JPangolin 1.0: APLICACIÓN PARA CONSULTAR, MODIFICAR Y ELIMINAR OBJETOS EN EL LENGUAJE DE PERSISTENCIA DE JAVA JPQL.

FABIÁN FERNEY ROA PRADA

EDGAR FABRICIO SANTOS RINCÓN

UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FISICOMECÁNICAS

ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

BUCARAMANGA

2014

JPangolin 1.0: APLICACIÓN PARA CONSULTAR, MODIFICAR Y ELIMINAR OBJETOS EN EL LENGUAJE DE PERSISTENCIA DE JAVA JPQL.

FABIÁN FERNEY ROA PRADA

EDGAR FABRICIO SANTOS RINCÓN

Trabajo de grado presentado como requisito para optar al título de Ingeniero de Sistemas

DIRECTOR

JACKSON SONNY GONZÁLES
Ingeniero de Sistemas

CODIRECTOR

FIDEL DAVID JIMÉNEZ
Ingeniero de Sistemas

UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FISICOMECÁNICAS

ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

BUCARAMANGA

2014

A la luz y motivación de mi vida Ángela Valeria

A mi abuelita Mercedes que desde el cielo me ilumina

Fabián

Valar Dohaeris

Edgar

Fabián:

A Dios y la virgen, por ayudarme a vencer todas las dificultades a lo largo de este camino.

A mi padre por su inmenso sacrificio, para hacer posible este logro.

A mi madre por su constante apoyo y comprensión.

A los ingenieros Fidel David Jiménez y Jackson González, por su orientación y apoyo.

A mi compañero y amigo Edgar por su apoyo y compañerismo.

Edgar:

A Dios por haberme guiado y cuidado hasta este momento.

A mis padres Virginia Rincón, Guillermo Santos y Pablo Coronado por su apoyo incondicional.

A mi familia y a mi iglesia.

Al ingeniero Fidel Jiménez por su paciencia y compromiso, y a Fabián por su dedicación.

TABLA DE CONTENIDO

LISTA DE TABLAS

LISTA DE ILUSTRACIONES

RESUMEN

Título:

JPangolin 1.0: APLICACIÓN PARA CONSULTAR, MODIFICAR Y ELIMINAR OBJETOS EN EL LENGUAJE DE PERSISTENCIA DE JAVA JPQL.[1]

Autores:

Fabián Ferney Roa Prada, Edgar Fabricio Santos Rincón.[2]

Palabras clave:

Hibernate, ORM, persistencia, JPA, Impedancia objeto-relacional, entidad.

Descripción:

En la actualidad el paradigma orientado a objetos y el modelo relacional son ampliamente usados para la creación de aplicaciones y la persistencia de los datos respectivamente. Debido a que existe una impedancia objeto-relacional entre ambos modelos, es decir, no son del todo compatibles, existe entonces cierto trabajo extra por parte de los desarrolladores al hacer persistir y recuperar sus objetos de una base relacional debido a la coexistencia de ambos modelos en las aplicaciones.
Muchos desarrolladores han optado por la utilización de herramientas de *mapeo* objeto-relacional que faciliten el *mapeo* de datos entre una base de datos relacional y el modelo de objetos de una aplicación. Una de estas herramientas, conocidas como ORM, es Hibernate.

En el proceso de desarrollo software es de vital importancia que el desarrollador cuente con herramientas que le permitan realizar su labor en el menor tiempo posible y de manera eficiente. Esta aplicación busca facilitar la implementación de la persistencia de los datos en las aplicaciones desarrolladas con el lenguaje de programación orientado a objetos JAVA e Hibernate como implementación del API de persistencia de Java (JPA), mediante la ejecución de consultas JPQL en una base de datos relacional permitiendo al desarrollador concentrarse en la realización de las mismas sin que este proceso implique la ejecución de todo el proyecto para conocer el resultado de una consulta determinada.

Para que esto sea posible, dicha aplicación deberá ser lo suficientemente flexible para que el usuario pueda conectarse con diversos motores de bases de datos relacionales sobre los cuales podrá ejecutar consultas en el lenguaje JPQL. También es importante que la aplicación permita agregar y eliminar entidades, entre otras funcionalidades que facilitarán la implementación de la persistencia de datos en proyectos de desarrollo que utilicen las tecnologías previamente mencionadas.

---

[1] Trabajo de grado

[2] Facultad de Ingenierías Fisicomecánicas, Escuela de Ingeniería de Sistemas e Informática Dirigido por el ingeniero Jackson Sonny Gonzales; codirigido por el ingeniero Fidel David Jiménez.

ABSTRACT

Title:

JPangolin 1.0: SOFTWARE TO RETRIEVE, MODIFY AND DELETE OBJECTS USING THE JAVA PERSISTENCE QUERY LANGUAGE (JPQL).[3]

Authors:

Fabián Ferney Roa Prada, Edgar Fabricio Santos Rincón.[4]

Keywords:

Hibernate, ORM, persistence, JPA, object-relational impedance, entity

Description:

Nowadays the object oriented paradigm and the relational model are widely used to create objects and persist data respectively. Since there is an object-relational impedance mismatch between both models, that is, they are not fully compatible, there is some extra work for developers when storing and retrieving the data stored in objects because of the coexistence of both models in applications.
Many developers have decided to use object-relational mapping technologies that ease the mapping of data between a relational database and the object model of an application. One of these technologies, known as ORM (Object-Relational Mapping) is Hibernate.

In the software development process it is very important for the developer to have tools that allow them to perform their work efficiently and as soon as possible. The purpose for this application is to ease the data persistence implementation in applications that are developed using the object oriented programming language JAVA and Hibernate as an implementation of the Java Persistence API (JPA), through the execution of JPQL queries directly to a relational database. Thus the programmer will be able to focus on the execution of these queries without running the entire project to determine the result of a particular query.

In order to achieve this purpose, this application must be versatile and allow the user to connect to several relational database engines on which the JPQL queries will be executed. It is also important that this application offers to add and delete entities, among other features that will contribute to the data persistence in development projects that use the technologies mentioned above.

[3] Work Degree

[4] Physique-Mechanics Sciences Faculty, Systems Engineering School. Directed by the engineer Jackson Sonny Gonzales, codirected by the engineer Fidel David Jiménez.

INTRODUCCIÓN

El mapeo objeto-relacional (ORM) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional (haciendo el mapeo tabla-clase, campo-atributo), utilizando un motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual sobre la base de datos relacional. Esta aproximación de alto nivel reduce significativamente la cantidad de código requerido para realizar todas las operaciones en una aplicación orientada a objetos y una base de datos relacional, incrementando así la productividad en el desarrollo.

Una dificultad que nace al trabajar con ORM en java es que existen muy pocos editores y ejecutores de código JPQL, debido a que hasta ahora se está creando una especificación estándar. Esto lleva a que muchas veces la única manera de ejecutar una consulta sobre la base de datos mediante el JPQL sea con la aplicación final en tiempo de ejecución. Por lo tanto, basados en una especificación de requisitos elaborada teniendo en cuenta el problema planteado, se propone la creación de una aplicación de escrito en el lenguaje de programación Java y usando Hibernarte como implementación de la especificación JPA, con el fin de facilitar la validación de las consultas en el lenguaje JPQL que la aplicación utilizará durante su ejecución. Para ello la aplicación brindará la posibilidad de cargar todas las clases implicadas en las consultas dejando a un lado la lógica de negocio de la aplicación, permitiéndole al programador concentrarse en las clases mapeadas que representan las tablas de la base de datos relacional que soporta la aplicación. Dicha aplicación además permitirá conectarse a un conjunto de motores de base de datos relacional más usados dentro de los que soporta Hibernate. Por lo tanto se garantiza una total flexibilidad de la aplicación para que pueda ser usada en cualquier proyecto de desarrollo que utilice Java como lenguaje de programación, Hibernate como implementación de JPA (Java Persistence API) y anotaciones como el método de mapeo.

