir siendo abstracta sin importar que no tenga métodos abstractos, lo que se logra es que esta clase no pueda ser instanciada directamente obligando a que exista una subclase, por ejemplo en el caso de `MySQLDeleteQuery` se puede ver la situación recién descripta:

Código completo de MySQLDeleteQuery

```
1  public class MySQLDeleteQuery extends DefaultDeleteQuery {
2
3          public MySQLDeleteQuery(Formatter formatter) {
4                  super(formatter);
5          }
6  }
```

Se ve que lo único que hace esta clase es heredar de `DefaultDeleteQuery` y declarar el constructor que recibe como parámetro a un objeto `Formatter` necesario para pasárselo a su superclase, esto es así por que a pesar de que se podría crear un nuevo objeto de este tipo in situ del modo `super(new MySQLFormatter());` esto hubiera significado que cada vez que se estuviera creando un nuevo objeto `MySQLDeleteQuery` se tendría que crear inútilmente por debajo otro objeto

más, puesto que `Formatter` es un conjunto de métodos de ayuda que pueden ser rehusados sin ninguna complicación y dado que es mucho más caro crear un nuevo objeto que usar una referencia ya existente a el mismo, la creación repetida de este objeto implica un desperdicio de recursos.
De todos modos el programador tampoco debe preocuparse por elegir la implementación adecuada de `Formatter` que se debe usar, por ejemplo para `MySQLDeleteQuery` es necesario usar una instancia de `MySQLFormatter`, por que la clase *factory* sera la encargada de elegir la correcta implementación a usar y además asegurarse de que sea una única instancia de `MySQLFormatter` la que se esté usando al pasarle a todas las peticiones de creación de objetos el mismo objeto `formatter`.

6.4. Pruebas

A medida que se iba implementando el proyecto se fueron realizando las correspondientes pruebas unitarias, las pruebas unitarias son aquellas que se realizan en la mínima unidad que pueda ser probada, lo que usualmente se traduce en POO en clases que deben ser probadas de manera aislada. Al tratarse de una librería es difícil realizar pruebas muy complejas por lo que la mayoría de las pruebas se resume en pruebas unitarias, en este caso algunas de las pruebas unitarias terminaron siendo pruebas de integración (entre las diferentes clases) con lo que se cubrió el funcionamiento básico del sistema, de todos modos una aplicación de ejemplo sera entregada como parte del proyecto que servirá a dos propósitos, el demostrar una aplicación practica a modo de tutorial y como prueba final que mostrara el paquete funcionando.
Estas pruebas se realizaron con la ayuda de la herramienta Junit que viene integrada con el entorno de desarrollo Eclipse, Junit[1] es una excelente herramienta para pruebas unitarias que integra una interfaz visual en eclipse para estudiar el resultado de las diferentes pruebas a las que es sometido el código de las clases, un punto muy fuerte de las pruebas unitarias es que una vez que están bien definidas resulta realmente sencillo ver cual es el impacto de un cambio o corrección en el código que se viene desarrollando.

6.4.1. Pruebas en el manejador de sentencias

Las pruebas unitarias dentro del manejador de sentencias fueron mucho más precisas ya que cada una de las sentencias debía ser probada por separado si o si puesto que estas eran independientes entre si, salvo por que algunas compartían el uso de clases auxiliares, pero que también poseen pruebas unitarias independientes, estas pruebas fueron realizadas frente a sentencias correspondientes a cada una de las sentencias en su respectivo dialecto. Otras clases como la de fabrica no tienen pruebas unitarias puesto que son extremadamente sencillas y su uso se comprueba en otras clases unitarias que hacen uso de dichas fabricas.
De todos modos no se puede asegurar que el proyecto este libre de errores de codificación y más aun de lógica puesto que la sentencias presentan una enorme cantidad de combinaciones posibles para sus parámetros, pero si se puede asegurar que se esta entregando un producto utilizable.

6.4.2. Pruebas en la capa de abstracción

Las pruebas en la capa de abstracción fueron un poco más integradoras pues para hacer uso de esta era necesario crear sentencias a ser enviadas a el motor por lo que era necesario crear instancias de las clases definidas en el manejador de sentencias de modo que también se ponía a prueba el correcto funcionamiento de las clases especificas de las sentencias. Al ser necesario el uso de las otras clases la capa solo se podía empezar a probar una vez que al menos una de las sentencias haya sido implementada y probada.

[1] Una descripción más completa puede verse en su pagina web http://junit.org/

Conclusiones

Después de analizar, diseñar e implementar queda como ultima tarea la de exponer la experiencia que resulto de todo el trabajo echo, como introducción se puede destacar que se tomo verdadera conciencia de lo que implica analizar un problema antes de iniciar con un proyecto aun cuando este no sea muy grande.