1 PRESENTACIÓN DEL PROYECTO

1.1 Descripción del proyecto

1.1.1 Objetivo general

Desarrollar una herramienta software que permita elaborar y ejecutar acciones de consulta, actualización y borrado de información en el Lenguaje de Persistencia de JAVA (JPQL).

1.1.2 Objetivos específicos

- Implementar un editor de código JPQL para crear consultas de SELECT, UPDATE y DELETE, que permita autocompletar los atributos de las entidades usadas y evidencie mediante mensajes los errores de sintaxis.

- El editor de código JPQL tendrá las siguientes funcionalidades:
  - Guardar en un archivo las consultas creadas en el editor.
  - Abrir archivos guardados. Copiar, cortar, y pegar texto en el editor.
  - Formatear y exportar las consultas para su uso directo en código Java.

- Mostrar los resultados de las consultas en una grilla, que permita ordenar los registros por cualquiera de sus campos.

- Implementar un editor de código java para la creación de objetos que serán usados como parámetros en las consultas, y que permita autocompletar los métodos de las entidades.

- Permitir al usuario incluir las entidades (tablas mapeadas) a manejar en las consultas, empaquetadas en un archivo .jar.

- Implementar un explorador de las entidades incluidas por el usuario, en el que se mostrarán agrupadas por archivo .jar en una estructura tipo árbol y se deberá mostrar los atributos persistentes de cada entidad.

- Permitir al usuario configurar una conexión con el servidor de base de datos.

- Implementar un documento en línea de ayuda sobre el manejo de las opciones de la herramienta

1.2 Justificación

1.2.1 Antecedentes y descripción del problema

Actualmente el paradigma orientado a objetos es el más usado en el desarrollo de aplicaciones (sobre el cual fueron creados lenguajes de programación como JAVA, C++, C#, entre otros), y el modelo relacional es el más usado para la construcción de las bases de datos con motores poderosos y populares como Oracle, PostgreSQL, MySQL Server, MySQL, entre otros. Mientras los lenguajes de programación orientados a objetos usan objetos para el manejo de la información, los motores de base de datos relacionales usan filas o tuplas. Es claro que la sincronización entre las dos tecnologías no es perfecto y que hay una incompatibilidad objeto-relacional entre ambas (Object-relational impedance mismatch).

Las bases de datos orientadas a objetos fueron una respuesta a este problema. El almacenamiento de la información se hace mediante objetos, los mismos que manejamos en los lenguajes de programación (haciéndolos **persistentes**) y eliminando en gran medida el problema. Esta solución, interesante en el ámbito académico no tuvo un gran impacto en el mundo empresarial. Las organizaciones llevaban años usando motores relacionales (estables y robustos) y el costo de migrar sus datos a un motor orientado a objetos (además muchos en etapa experimental) era demasiado.

Debido a que el estándar de facto para el almacenamiento de los datos en el mundo empresarial es el modelo relacional (fracasando momentáneamente las bases de datos orientadas a objetos) y que, como ya se mencionó anteriormente, para el desarrollo de nuevas aplicaciones se ha venido usando lenguajes orientados a objetos, se llegó a una solución rentable para que estos lenguajes se entendieran mejor con las bases de datos relacionales: el **ORM** (Object-Relational mapping) Pero el ORM por su relativa juventud no cuenta

con muchos editores y ejecutores de código que faciliten al programador probar sus consultas sin tener que hacerlo en la ejecución de la aplicación.

Basado en lo anterior nace la necesidad de crear una herramienta que facilite a los programadores probar todo tipo de consultas JPQL, en la cual se cargaran solo las clases correspondientes a las consultas dejando a un lado la lógica de negocio.

Actualmente lo más aproximado al software que se va a desarrollar en este trabajo de grado es un plugin para el IDE de desarrollo de Java "Eclipse", el cual permite, entre otras cosas, la ejecución de consultas sobre una base de datos relacional con clases mapeadas mediante un documento XML (utilizado en antiguas versiones de Hibernate antes de la estandarización con la especificación JPA). Nuestro proyecto no requiere de un documento XML adicional sino de clases con anotaciones siendo estas el único insumo para poder operar sobre la base de datos al momento de realizar las pruebas que el desarrollador planee ejecutar.

1.2.2 Impacto

Esta herramienta hará mucho más ágil y eficiente el trabajo de los desarrolladores que utilizan java como lenguaje de programación e Hibernate como implementación de JPA (Java Persistence API), pues les permitirá probar todas sus consultas con mucha comodidad, además de todas las facilidades que brinda gracias a sus características: Formateo de consultas, Carga dinámica de clases empaquetadas en jars, autocompletar código JPQL, creación de objetos java en tiempo de ejecución, etc.

1.2.3 Viabilidad

La viabilidad del proyecto es total, puesto que se cuenta con los recursos humanos y técnicos necesarios para garantizar el éxito de su desarrollo. El sistema se desarrolla bajo la supervisión de los directores del proyecto, los cuales cuentan con una amplia experiencia en diseño y desarrollo de sistemas. Además las herramientas de desarrollo utilizadas son de distribución libre.

2 MARCO TEÓRICO

2.1 Persistencia de objetos

2.1.1 ¿Qué es persistencia de objetos?

Es la capacidad que tienen los objetos de sobrevivir al proceso que los creó; permitiendo al programador almacenar, transferir, y recuperar el estado de los objetos.[5]

También se puede definir como: La acción de preservar la información de un objeto de forma permanente (guardar), pero a su vez también se refiere a poder recuperar la información del mismo (leer) para que pueda ser nuevamente utilizada. En el caso de persistencia de objetos la información que persiste en la mayoría de los casos son los valores que contienen los atributos en ese momento, no necesariamente la funcionalidad que proveen sus métodos.[6]

2.1.2 Métodos de persistencia de objetos

- Serialización.
- Bases de Datos Orientadas a Objetos (ODBMS).
- Bases de Datos Relacionales.

2.1.2.1 Serialización[7]

Es el proceso de convertir un objeto en una secuencia de bytes para conservarlo en memoria, una base de datos o un archivo. Su propósito principal es guardar el estado de un objeto para poder crearlo de nuevo cuando se necesita. El proceso inverso se denomina deserialización.

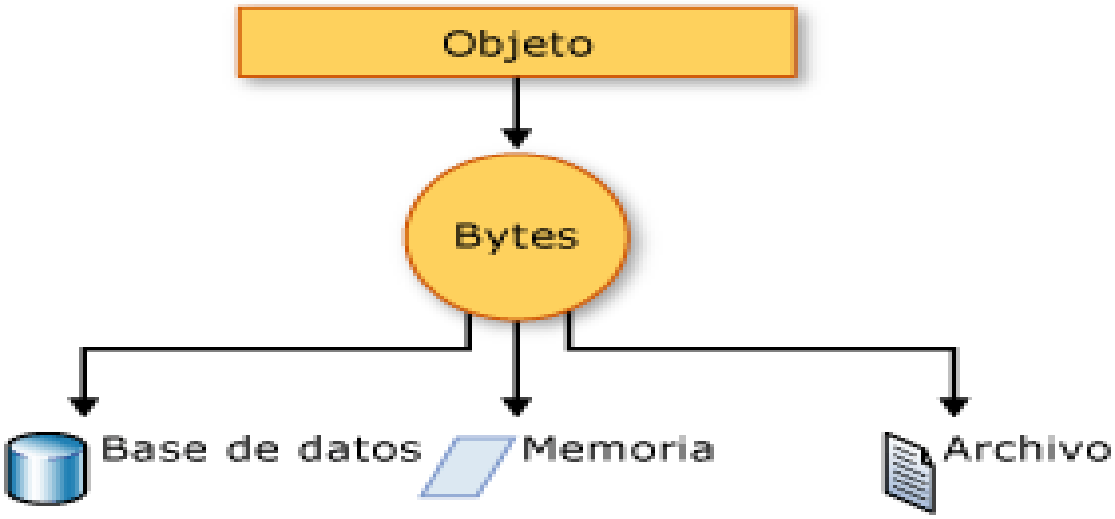
---

[5] WIKIPEDIA "Persistencia de objetos". {En línea}. {10 agosto de 2013} disponible en: (http://es.wikipedia.org/wiki/Persistencia_de_objetos.)