6.5. Resultado de el proyecto

JDBGM es un paquete "liviano" tanto en tamaño, pesa aproximadamente $69kb$ cuando es empaquetado, como en términos de carga de CPU, pero a pesar de este "liviandad" cumple con todas las necesidades que se plantearon inicialmente en el proyecto de una manera bastante sencilla de utilizar para todo aquel que tenga un conocimiento básico de SQL[2]. Además de esto tiene mínimas dependencias de otras librerías requiriendo tan solo de *JDBC* y del driver correspondiente a el motor que se quiera utilizar. Para empezar a usarlo solamente requiere que se le informe de una URI hacia la base de datos que se quiere utilizar junto con los credenciales de acceso correspondientes y ya es posible empezar a hacer consultas o actualizaciones sobre dicha base de datos[3]. El paquete se encargara por detrás de lidiar con *JDBC* para evitar algunas tareas repetitivas a el programador y además provee una sintaxis propia implementada en el API del manejador de sentencias que abstrae el uso de dialectos SQL específicos de los motores que están soportados.
La liviandad que se busco tanto en tamaño físico como de carga extra de recursos tiene también su lado discutible, puesto que simplicidad puede traducirse en menos opciones tanto de uso como de configuración:

- ***JDBC***: en orden de ocultar el uso de *JDBC* y de proveer una interfaz mucho mas simple se ocultan muchas de las funcionalidades del mismo, pero también muchas de estas posibilidades están fuera del alcance actual de este proyecto.

- **Manejo de los datos extraídos**: dado que *JDBC* ya provee una manera independiente del *DBMS* que se este utilizando para acceder a los datos que se obtuvieron de una consulta este aspecto no es cubierto directamente por *JDBGM* si no que utiliza `ResultSet` que es la interfaz que provee *JDBGM*, por lo que es necesario conocer el uso de esta interfaz para poder utilizar los datos extraídos desde el *DBMS*.

- **Control de errores**: en el manejador de sentencias solo se controlan los errores mas triviales que puedan ocurrir cuando no se respeta la sintaxis de creación de sentencias, esto evita controles redundantes puesto que el DBMS al recibir la sentencia vuelve a controlar los errores. En el lado de la capa de abstracción de *JDBC* los errores no deben ser manejados si no que deben ser relanzados, en este sentido la depuración del programa puede resultar mas compleja que si se utilizara únicamente el API de Java.

- **API simplificada**: hay demasiados configuraciones posibles para las sentencias SQL por lo que no se cubren algunos aspectos principalmente con el uso de funciones, hay tantas diferencias entre las mismas que no es factible entregar un API que las contemple a todas ellas sin caer en la dependencia de un motor en especifico.

[2] Pues los métodos fueron nombrados acorde a dicho lenguaje
[3] Siempre que el motor con el que se quiere trabajar este soportado

- **Sintaxis genérica**: la sintaxis en común que se definió entre los tres motores a los que se le esta dando soporte limita mucho las capacidades de los diferentes motores, esto debido principalmente a la presencia de *SQLite* entre los motores puesto que este es un motor minimalista que no tiene muchas de las características que los otros si, aun así algunas de estas características pueden ser implementadas si se las "programa" por cuenta propia pero esto ya implicaría que se debe realizar trabajo extra para migrar de motor y por lo tanto se perdería independencia.