[6] WIKIBOOKS "Java persistence". {En linea}. *{10 agosto de 2013} disponible en:* (http://en.wikibooks.org/wiki/Java_Persistence.)

[7] MICROSOFT DEVELOPER NETWORK "Serialización". {En linea}. {10 agosto de 2013} disponible en: (http://msdn.microsoft.com/es -es/library/ms233843.aspx)

Ilustración 1: funcionamiento Serialización



Fuente*: http://msdn.microsoft.com/es-es/library/ms233843.aspx*

El objeto se serializa en una secuencia que, además de los datos, contiene información sobre el tipo de objeto, como la versión, referencia cultural y nombre de ensamblado. Esa secuencia se puede almacenar en una base de datos, un archivo o en memoria.

La serialización permite al desarrollador guardar el estado de un objeto y volver a crearlo cuando es necesario, y proporcionar almacenamiento de objetos e intercambio de datos. A través de la serialización, un desarrollador puede realizar acciones como enviar un objeto a una aplicación remota por medio de un servicio Web, pasar un objeto de un dominio a otro, pasar un objeto a través de un firewall como una cadena XML[8] o mantener la seguridad o información específica del usuario entre aplicaciones.

Sin embargo no soporta transacciones, consultas o acceso compartido a los datos entre usuarios múltiples y se la utiliza sólo para proporcionar persistencia en aplicaciones simples o en entornos empotrados que no pueden gestionar una base de datos de forma eficiente. Es evidente que, dada la tecnología

[8] XML: lenguaje de marcado extensible.

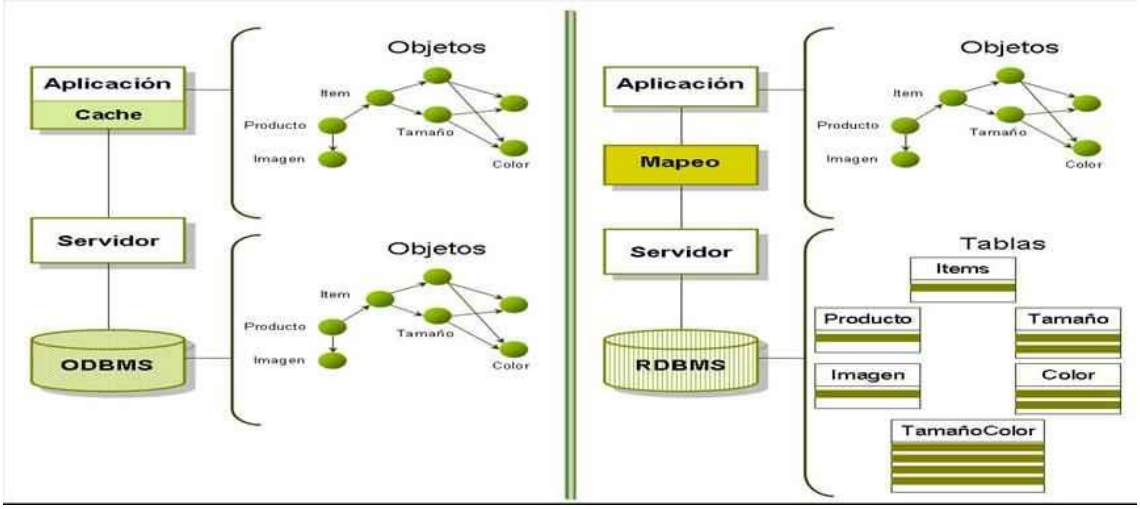
actual, la serialización como método de persistencia es insuficiente para la alta concurrencia web y aplicaciones empresariales.

2.1.2.2 Bases de datos orientadas a objetos

Un ODBMS, o base de datos orientada a objetos, proporciona un método transparente para la persistencia. Permite consultar y trabajar con objetos directamente. [9]

La persistencia transparente se refiere a la habilidad de manipular directamente, sin traducción o conversión, datos almacenados en la base utilizando un entorno de programación basado en objetos como Smalltalk, Java o C#. Esto contrasta con los RDBMSs donde se utiliza un sublenguaje como SQL o una interfaz de operación como ODBC o JDBC, luego, se utiliza código adicional para hacer la conversión a objetos.

Ilustración 2: almacenamiento de objetos vs mapeo de objetos



Fuente: *http://www.db4o.com/espanol/db4o%20Whitepaper%20-%20Bases%20de%20Objetos.pdf*

[9] Bauer, Christian., & Gavin, King. Java Persistente with Hibernate. Manning Publications. (2007).85p

La mayoría de los ODBMSs implementan un esquema de persistencia por capacidad de alcance. Ello significa que cualquier objeto referenciado por un objeto persistente también es persistido. Normalmente el programador puede especificar la profundidad de operación de este esquema en un árbol de objetos. De esta manera, conjuntos completos de objetos pueden ser almacenados y recuperados con una sola llamada; el ODBMS maneja automáticamente los detalles de mantenimiento de las referencias cuando se guardan y recuperan objetos.

Las bases de datos orientadas a objetos son quizás la forma más sencilla de persistir un modelo de objetos, aunque el mercado de las tecnologías de bases de datos orientadas a objetos es aún pequeño e inestable comparado con el mercado de las bases de datos relacionales.[10]

2.1.2.3 Bases de datos relacionales

Una descripción muy sencilla y directa del modelo relacional, es aquella en donde se define que la responsabilidad de las bases de datos relacionales es modelar la información basándose en relaciones definidas en conjuntos finitos de valores llamados dominios.

Una relación es un conjunto de listas ordenadas de valores llamadas tuplas. Cada tupla es un elemento del producto cartesiano de dominios. Cada ocurrencia de un dominio en la definición de una relación se denomina atributo, y el mismo dominio puede llegar a repetirse dentro de ella. Los dominios y las relaciones son implementados como tablas con m filas y n columnas. Cada fila corresponde a una tupla y cada columna a un atributo relacional.

El acceso a los datos se realiza a través de un conjunto básico de operaciones: selección, proyección, producto, join, unión, intersección y diferencia. Estas operaciones se expresan a través de un lenguaje denominado SQL (Lenguaje de Consultas Estructurado) usado para guardar, recibir y modificar información en la base de datos.

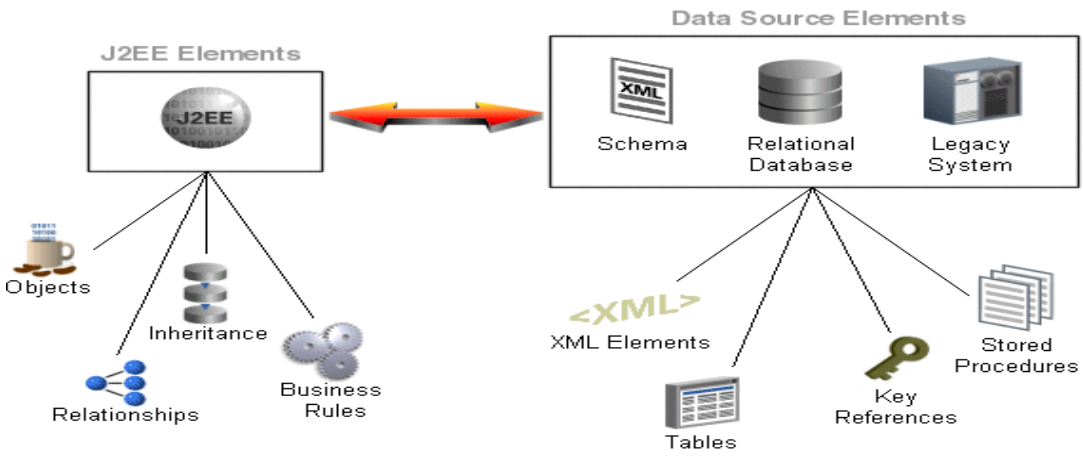
[10] VISCUOSO, German. "Bases de objetos ". {En línea}.{10 septiembre de 2013} disponible en: (http://www.db4o.com/espanol/db4o%20Whitepaper%20-%20Bases%20de%20Objetos.pdf)

Para recibir información se debe realizar una selección (caracterizada por la consulta SELECT en SQL). La selección de las filas permite especificar ciertas condiciones para los atributos acompañada generalmente con una proyección que significa que sólo un subconjunto de las columnas son seleccionadas. Para consultas más complejas se utiliza el operador JOIN que crea una tabla temporal con conjuntos de columnas pertenecientes a dos o más tablas.
De las tres formas de persistencia, sólo las bases de datos relacionales han demostrado ser escalables, robustas y lo suficientemente estándares para las aplicaciones empresariales.
No obstante cuando se quiere persistir los objetos utilizando una de ellas, se puede observar que hay un problema de compatibilidad entre el paradigma de la Orientación a Objetos y el modelo relacional, la también llamada diferencia o impedancia objeto-relacional.

2.1.3 Impedancia Objeto- Relacional

"Impedancia Objeto-Relacional", se define como un conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito bajo el paradigma de la Orientación a Objetos. [11]
La siguiente figura muestra algunas diferencias.

Ilustración 3: impedancia objeto – relacional



Fuente: *http://docs.oracle.com/cd/B32110_01/web.1013/b28218/undtl.htm*

[11] Scott W. Ambler "Mapping objects to relational databases: O/R mapping in detail". {En línea}. {12 septiembre de 2013}. Disponible en: (URL http://agiledata.org/essays/mappingObjects.html.)

Un ejemplo claro de esta impedancia se observa en el hecho que en el mundo de la programación orientada a objetos, se tiene un claro sentido de la pertenencia, a cada objeto le pertenecen sus correspondientes atributos; por ejemplo para el objeto Agenda Telefónica podríamos especificar como atributos a una colección de objetos llamados "persona", en la que a cada persona le corresponde su correspondiente atributo "teléfono", al transformar esto hacia el mundo relacional se ocuparía más de una tabla para almacenar la información, este simple hecho, hace notar que las tablas del modelo relacional son inconscientes de cómo están relacionadas con otras tablas a un nivel fundamental, puesto que aun cuando posean *constraints* para definir sus relaciones, para reconstruir el objeto originalmente persistido se debe construir un query, y dicho query debe especificar explícitamente como se relacionan las tablas entre sí, con esto se demuestra además que el lenguaje SQL a pesar de los constraints se mantiene inconsciente de las relaciones que a nivel de objeto poseen las tablas entre ellas.
Así como lo anteriormente expuesto se pueden enumerar distintos problemas que surgen entre los dos modelos:

Reglas de Acceso: En el modelo relacional los atributos pueden ser accedidos y/o modificados a través de operadores relacionales predefinidos, mientras que en el modelo orientado a objetos, se permite que cada clase defina la forma en que serán alterados los atributos así como la interface que ocupará para ello.

Ataduras del Esquema: Los objetos del modelo de la POO, no deben seguir ningún esquema en cuanto a qué atributos deben o pueden tener, puesto que son definidos por el programador, mientras que las tablas deben seguir el esquema entidad-relación.

Identificador único: Las llaves primarias de una fila tienen generalmente una forma de poder representarse como texto visible, mientras que los objetos no requieren un identificador único externamente visible.

Estructura vs Comportamiento: La orientación a objetos se concentra primordialmente en asegurar que la estructura del programa sea razonable (entendible, extensible, reutilizable, segura, etc.), mientras que los sistemas relacionales ponen el énfasis en tipo de comportamiento que el sistema tendrá una vez en producción (eficiencia, adaptabilidad, rapidez, etc.). Los métodos de la programación orientada a objetos asumen que el principal usuario del código orientado a objetos y sus beneficios es el desarrollador de aplicaciones, mientras que el modelo relacional enfatiza que la forma en que los usuarios finales perciben el comportamiento del sistema es mucho más importante.

De todo esto surge la necesidad de utilizar algún mecanismo para integrar la información contenida en nuestros objetos con los datos almacenados en la base de datos relacional.
Esto típicamente se logra a través de una capa de traducción objeto – relacional.

2.1.4 ORM (mapeo objeto relacional)

2.1.4.1 ¿Qué es ORM y cómo funciona?[12]

El mapeo objeto-relacional ORM (Object-relationalmapping) es una técnica de programación para la conversión de datos de sistemas incompatibles (bases de datos relacionales) a lenguajes de programación orientada a objetos y viceversa. Esto permite la creación de "bases de datos virtuales de objetos" que pueden ser usadas y manipuladas fácilmente con el lenguaje de programación orientado a objetos.

De esta manera se permite el almacenamiento de los objetos creados con sus atributos y comportamiento en una base de datos relacional para ser reutilizados cuando se necesiten.

[12] Bauer, Christian., & Gavin, King. Java Persistente with Hibernate. Manning Publications. (2007).105p

2.1.4.2 Componentes

Una solución ORM está formada por al menos cuatro partes:

- Una API que posibilite la realización de operaciones de creación, actualización y borrado sobre objetos de clases persistentes.

- Un lenguaje o API para poder especificar consultas sobre dichas clases.

- Una opción para especificar mapeo de metadatos (archivo XML, anotaciones directamente en las clases que representan las tablas de la base de datos).

- Una técnica para que la implementación del ORM pueda llevar a cabo búsquedas, asociaciones u otras funciones de optimización.

2.1.4.3 ¿Por qué elegir ORM?

- **Productividad:** El código relativo a la persistencia es una de las partes más tediosas y difíciles en el desarrollo de una aplicación. ORM elimina gran parte de ese trabajo tedioso y permite centrarse en la propia lógica de negocio, independientemente de la estrategia que se adopte. ORM convenientemente usado reduce de forma importante el tiempo de desarrollo.

- **Mantenibilidad:** Al tener menos de líneas de código, nuestra aplicación es más fácil de entender y mantener dado que se da énfasis a la lógica de negocio, tal y como se mencionó anteriormente. En las aplicaciones donde la persistencia se implementa "a mano" mediante un driver JDBC y código SQL, los dos mundos (el relacional y el de los objetos) no se llevan muy bien, y si se produce algún cambio en el diseño de uno de ellos, casi con toda seguridad esto implicará algún cambio en el otro. Además, los diseños de ambos están pensados para evitar conflictos entre ellos (generalmente, el diseño de las clases está hecho para evitar

conflictos con el diseño Entidad/Relación) cualquier herramienta ORM (Hibernate, TopLink, EclipseLink) establece, por así decirlo, un "buffer" entre ambos diseños para permitir una mayor flexibilidad.

- **Rendimiento:** Continuando con los beneficios, otro posible beneficio es el rendimiento. Esto puede resultar paradójico, ya que se suele argumentar que con la persistencia "ORM manual" se obtiene un rendimiento mayor que con herramientas ORM. A priori, esto es cierto (al igual que lo es que un programa escrito en ensamblador es más rápido que uno escrito en Java, C# o cualquier otro lenguaje de alto nivel), lo que ocurre es que hay que tener en cuenta las restricciones de tiempo que existen en el desarrollo de aplicaciones en el mundo real. Una herramienta ORM provee muchos beneficios que agilizan el desarrollo de las aplicaciones, hacen que el código sea más fácil de leer, evita en gran manera la repetición del código y hacen que el acceso a los datos sea más abstracto y portable. Además es lógico pensar que quienes han desarrollado una herramienta ORM han dedicado más tiempo que cualquiera de nosotros a la búsqueda de optimizaciones.

2.1.4.4 Consideraciones al implementar ORM

Este paso es muy importante ya que podemos quedar ligados a un producto poco eficiente, que desaparece del mercado y/o carece de mantenimiento.
La experiencia del desarrollador en el manejo de un ORM es determinante en la decisión de implementar una solución ya que de nada sirve poseer el mejor ORM si no se lo sabe usar.
Luego de obtener el ORM hay que definir como se trabajará, si se usará POJOs[13] o no, definir donde se encontrará el archivo de configuración si el *framework* ORM permite tener un archivo de configuración, creación de la base de datos, definición de las clases persistentes y no persistentes; definición del mapeo registro-objeto; definir los métodos de acceso a datos y de persistencia de los datos; programar los métodos y probarlos.