6.6. Conclusiones generales

Mas allá de que el principal objetivo de este trabajo sea demostrar los conocimientos adquiridos sobre programación durante el cursado de la carrera de Computador Universitario, es inevitable caer (pensando exclusivamente en un programador) en las tareas de análisis y diseño debido a que el proyecto es relativamente grande y es en si una tarea necesaria en el desarrollo de cualquier software, pero que a veces se hace de manera implícita si es que la aplicación es lo suficientemente pequeña. Y fue ese el problema inicial al desarrollar este proyecto, no considerar el tamaño del proyecto y por lo tanto no tomar real conciencia de la importancia del análisis y posterior diseño sino hasta que se llego al punto en que se tenia una estructura básica funcional pero no se sabia que faltaba o si esta se ajustaba a las necesidades del proyecto debido a que no se poseía total conocimiento de los requisitos por que el análisis inicial fue muy rudimentario o como anteriormente se expreso fue una tarea que se realizo de manera implícita. Además en medio de ese desarrollo inicial se había incluido una librería externa que servía como base para el desarrollo de el manejador de sentencias, dicha inclusión ayudo también a evitar realizar explícitamente tareas de diseño pues de ella se tomaron las ideas bases del diseño inicial. Entonces cuando ya no se supo que faltaba fue que se tomo en cuenta la importancia de la necesidad del análisis y diseño, y mas en este tipo de desarrollo donde se deben contemplar muchas pequeñas características que no pueden ser descubiertas a menos que se la explicite y mas aun por que para obtener los requisitos del sistema es necesario estudiar documentación existente de diferentes fuentes.
El párrafo anterior trata en definitiva de un problema de metodología de desarrollo mas específicamente de la inexistencia inicial de metodología de desarrollo aplicada a el proyecto, principalmente por lo nula experiencia con este tipo de metodologías y secundariamente por que no se exigía uso de metodología alguna aunque claro se lo puede tomar como un requisito implícito. Como principal experiencia de este aspecto del desarrollo se concluye que es un requisito necesario que el programador además de tener experiencia en el manejo de un lenguaje debe tener experiencia en metodologías de desarrollo, y lastimosamente al parecer de este estudiante solo se toma real conciencia de esto cuando se esta en frente a proyectos relativamente grandes con limitado tiempo de desarrollo[4].
Mas allá de los problemas de análisis, diseño o implementación que se comentaron el mayor desafío que se enfrento en el desarrollo del proyecto fue la dificultad al decidir como diseñar las clases en si[5] más que la arquitectura que conformaban entre ellas pues es muy distinto escribir código que sera usado para escribir más código, que código que no sera directamente utilizado, por ejemplo se podrían obviar las buenas practicas a la hora de implementar un sistema eminentemente visual o mejor dicho una herramienta para uso final, un programa cualquiera, pues el usuario no ve directamente el código, para el esta todo bien si es que la interfaz responde o se adecua a sus necesidades, pero a la hora de escribir una librería por ejemplo es la sintaxis que se usa, los métodos que se ponen a disponibilidad, la facilidad de uso que se da la que ponen en juego la utilidad o complacencia del usuario programador frente a el trabajo realizado. Este echo demostró la importancia de seguir estándares (convenciones o buenas practicas) a la hora de desarrollar pues ello hace que sea más amigable la lectura, mantenibilidad y en este caso usabilidad del código que aunque es interpretado por maquinas debe ser desarrollado, al menos por ahora, por humanos.

[4] Aunque para este proyecto los tiempos eran manejados por el propio alumno.
[5] ver la sección 6.3.1 de el capitulo de Implementación

Finalmente el trabajar en este proyecto profundizo los conocimientos que se tenían sobre JSE[6] y además sirven como un futuro puntapié para adentrarse en el mundo de JEE[7], recordando que ambos son SDK de Java pero orientados a sectores diferentes. El estudio de los diferentes motores que soporta este proyecto también sirven como punto de partida para una mayor profundización en el uso o elección de los mismos en proyectos futuros. Y como ultima y más importante experiencia se rescata que más allá de lo aprendido específicamente sobre las tecnologías utilizadas también se aprendieron muchos conceptos abstractos que luego se pueden extrapolar a otras situaciones, como por ejemplo el uso de patrones de desarrollo y un mayor conocimiento y experiencia en POO.

[6] *Java Standard Edition*
[7] *Java Enterprise Edition*

Trabajo Futuro

El resultado final del trabajo contempla en mayor o menor medida todos los requisitos que se plantearon inicialmente, de todos modos existen aspectos a mejorar y corregir que no fueron encontrados o tenidos en cuenta en las pruebas que se realizaron, para ello en un futuro cuando la librería sea utilizada para desarrollar sera posible pulir más las funcionalidades disponibles, o si es requerido implementar soporte a una mayor cantidad de motores, algo que en definitiva puede ser desarrollada por un tercero quedando como trabajo unificar ese trabajo con la distribución principal del proyecto.
En cuanto a la ampliación de funcionalidades puede ser importante implementar soporte para concurrencia en donde la librería se deba enfrentar a múltiples peticiones desde diferentes hilos (o incluso clientes en un modelo cliente servidor) entrando más en el terreno de JEE que podría ser un lanzador de la librería para su uso dado que java es más popular con JEE que con JSE.

Apéndices

Apéndice A

Sintaxis y convenciones utilizadas

Este informe utiliza las siguientes convenciones:

- Para código tanto de sentencias SQL como de Java se utiliza `tipografía monoespaciada`.
- Para palabras en ingles se utilizan *itálicas*, estas usualmente se utilizan para nombrar marcas o conceptos que se entienden mejor por su nombre original.
- Para la definición de las sentencias SQL una sintaxis cuya definición se puede ver a continuación.

A.1. Sintaxis utilizada para definir las sentencias SQL

Esta sintaxis se basa en la sintaxis vista en algunos extractos de la sintaxis usada para definir el estándar SQL, además es similar a la sintaxis usada por *PostgreSQL* y *MySQL* en su documentación oficial. A continuación la definición de la sintaxis:

1. `<nombre etiqueta>` La etiqueta es el componente principal, sirve para definir elementos y nombrarlos.
2. `::=` El símbolo "dos puntos dos puntos igual" sirve para asignar una definición a la etiqueta, la definición de una etiqueta puede estar compuesta por otras etiquetas y de los componentes que se describirán a continuación.
3. `|` La barra vertical sirve para declarar alternativas para un componente, por ejemplo `<tag> ::= <tag 1> | <tag 2>` indica que `<tag>` debe tomar uno y solo uno de los valores definidos por `<tag 1>` y `<tag 2>`.
4. `[]` Los corchetes sirven para indicar componentes opcionales.
5. `{}` Las llaves sirven para indicar componentes obligatorios cuando sea preciso resaltar dicha obligatoriedad.
6. `...` Los tres puntos o puntos suspensivos indican que un componente se puede repetir indefinidamente.
7. `NOMBRE` Se pueden utilizar palabras en mayúsculas para indicar palabras reservadas de SQL como por ejemplo `SELECT`.

Los componentes recién nombrados pueden ser mezclados libremente, por ejemplo una lista de alternativas encerradas entre corchetes (`[<elem> | <elem> | <elem>]`) indica que se puede elegir o no una de las alternativas dadas, en cambio si estuviera encerrada entre corchetes (`{<elem> | <elem> | <elem>}`) indica que la elección de una de las alternativas es obligatoria. El siguiente es un ejemplo tomado del capitulo 5.

```
CREATE [ TEMP | TEMPORARY ] TABLE <database name> <dot> <table name>
[ IF NOT EXISTS ] <table contents source>

<table contents source> ::=
  <left paren> <table element> [ { <comma> <table element> }... ] <right paren>
| AS <select stmt>
```

En el se puede ver el uso de palabras reservadas y un ejemplo de uso de los puntos suspensivos. `[ { <comma> <table element> }... ]` este código significa que el componente `<comma> <table element>` es opcional, pero de existir debe estar compuesto por las dos etiquetas encerradas entre las llaves y lo que ellas significasen. Los puntos suspensivos indican que el elemento opcional puede estar repetido tantas veces como se desee.

Apéndice B

Manual de usuario

JDBGM es sencillo de utilizar, para empezar es necesario contar con *JDBC* que usualmente esta disponible por defecto en el SDK de JSE[1] y el driver del motor que se pretenda utilizar aparte claro de contar con *JDBGM*. Una vez que se tienen a disposición las dependencias nombradas es necesario hacerle conocer al compilador la ubicación de los archivos necesarios[2].

B.1. Primeros pasos

Para empezar a usar el paquete es necesario importar algunas librerías, el siguiente seria un ejemplo típico de las librerías necesarias.

Librerías a importar

```
1  import java.sql.ResultSet;
2  import java.sql.Types;
3
4  import com.crossdb.sql.Column;
5  import com.crossdb.sql.CreateTableQuery;
6  import com.crossdb.sql.InsertQuery;
7  import com.crossdb.sql.SQLFactory;
8  import com.crossdb.sql.SelectQuery;
9  import com.nelsonx.jdbgm.GenericManager;
10 import com.nelsonx.jdbgm.JDException;
11 import com.nelsonx.jdbgm.ManagerFactory;
```

Como se ve se esta importando por un lado a el manejador de sentencias que se ubican dentro del paquete `com.crossdb`, la capa de abstracción dentro del paquete `com.nelsonx.jdbgm` y por ultimo algunas clases de *JDBC*. Esto se podría hacer de una manera resumida utilizando el símbolo asterisco como se ilustra a continuación:

Librerías a importar forma resumida

```
1  import java.sql.ResultSet;
2  import java.sql.Types;
3
4  import com.crossdb.sql.*;
5  import com.nelsonx.jdbgm.*;
```

Pero de este modo se importan todas las clases incluidas dentro de un paquete en particular, lo que puede significar que se estén importando clases que no sean necesarias, de todos modos esta elección queda en manos del programador. Un detalle importante a notar es que en ningún momento se están importando los paquetes con las implementaciones especificas de las sentencias, esto debido a que se esta usando el patrón *Abstract Factory* el cual hace transparente la elección de las implementaciones especificas de las sentencias, a modo informativo se señala que las implementaciones especificas de las sentencias se encuentran en los siguientes paquetes:

- Para *PostgreSQL* en `com.nelsonx.postgre`

[1] Y por supuesto también en JEE.
[2] No se dará una explicación de como hacer esto pues no es la intención del manual enseñar el uso de Java.