[13] POJOs: Simple clase Java.

2.1.4.5 Desventajas de ORM

- Tiempo utilizado en el aprendizaje. Este tipo de herramientas suelen ser complejas por lo que su correcta utilización lleva tiempo.

- Aplicaciones algo más lentas. Esto es debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

- Existen pocos editores y ejecutores de código, debido a que hasta ahora se está creando una especificación estándar para ORM en JAVA. Esto lleva a que la única manera de ejecutar una consulta sobre la base de datos mediante el JPQL sea con la aplicación final en tiempo de ejecución. Un ejemplo sería que para ejecutar consultas SQL en una base de datos en MySQL sólo se pudiera mediante la aplicación (por ejemplo en C++) que hace uso de los datos, y no a través de programas como PhpMyAdmin o MySQLWorkbench, que cuentan con un editor de código SQL y demás funciones que facilitan el acceso directo a los datos.
  No contar con una herramienta donde se pueda editar consultas y probarlas (tanto funcional como sintácticamente) causa ineficiencia en el desarrollo de aplicaciones que usan un ORM para el lenguaje JAVA.

2.1.4.6 Motores de persistencia más comunes

Se presenta un listado de los motores de persistencia más comunes y el lenguaje para el cual están hechos.[14]

- C#: **Entity Framework** es un conjunto de APIs que proporcionan acceso a datos en .NET. Se distribuye junto con el .NET framework y tiene 3 posibles modos de trabajo, *Database First, Model First* y *Code First.*

[14] Bisbe, Roberto "¿Qué es un ORM y por qué nos interesa?". {En linea}. {12 septiembre de 2013} disponible en: (http://rlbisbe.net/tag/orm/)

- Java: **Hibernate**[15] es una herramienta de mapeo relacional para la plataforma Java, que emplea atributos declarativos mediante XML o anotaciones, se distribuye con licencia GNU/LGPL.

- Objective-C: **Core Data** (Parte de la API de Cocoa), proporciona la capacidad de persistencia mediante serialización para dispositivos con Mac OSX o iOS.

- PHP: **Doctrine**, un proyecto independiente, con la especialidad de que posee su propio lenguaje para el acceso a datos, llamado Doctrine Query Language.

- Ruby: **ActiveRecord** (Parte de Ruby On Rails), genera un modelo de persistencia basado en las clases, proporciona acceso a datos en el framework y es la clase base para los modelos del mismo.

2.1.5 Tecnología usada en el proyecto

2.1.5.1 Arquitectura multicapa

El objetivo de la arquitectura es la separación de todo el sistema en capas diferentes en las que se dividen las funciones del sistema. La ventaja principal de este estilo es que el desarrollo se puede llevar a cabo en varios niveles y, en caso de que sobrevenga algún cambio, sólo se ataca al nivel requerido sin tener que revisar entre código mezclado.

En el diseño de sistemas informáticos actual se suelen usar las arquitecturas multinivel o Programación por capas, lo que permite el diseño de arquitecturas escalables (que pueden ampliarse con facilidad en caso de que las necesidades aumenten). El diseño más utilizado actualmente es el diseño en tres capas:

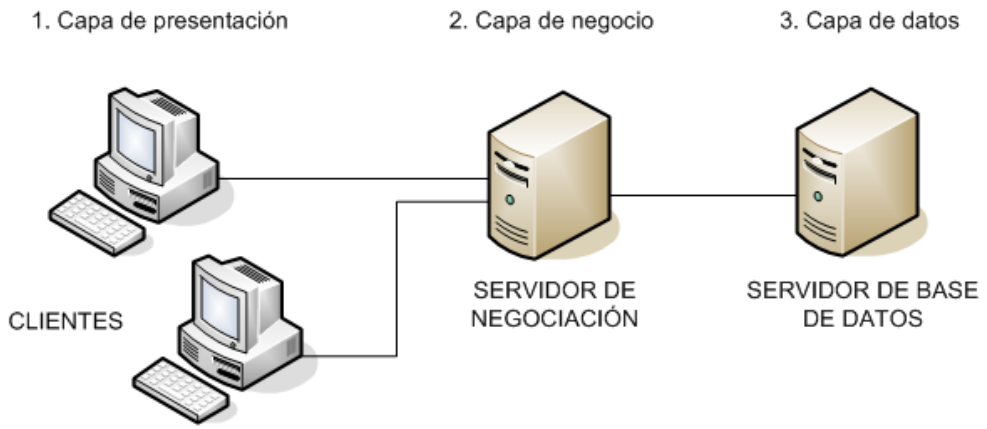
[15] HIBERNATE "Everything data". {En linea}. {12 septiembre de 2013} disponible en: (http://hibernate.org)

Capa de presentación: es la que se tiene contacto con el usuario. Presenta el sistema al usuario, le comunica la información y captura la información del usuario dando un mínimo de proceso (realiza un filtrado previo para comprobar que no hay errores de formato). Esta capa se comunica únicamente con la capa de negocio.

Capa de la lógica de negocio: es donde residen los programas que se ejecutan, se reciben las peticiones del usuario y se envían las respuestas tras el proceso. Se denomina capa de negocio pues es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al gestor de base de datos para almacenar o recuperar datos de él.

Capa de datos: es donde residen los datos y es la encargada de acceder a los mismos. Está formada por uno o más gestores de bases de datos que realizan todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio.

Ilustración 4: Arquitectura multicapa



Fuente: *http://es.wikipedia.org/wiki/Archivo:Tres_capas.PNG*

2.1.5.2 Entorno de desarrollo

Nuestra herramienta JPangolin se realizó en JavaSE 7[16] Utilizando las siguientes librerías y herramientas:

- **Swing**[17]

Es una biblioteca gráfica para Java, provee los elementos necesarios para crear cualquier tipo de interfaz gráfica de usuario. Se mostrara a continuación un listado de los elementos más comunes.

- *JFrame*: Representa una ventana básica, capaz de contener otros componentes. Casi todas las aplicaciones construyen al menos un *Jframe.*
- *JDialog, JOptionPane, etc*: Los cuadros de diálogo son *JFrame* restringidos, dependientes de un *JFrame* principal. Los *JOptionPane* son cuadros de diálogo sencillos predefinidos para pedir confirmación, realizar advertencias o notificar errores. Los *JDialog*.
- -JInternalFrame: Consiste simplemente en una ventana hija, que no puede salir de los límites marcados por la ventana principal.

- *JPanel:* Un panel sirve para agrupar y organizar otros componentes. Puede estar decorado mediante un borde y una etiqueta.
- *JScrollPane:* Es un panel que permite visualizar un componente de un tamaño mayor que el disponible, mediante el uso de barras de desplazamiento.
- *JSplitPane*: Permite visualizar dos componentes, uno a cada lado, con la posibilidad de modificar la cantidad de espacio otorgado a cada uno.
- *JTabbedPane:* Permite definir varias hojas con pestañas, que pueden contener otros componentes. El usuario puede seleccionar la hoja que desea ver mediante las pestañas.

[16] ORACLE "Platform, Standard Edition 7 API Specification". {En linea}. {20 septiembre de 2013} disponible en: (http://docs.oracle.com/javase/7/docs/api/).

[17] ORACLE "Using Swing components". {En linea}. {20 septiembre de 2013} disponible en: (http://docs.oracle.com/javase/tutorial/uiswing/components/).