- Para *MySQL* en `com.spaceprogram.sql.mysql`
- Para *SQLite* en `com.nelsonx.sqlite`

A continuación lo que se debe hacer es registrar el motor que se quiere utilizar tal como se ilustra a continuación:

Registrando el motor que se quiere utilizar.

```
public void test() throws JDException{

                GenericManager manager = ManagerFactory.getManager(
                                ManagerFactory.MYSQL_DB
                                , "user"
                                , "localhost/test"
                                , "password");
                SQLFactory sentencesFactory = ManagerFactory.getSQLFactory();
}
```

El método `ManagerFactory.getManager` es el que precisamente sirve para registra el motor y obtener una instancia del manejador del mismo, asignada a la variable `manager` en este caso. Los parámetros que recibe son:

- Una constante que identifica el motor que se quiere registrar y cuyos valores posibles son `ManagerFactory.MYSQL_DB, ManagerFactory.SQLITE_DB` y por ultimo `ManagerFactory.POSTGRE_DB`. De agregarse soporte para más motores se deben agregar sendas constantes identificativas.
- Un nombre de usuario con acceso a la base de datos.
- Una URI hacia la ubicación de la base de datos.
- La contraseña del usuario que se le proporciono en el parámetro anterior.

De ser los parámetros correctos ya se podrá empezar a interactuar con la base de datos señalada por la `URI` que se le dio al método. Es importante notar que el método que se esta usando de ejemplo relanzara los errores que puedan ocurrir en cualquiera de los métodos que se están usando de *JDBGM*, de otro modo se tendría que haber utilizado los bloques `try/catch` para manejar los mismos.
Como ultimo paso antes de poder empezar a trabajar sobre el motor es necesario obtener una fabrica de sentencias, para lo cual se utiliza el método `getSQLFactory` que es un "acceso directo" al método de clase que brinda `SQLFactory`.

B.2. Construyendo sentencias

Una vez obtenida la fabrica de sentencias, las sentencias pueden empezar a crearse a partir del objeto fabrica, siempre se debe utilizar las interfaces genéricas de las sentencias como tipos de datos de modo de que no se dependa de una implementación en especifico de la sentencia, esto se puede observar en el siguiente ejemplo:

Uso de la fabrica de sentencias

```
CreateTableQuery create = sentencesFactory.getCreateTableQuery();
InsertQuery insert = sentencesFactory.getInsertQuery();
UpdateQuery update = sentencesFactory.getUpdateQuery();
AlterTableQuery alter = sentencesFactory.getAlterTableQuery();
SelectQuery select = sentencesFactory.getSelectQuery();
```

En el extracto de código anterior se obtuvieron objetos que representan a cada una de las sentencias, a estas se las puede empezar a poblar de datos para después obtener las sentencias SQL que se deseen, a continuación un ejemplo para `insert`:

Uso de una sentencia INSERT

```
InsertQuery insert = sentencesFactory.getInsertQuery();
insert.setTable("example_table");
insert.addColumn("columna_pk", 1);
insert.addColumn("columna_clave_foranea", 3);
manager.update(insert);
```

Como se ve la sintaxis ofrecida es bastante amigable con el lenguaje utilizado por SQL, para una completa referencia de los métodos disponibles para cada una de las sentencias es necesario referirse a la documentación incluida en el código fuente del proyecto. La sentencia formada con los métodos anteriores se corresponde con

```
INSERT INTO example_table
(columna_pk, columna_clave_foranea)
VALUES
(1, 3)
```

La cual es en la ultima porción del código enviada a el motor mediante el método `manager.update(insert)`, en este caso no se recupera el valor devuelto por dicha función que indica el numero de filas afectadas por la sentencia, en este caso seria 0 por que insert agrega una fila, no altera ninguna. Este numero puede ser utilizado para saber si la consulta afecto o no a alguna fila para una consulta `UPDATE` por ejemplo.

Uso de una sentencia SELECT

```
1 SelectQuery select = sentencesFactory.getSelectQuery();
2 select.addTable("example_table");
3 select.addColumn("column_pk");
4 select.addJoin().innerJoin("anoter_table","example_table.id=anoter_table.id");
5 select.addOrderBy("column_pk");
6 ResultSet result = manager.query(select);
```

El segmento de código anterior es para una sentencia `SELECT` y la sentencia en particular que se formo es

```
SELECT column_pk
FROM example_table INNER JOIN anoter_table
ON example_table.id = anoter_table.id
ORDER BY column_pk
```

Y al igual que en el ejemplo anterior se envía también la sentencia hacia el motor pero mediante el método `query()` que es el apropiado para enviar las sentencias del tipo `SELECT`, todas las otras sentencias se envían mediante `update()`.
Siempre que se hace una consulta, su resultado debe ser obtenido a través de un objeto `ResultSet`, este objeto es parte de *JDBC* por lo que para una completa guía de su uso es necesario consultar la documentación[1] de Oracle disponible en su web. Como ya se comento para una completa referencia de todos los métodos disponibles para cada una de las sentencias es necesario referirse a su documentación, la cual esta también disponible en el código fuente de *JDBGM* en formato de `javadoc`. No se profundiza más en el uso de las sentencias puesto que los nombres de los métodos son bastante auto descriptivos.

B.3. Realizando transacciones

Independientemente de cuales sentencias se quieran enviar hacia el motor estas se pueden hacer de dos maneras, la normal que es la que se explico en la sección anterior y para ese caso si la sentencia puede ser ejecutada, no tienen errores, los cambios que ella indique son permanentes, la segunda manera es hacerlo mediante transacciones. Una transacción trata un grupo de sentencias como si de una sola se tratase, de modo que si alguna de ellas falla o es imposible de realizar se revierten todos los cambios que se estaban por realizar, quedando la base de datos en el mismo estado que estaba antes de ejecutar la transacción, si ningún error ocurre recién todos los cambios que generaría la transacción son impactados permanentemente en la base de datos, la manera de realizar esto con *JDBGM* es de la siguiente:

Realizando una transacción

```
1 manager.beginTransaction();
2         for (int i = 0; i < 10; i++) {
3                 manager.update(insert);
4         }
5 manager.endTransaction();
```

Los métodos `beginTransaction()` y `endtransaction()` demarcan el inicio y fin de la transacción, cualquier sentencia que se ejecute en medio de estas dos no impactara completamente hasta que la transacción sea finalizada, de ocurrir algún error toda la transacción, o lo que se había ejecutado de ella, sera desecha (`rollback`) y además se detendrá la ejecución del código pues se lanzara una excepción. En el ejemplo, suponiendo que el objeto `insert` represente una sentencia valida, supóngase que uno de los `insert` enviado hacia el motor no pueda ser ejecutado sobre la base de datos entonces los cambios previos dentro de la transacción serán desechos y el ciclo interrumpido pues se lanzara una excepción que debe ser o bien relanzada o capturada dentro de un bloque `try/catch`. Los métodos `GenericManager.queryAndClose()` y `GenericManager.updateAndClose()` no se pueden usar dentro de una transacción puesto que estas trabajan independientemente de la única conexión que maneja `GenericManager` por lo que no se verán afectadas por la transacción.

B.4. Liberando recursos

Con *JDBGM* el único momento en que el programador se debe preocupar en liberar recursos se da en dos situaciones:

1. Cuando se esta obteniendo datos desde un `ResultSet` puesto que este no se puede cerrar hasta que se terminen de obtener los datos del mismo y esto significa que hay un objeto `Statement` consumiendo recursos por detrás que no puede ser cerrado. Por lo tanto cuando se terminen de obtener los resultados del mismo es necesario cerrarlo mediante el método `ResultSet.close()`.

2. Cuando se considere que la base de datos ya no se requiera o bien que la base de datos no vaya a ser utilizada por bastante tiempo, se debe cerrar toda conexión con el motor mediante `GenericManager.endConection()`.

Con eso se cubren todos los aspectos básicos del uso de *JDBGM*, pero para empezar a usarlo se recomienda que antes se lea la documentación del mismo, al menos de las clases importantes como son `GenericManager, ManagerFactory, SQLFactory` y todas las clases que definen a las sentencias.

Apéndice C

Una aplicación de ejemplo

Como *JDBGM* es una librería, no se la puede mostrar directamente en funcionamiento por lo que como parte del proyecto se planteo entregar una pequeña aplicación de ejemplo. Para mantener la simplicidad de la aplicación de ejemplo se decidió realizar la misma en dos partes. La primer parte es un *script* que crea una base de datos, la segunda es una aplicación un poco más compleja que muestra los datos que se crearon con el *script* anterior.