- *JButton*, *JCheckBox* y *JRadioButton*: Distintos tipos de botones. Un check box sirve para marcar una opción. Un radio button permite seleccionar una opción entre varias disponibles
- *JToolBar*: Es un contenedor que permite agrupar otros componentes, normalmente botones con iconos en una fila o columna.
- *JComboBox*: Las combo boxes o listas desplegables que permiten seleccionar un opción entre varias posibles.
- *JList*: Listas que permiten seleccionar uno o más elementos.
- *JTextField*, *JFormattedTextField*, *JpasswordField:* Distintos tipos de editores Para que el usuario inserte un valor.
- *JSlider*: Un slider permiten introducir un valor numérico entre un máximo y un mínimo de manera rápida.
- *JPopupMenu*: Un menú que se obtiene al pulsar con el botón derecho del ratón sobre una zona determinada. Los menús están compuestos por distintos ítems:
- *JColorChooser*: Consiste en un selector de colores.
- *JFileChooser*: Permite abrir un cuadro de diálogo para pedir un nombre de fichero.
- *JTree*: Su función es mostrar información de tipo jerárquico.
- *JLabel:* Permite situar un texto, un texto con una imagen o una imagen únicamente en la ventana.
- *JProgressBar*: Permite mostrar que porcentaje del total de una tarea a realizar ha sido completado.

➢ Reflection[18]

El API "*Reflection*" es, para java, un API nativo que permite reflejar clases, objetos y métodos de un determinado programa para extraer de ahí la información pertinente. Esto es realmente útil cuando se escriben aplicaciones como navegadores de clases, depuradores, u otras herramientas de desarrollo en donde no se puede acceder a algunas características de las clases, objetos y otros componentes de diseño del programa que se esté examinando hasta tiempo de ejecución.

[18] GONZALES ENEKO, Benito Java hispano. ". {En linea}. {22 septiembre de 2013} disponible en: (http://www.javahispano.org/storage/contenidos/reflection.pdf)

- **JPA**[19]

La API de persistencia Java es el estándar de transformación objeto/relacional que permite a los desarrolladores Java manejar datos relacionales en las aplicaciones Java mediante anotaciones o con descriptores XML.
En este proyecto es realizado para trabajar con clases mapeadas con anotaciones por eso enfocamos nuestras explicaciones hacia ese tipo de mapeo.

La especificación del JPA cubre tres áreas importantes de la persistencia:

- API de persistencia Java.(*EntityManager*)
- Metadatos de transformación objeto/relacional (Anotaciones).
- Lenguaje de consulta.

- **Arquitectura JPA**.

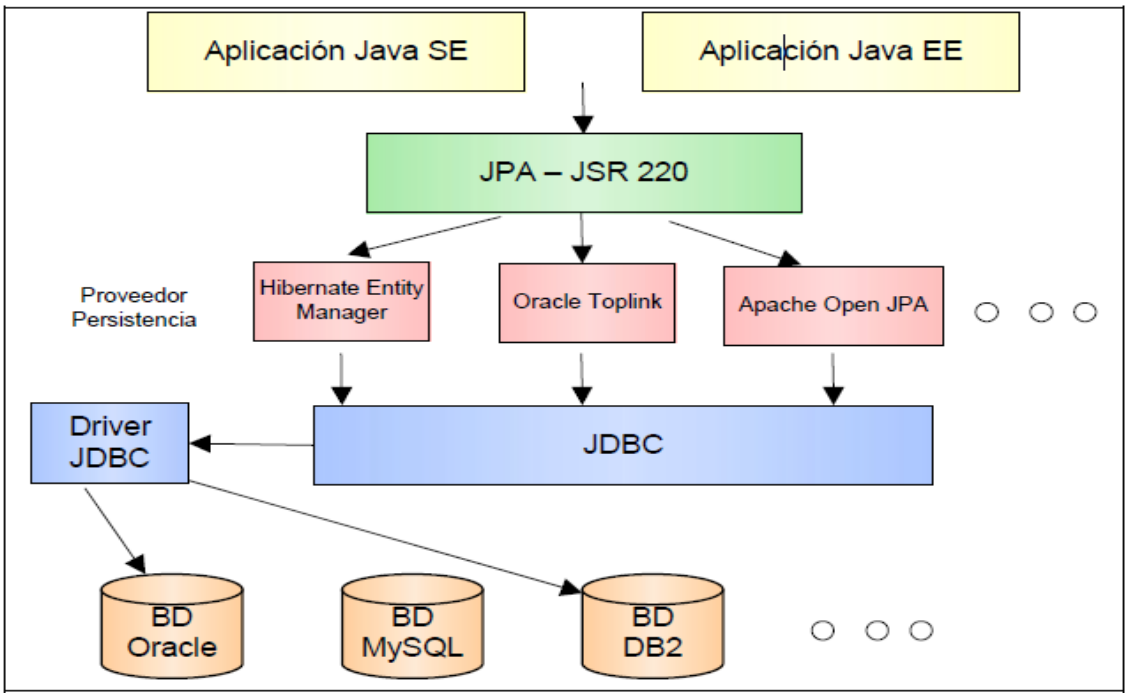
Para utilizar JPA, es necesario elegir un proveedor de persistencia el cual se ocupa de lo relacionado con la carga y almacenamiento de los datos, cuando refrescar cada instancia y de la sincronización entre los objetos.

- **Elementos**

- *POJOs*
- Anotaciones
- Entidades
- Administrador de Entidades

[19] ORACLE "The Java EE 6 tutorial". {En linea}. {28 septiembre de 2013} disponible en: (http://docs.oracle.com/javaee/7/tutorial/doc/)

Ilustración 5: Arquitectura externa JPA



- *Pojos (Plain Old Java Object)*

Son clases simples de java surgen como una reacción en el mundo Java a los *frameworks* cada vez más complejos que esconden el problema que realmente se está modelando.

La principal caracteristica de un objeto POJO es que es instancia de una clase que no extiende ni implementa nada en especial, simplemente define variables y sus métodos *getters y setters.*

Ilustración 6: ejemplo de POJO

```
public class Materia {

    private int codigoMateria;
    private int creditosMateria;
    private String nombreMateria;

    public int getcodigoMateria() {
        return codigoMateria;
    }

    public void setCodigoMateria(int codigoMateria) {
        this.codigoMateria = codigoMateria;
    }

    public String getnombreMateria() {
        return nombreMateria;
    }

    public void setnombreMateria(String nombreMateria) {
        this.nombreMateria = nombreMateria;
    }

    public int getcreditosMateria() {
        return creditosMateria;
    }

    public void setcreditosMateria(int creditosMateria) {
        this.creditosMateria = creditosMateria;
    }
}
```

- **Anotaciones**

Una anotación o metadato proporciona un recurso adicional al elemento de código al que va asociado en el momento de la compilación. Cuando la máquina virtual de Java (JVM) ejecuta la clase busca estos metadatos y

determina el comportamiento a seguir con el código al que va unido la anotación.

A continuación se describen las principales anotaciones:

@Entity: Indica que es una Entidad.

name: Por defecto el nombre de la clase pero se puede especificar otra diferente.

@Table: Específica la tabla principal relacionada con la entidad.

name: Nombre de la tabla, por defecto el de la entidad si no se especifica.

catalog: Nombre del catálogo.

schema: Nombre del esquema.

uniqueConstraints: Constrains entre tablas relacionadas con la anotación @Column y @JoinColumn.

@SecondaryTable: Especifica una tabla secundaria relacionada con la entidad si éste englobara a más de una. Tiene los mismos atributos que @Table.

@SecondaryTables: Indica otras tablas asociadas a la entidad.

@UniqueConstraints: Específica que una única restricción se incluya para la tabla principal y la secundaria.

@Column: Especifica una columna de la tabla a mapear con un campo de la entidad.

name: Nombre de la columna.
unique: Si el campo tiene un único valor.
nullable: Si permite nulos.
insertable: Si la columna se incluirá; en la sentencia INSERT generada.
updatable: Si la columna se incluirá; en la sentencia UPDATE generada.
table: Nombre de la tabla que contiene la columna.
length: Longitud de la columna.
precision: Número de dígitos decimales.
scale: Escala decimal.

@JoinColumn: Especifica una campo de la tabla que es foreign key de otra tabla definiendo la relación del lado propietario.

name: Nombre de la columna de la foreign key.
referenced: Nombre de la columna referencia.
unique: Si el campo tiene un único valor.
nullable: Si permite nulos.
insertable: Si la columna se incluirá; en la sentencia INSERT generada.
updatable: Si la columna se incluirá; en la sentencia UPDATE generada.
table: Nombre de la tabla que contiene la columna.