C.1. Creando la base de datos

La primer parte de la aplicación de ejemplo es un mini-programa contenido en el fichero `MakeDB.java` el cual contiene la clase `MakeDB` que puede crear una base de datos y poblarla con datos al azar que se crean en base a una lista de nombres, direcciones, y domicilios. Los datos generados son para crear una base de datos que lleva el control de asistencia de una lista de alumnos de diferentes grados de un establecimiento educativo. La estructura de los datos generados en si no es importante, lo que interesa es demostrar como se puede usar *JDBGM* para poblar de datos una base de datos, además de crearla claro esta. EL mini-programa puede crear dicha base de datos en cualquiera de los motores que están siendo soportados por el proyecto, para ello es necesario cambiar una única linea de código. Si se observa el código fuente de esta primer parte se puede ver que se muestra una de las posibles formas de usar las clases que representan a `CREATE TABLE, INSERT` y `SELECT`.

Código que debe alterarse para elegir el motor.

```
1  // Constructor de la clase MakeDB
2  public MakeDB(String , String , String ) throws JDException {
3          ...
4          // Debe elegirse entre SQLITE_DB, POSTGRE_DB y MYSQL_DB
5          manager = ManagerFactory.getManager(ManagerFactory.SQLITE_DB
6                                              , user
7                                              , location
8                                              , password);
9          ...
10 }
```

Como ultimo comentario cabe recalcar que en aplicaciones mucho más complejas las bases de datos deben ser creadas por aparte, en primer lugar por que el soporte para `CREATE` esta restringido únicamente a `CREATE TABLE` y en segundo lugar por que el objetivo de este proyecto es cubrir las necesidades de las operaciones `CRUD` y no el de la creación de estructuras de datos.

C.2. Presentando los datos generados

La segunda parte de la aplicación de ejemplo trata de una interfaz gráfica que muestra los datos que fueron cargados en la base de datos que se creo con `MakeDB`. El foco de esta segunda parte es mostrar un uso un poco más complejo de `SELECT` y del manejo de errores.
Esta segunda parte se denomino EasyList, y como se puede ver en la figura C.1 permite elegir el nombre de usuario y contraseña con el cual se desea acceder a la base de datos. La ubicación

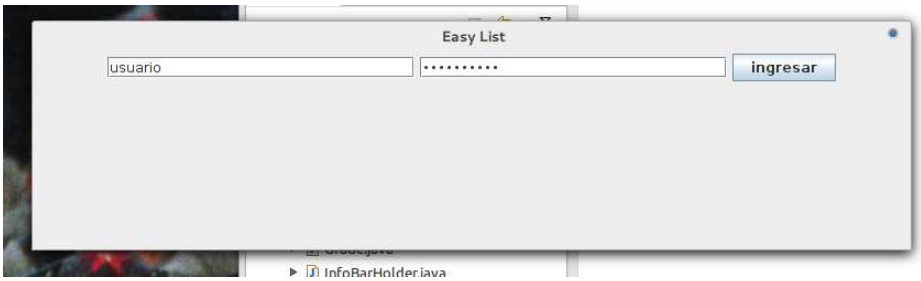


Figura C.1: EasyList pantalla inicial.

de la base de datos y el proveedor de *DBMS* que se usara deben especificarse en el código de la clase, por lo que si se pretende usar un motor o base diferente eso debe ser cambiado en el código fuente.

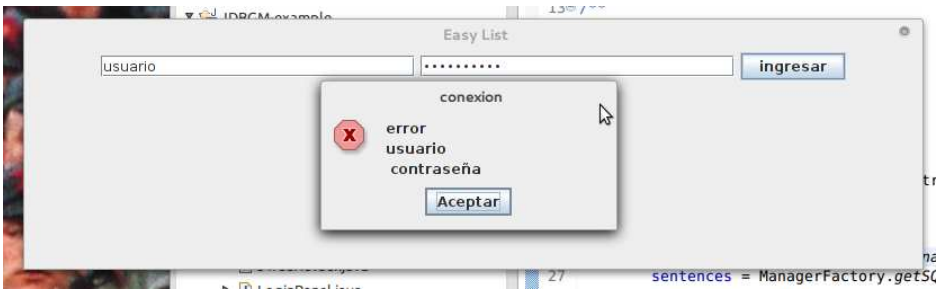


Figura C.2: Manejo de errores en EasyList

Como se puede ver en la figura C.2 se incluyo un ejemplo de manejo de excepciones para la vista inicial de *login*. Si se intenta acceder con un nombre de usuario y/o contraseña incorrecto `ManagerFactory` lanzara una excepción pues no va a poder realizar la conexión con motor, por lo que es posible para EasyList capturar esta excepción e informarle a el usuario de la aplicación de dicho error[1] sin que el programa se cuelgue, o sea que el programa posee un modo de recuperarse de algunos errores.

Figura C.3: Acceso correcto en EasyList

La figura C.3 muestra como se ve EasyList cuando el usuario pudo ingresar satisfactoriamente sus datos de autenticación. El único elemento realmente funcional de esta interfaz es el árbol de la izquierda que muestra los años que fueron registrados, y por cada año los cursos que fueron registrados por el programa y por cada uno de estos cursos se muestra los alumnos que

[1] En este caso en el mensaje de error se muestra la contraseña en texto plano, algo que siempre debe evitarse por cuestiones de seguridad.

estaban matriculados en ellos. La lógica de todas estas consultas están contenidas en la clase `DataObtainer` que es la encargada, como su nombre lo dice, de obtener todos los datos de la base de datos. Todas las otras clases que componen esta segunda aplicación son necesarias únicamente para crear la interfaz gráfica.
Al igual que en la aplicación anterior es necesario únicamente cambiar una linea de código para poder usar un *DBMS* distinto. La elección de un motor distinto se podría haber echo dinámicamente antes de que el usuario ingrese sus credenciales de acceso, pues como ya se menciono anteriormente basta con elegir una constante diferente para indicarle a `ManagerFactory` que se desea usar un motor diferente[2], pero como se quería mantener lo más simple posible la aplicación de ejemplo esto no se implemento.

C.3. Resumen

La importancia de la aplicación de ejemplo es la de mostrar la facilidad con la que se puede migrar de un motor a otro cuando se usa *JDBGM*, es por ello que las dos partes que componen el ejemplo son tan simples. Además estas clases que se crearon sirven como guía básica, aparte del manual, para comprender como se usa la librería presentada en este proyecto.
Se puede encontrar todo el código de la aplicación de ejemplo en el repositorio publico `https://github.com/nerones/JDBGM-example` alojado en los servidores de Github[3].

[2]Suponiendo que la ubicación del motor elegido sea siempre la misma que la de los otros motores
[3]Github es un popular servicio que ofrece alojamiento gratuito para repositorios git que alojen proyectos *open source*

Bibliografía

[1] JDBC Database Acces. Jdbc. http://docs.oracle.com/javase/tutorial/jdbc/, 2012. [Internet; descargado 2-febrero-2011.

[2] Steven John Metsker. *Design patterns Java workbook*. The software patterns series. Addison-Wesley, pub-AW:adr, 2002.

[3] Pagina oficial de MySQL. Datatypes in sqlite version 3. http://www.sqlite.org/datatype3.html, 2012. [Internet; descargado 2-febrero-2011.

[4] Pagina oficial de MySQL. Mysql :: Mysql 5.1 reference manual :: 12 functions and operators. http://dev.mysql.com/doc/refman/5.1/en/functions.html, 2012. [Internet; descargado 2-febrero-2011.

[5] Pagina oficial de MySQL. Mysql :: Mysql 5.1 reference manual :: 13 sql statement syntax. http://dev.mysql.com/doc/refman/5.1/en/sql-syntax.html, 2012. [Internet; descargado 2-febrero-2011.

[6] Pagina oficial de PostgreSQL. Datatypes in sqlite version 3. http://www.sqlite.org/datatype3.html, 2012. [Internet; descargado 2-febrero-2011.

[7] Pagina oficial de PostgreSQL. Postgresql: Documentation: 8.4: Functions and operators. http://www.postgresql.org/docs/8.4/static/functions.html, 2012. [Internet; descargado 2-febrero-2011.

[8] Pagina oficial de PostgreSQL. Postgresql: Documentation: 8.4: The sql language. http://www.postgresql.org/docs/8.4/static/sql.html, 2012. [Internet; descargado 2-febrero-2011.

[9] Pagina oficial de SQLite. Datatypes in sqlite version 3. http://www.sqlite.org/datatype3.html, 2012. [Internet; descargado 2-febrero-2011.

[10] Pagina oficial de SQLite. Sql as understood by sqlite. http://www.sqlite.org/lang.html, 2012. [Internet; descargado 2-febrero-2011.

[11] Pagina oficial de SQLite. Sqlite query lenguage: Core functions. http://www.sqlite.org/lang_corefunc.html, 2012. [Internet; descargado 2-febrero-2011.

[12] Sun Developer Network (SDN). Java blueprints patterns. http://java.sun.com/blueprints/patterns/index.html, 2012. [Internet; descargado 2-febrero-2011.

[13] Sun Developer Network (SDN). Jdbc. review url, 2012. [Internet; descargado 2-febrero-2011.

[14] Sun Developer Network (SDN). Lesson: Exceptions. http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html, 2012. [Internet; descargado 2-febrero-2011.