@JoinColumns: Anotación para agrupar varias JoinColumn.

@Id: Indica la clave primaria de la tabla.

@GeneratedValue: Asociado con la clave primaria, indica que ésta se debe generar por ejemplo con una secuencia de la base de datos.

- *strategy* – estrategia a seguir para la generación de la clave: AUTO (valor por defecto, el contenedor decide la estrategia en función de la base de datos.
- *generator* – forma en la que general la clave.

@SequenceGenerator: Define un generador de claves primarias utilizado junto con la anotación @GeneratedValue. Se debe especificar la secuencia en la entidad junto a la clave primaria.

- *name* – nombre del generador de la clave.
- *sequence* – nombre de la secuencia de la base de datos del que se va a obtener la clave.
- *initialValue* – valor inicial de la secuencia.
- *allocationSize* – cantidad a incrementar de la secuencia cuando se llegue al máximo.

@TableGenerator: Define una tabla de claves primarias generadas. Se debe especificar en la anotación @GeneratedValue con strategy = GenerationType.TABLE.

- *name* – nombre de la secuencia.
- *table* – nombre de la tabla que guarda los valores generados.
- *catalog* – catalog de la tabla.
- *schema* – esquema de la tabla.
- *pkColumn Name()* - nombre de la clave primaria de la tabla.
- *valueColumn Name()* - nombre de la columna que guarda el último valor generado.
- *pkColumn Value()* - valor de la clave primaria.
- *initialValue* – valor inicial de la secuencia.
- *allocationSize* – cantidad a incrementar de la secuencia.
- *uniqueConstraints* - constrains entre tablas relacionadas.

@AttributeOverride: Indica que sobrescriba el campo con el de la base de datos asociado.

- *name* – nombre del campo
- *column* – columna del campo

@AttributeOverrides: Mapeo de varios campos.

@EmbeddedId: Se utiliza para formar la clave primaria con múltiples campos.

@IdClass: Se aplica en la clase entidad para especificar una composición de la clave primaria mapeada a varios campos o propiedades de la entidad.

@Transient: Indica que el campo no se debe persistir.

@Version: Se utiliza a la hora de persistir la entidad en base de datos para identificar las entidades según su versión. Se actualiza automáticamente cuando el objeto es mapeado en la base de datos.

@Basic: Mapeo por defecto para tipos básicos: tipos primitivos, wrappers de los tipos primitivos, String, BigInteger, BigDecimal, Date, Calendar, Time, Timestamp, byte[], Byte[], char[], Character[], enumerados y cualquier otra clase serializable.

- *fetch* – determina la forma en que se cargan los datos (relaciones entre tablas): FetchType.LAZY (carga de la entidad únicamente cuando se

utiliza), FetchType.EAGER (carga de todas las entidades relacionadas con ella).

- *optional* - si permite que el campo sea nulo. Por defecto true.

@OneToOne: (1:1) Indica que un campo está; en relación con otro (ej: dentro de una empresa, un empleado tiene un contrato de trabajo).

- *cascade* – forma en que se deben actualizar los campos: ALL, PERSIST, MERGE, REMOVE y REFRESH.
- *fetch* – determina la forma en que se cargan los datos: FetchType.LAZY (carga de la entidad únicamente cuando se utiliza), FetchType.EAGER (carga de todas las entidades relacionadas con ella).
- *optional* – si la asociación es opcional.
- *mappedBy* – el campo que posee la relación, únicamente se especifica en un lado de la relación.

@ManyToOne: (N:1) Indica que un campo está; asociado con varios campos de otra entidad (ej: las cuentas que posee un banco o los empleados que pertenecen a un departamento).

- *cascade, fetch* y *optional* – igual que la anterior anotación.

@OneToMany: (1:N) Asocia varios campos con uno (ej: varios departamentos de una empresa).

- *cascade, fetch* y *optional* – igual que la anterior anotación.
- *mappedBy* – El campo que posee la relación. Es obligatorio que la relación sea unidireccional.

@ManyToMany: (N:M) Asociación de varios campos con otros con multiplicidad muchos-a-muchos (ej: relaciones entre empresas).

- *cascade, fetch* y *mappedBy* – igual que la anterior anotación.

@Lob: Se utiliza junto con la anotación @Basic para indicar que un campo se debe persistir como un campo de texto largo si la base de datos soporta este tipo.

@Temporal: Se utiliza junto con la anotación @Basic para especificar que un campo fecha debe guardarse con el tipo java.util.Date o java.util.Calendar. Si no se especificara ninguno de estos por defecto se utiliza java.util.Timestamp.

@Enumerated: Se utiliza junto con la anotación @Basic e indica que el campo es un tipo enumerado (STRING), por defecto ORDINAL.

@JoinTable: Se utiliza en el mapeo de una relación ManyToMany o en una relación unidireccional OneToMany.

- *name* – nombre de la tabla join a donde enviar la foreing key.
- *catalog* – catalog de la tabla.
- *schema* – esquema de la tabla.
- *joinColumns* – columna de la foreign key de la tabla join que referencia a la tabla primaria de la entidad que posee la asociación (sólo en un lado de la relación).
- *inverseJoinColumns* – columnas de la foreign key de la tabla join que referencia a la tabla primaria de la entidad que no posee (lado inverso de la relación)
- *uniqueConstraints* – constraints de la tabla.

@MapKey: Especifica la clave de una clase de tipo java.util.Map

@OrderBy: Indica el orden de los elementos de una colección por un item específico de forma ascendente o descendente.

@Inheritance: Define la forma de herencia de una jerarquía de clases entidad, es decir la relación entre las tablas relacionales con los Beans de entidad.

- *strategy* – SINGLE_TABLE que indica una única tabla para varias entidades (por defecto), JOINED, TABLED_PER_CLASS

@DiscriminatorColumn: Relacionada con @Inheritance cuando ésta define la estrategia SINGLE_TABLE. Define la columna común de las entidades que forman la jerarquía..

- *name* – nombre de la columna, por defecto DTYPE

- *DiscriminatorType* – Tipo de la columna que se utiliza para identificar la entidad. Por defecto DiscriminatorType.STRING.
- *length* – longitud de la columna, por defecto 31.

@PrimaryKeyJoinColumn: Especifica la clave primaria de la columna que es clave extranjera de otra entidad.

- *name* – nombre de la clave primaria
- *referenced* – nombre de la columna

@PrimaryKeyJoinColumns: Agrupación de varias claves iguales que la anterior.

@Embeddable: Con esta anotación podemos especificar en la propia entidad un objeto embebido de una entidad diferente.

@Embedded: Especifica un campo persistente de una entidad especificado como campo embebido de otra entidad mediante la anotación@Embeddable.

@MappedSuperclass: Asigna la información mapeada a la superclase de la entidad.

@NamedQuery: Especifica el nombre del objeto query utilizado junto a EntityManager.

- name: Nombre del objeto query.
- query: Especifica la query a la base de datos mediante lenguaje Java Persistence Query Language (JPQL).

@NamedQueries: Específica varias queries como la anterior.

@NamedNativeQuery: Específica el nombre de una query SQL normal.

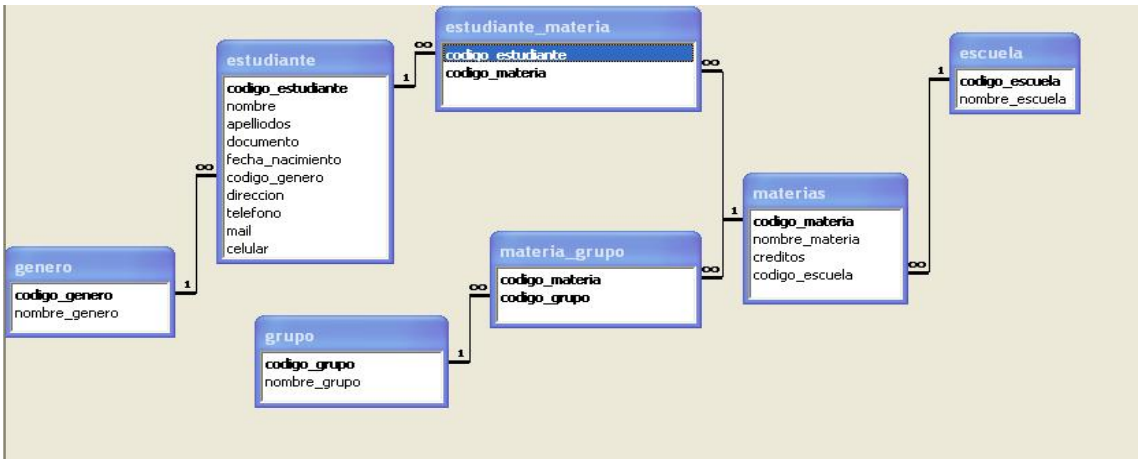
- name: Nombre del objeto query.
- query: Especifica la query a la base de datos.
- resultClass: Clase del objeto resultado de la ejecución de la query.
- resultSetMapping: Nombre del SQLResultSetMapping definido.

@NamedNaviteQueries: Específica varias queries SQL.

- **Ejemplo de mapeo**

A continuación se muestra un ejemplo de cómo mapear una tabla de la base de datos con las anotaciones más comunes para esto utilizaremos una tabla de la base de datos mostrada en la siguiente figura.

Ilustración 7: Modelo de base de datos para ejemplo



La tabla que será mapeada en este ejemplo es la tabla estudiante y se explicara por pasos la creación de una clase Java mediante la cual se creara la entidad Estudiante.

```
/* Importamos las librerías necesarias para el funcionamiento de
la clase */
```

Ilustración 8: ejemplo mapeo paso 1

```
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.ManyToOne;
import javax.persistence.Temporal;
```

/* Se Indica que es una entidad, se declara el nombre de la tabla y definen las variables */

Ilustración 9: ejemplo mapeo paso 2

```
/* Indica que es una Entidad */
@Entity
/* Tabla a mapear, en este caso se mapeará a la tabla 'estudiante' */
@Table(name = "estudiante")
/* Cada de Entidad debe implementar la interfaz Serializable */
public class Estudiante implements Serializable {

    /*Se definen la variables de la clase  como se puede observar ellas
      corresponden a las columnas de la tabla,se debe definir el tipo
      de cada una y en el caso de la variable  codigoGenero es de tipo
      Genero pues es una clave foránea que nos enlaza con la tabla genero*/

    private int codigoEstudiante;
    private String nombresEstudiante;
    private String apellidosEstudiante;
    private int documentoEstudiante;
    private Date fechaNacimiento;
    private Genero codigoGenero;
    private String direccionEstudiante;
    private String telefonoEstudiante;
    private String emailEstudiante;
    private Set<Materia> materias = new HashSet<Materia>();

    /* En el constructor se crea un objeto de tipo Genero y se asigna
     a la variable codigoGenero  que representa la llave foránea hacia la tabla genero */
    public Estudiante() {

        this.codigoGenero = new Genero();
    }
```

/* Se Indica cual es la llave primaria por medio de la anotación @Id, se genera un valor para la llave primaria, se definen las columnas relacionadas a los campos y sus métodos *getters* y *setters*. En este ejemplo no se están mostrando todos los campos por razones de simplicidad */

Ilustración 10: ejemplo mapeo paso 3

```
/* Clave primaria de la tabla 'estudiante' */
@Id
/* Indica la generación de la clave */
@GeneratedValue
/* Se definen las columnas y sus metodos getters y setters*/
@Column(name = "codigo_estudiante", nullable = false)

public int getCodigoEstudiante() {
    return codigoEstudiante;
}

public void setCodigoEstudiante(int codigoEstudiante) {
    this.codigoEstudiante = codigoEstudiante;
}

@Column(name = "nombre", nullable = false)
public String getNombresEstudiante() {
    return nombresEstudiante;
}

public void setNombresEstudiante(String nombresEstudiante) {
    this.nombresEstudiante = nombresEstudiante;
}

@Column(name = "apellidos", nullable = false)
public String getApellidosEstudiante() {
    return apellidosEstudiante;
}
```

```
/*  La columna codigo_genero se define un tanto diferente pues se debe especificar que es una llave foránea con relación de muchos a uno o*/
```

Ilustración 11: ejemplo mapeo paso 4

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "codigo_genero", referencedColumnName = "codigo_genero")
public Genero getCodigoGenero() {

    return codigoGenero;
}

public void setCodigoGenero(Genero codigoGenero) {

    this.codigoGenero = codigoGenero;
}
```

- **Entidades**

Se puede definir como entidad a cualquier objeto, real o abstracto, que existe en un contexto determinado o puede llegar a existir y del cual deseamos guardar información, por ejemplo: "PROFESOR", "CURSO", "ALUMNO".

Entidad = POJO + anotaciones.

- ✓ Típicamente representa una tabla de una base de datos relacional.
- ✓ Cada instancia de una entidad corresponde normalmente a una fila de la tabla.

- **Requerimientos para que una clase sea entidad**

✓ La clase debe llevar la anotación @Entity.

✓ Tener un constructor por omisión público o protegido. Sin argumentos.

✓ La clase, los métodos o las variables de instancia persistentes no deben ser declarados final.

✓ Si una instancia de un *entity* es pasada por valor como un objeto debe implementar la interfaz *java.io.Serializable.*

✓ Las variables persistentes deben ser declaradas *private* o *protected* y solamente pueden ser accedidas a través de los correspondientes métodos de acceso: *getters* o *setters.*

✓ Un Entity debe declarar una llave primaria. Esto se hace marcando el campo con la anotación @Id. Al igual que en las bases de datos relacionales, la llave primaria hace del *entity* un objeto único. Cuando se requiere de una llave compuesta se puede usar una clase como llave primaria compuesta. Para este tipo de claves, se utilizan las anotaciones *javax.persistence.EmbededId y javax.persistence.IdClass*.

- **Administrador de entidades (EntityManager)**

Una instancia del EntityManager está asociada con un contexto de persistencia el cual es un conjunto de instancias de entidades en el cual, para cada entidad persistente hay una instancia de entidad única. El EntityManager Proporciona métodos para gestionar la persistencia de una entidad. Permite añadir, eliminar, actualizar y consultar así como manejar su ciclo de vida. Sus métodos más importantes son:

- **persist(Object entity):** Almacena el objeto Entity en la base de datos.

- **merge(T entity):** Actualiza las modificaciones en la entidad devolviendo la lista resultante.

- **remove(Object entity):** Elimina la entidad.

- **find(Class<T> entity, Object primaryKey):** busca la entidad a través de su clave primaria.

- **flush():** Sincroniza las entidades con el contenido de la base de datos.

- **refresh(Object entity):** Refresca el estado de la entidad con su contenido en la base de datos.

- **createQuery(String query**): Crea una query utilizando el lenguaje JPQL.

- **createNativeQuery():** Crea una query utilizando el lenguaje SQL.

- **isOpen():** Comprueba si está; abierto el EntityManager.

- **close():** Cierra el EntityManager.

Existen varias maneras de obtener una referencia al EntityManager, la seleccionada para este proyecto es mediante una instancia del objeto EJB3Configuration al cual se le agregan todas las entidades que formarán parte del contexto de persistencia junto con toda la configuración adicional como los datos de conexión, el dialecto que usará Hibernate para comunicarse con la base de datos y otras propiedades del mapeo que normalmente se definirían mediante un archivo XML. El objeto se usa para crear una instancia del EntityManager de la siguiente forma:

- Se crea una instancia del objeto *EJB3Configuration*

```
EJB3Configuration objetoEjb3 = new EJB3Configuration();
```

- Se crea un objeto tipo Properties para agregar todas las configuraciones de hibérnate.

```
Properties propiedades new Properties();
```

- Una vez se agregan todas las configuraciones al objeto propiedades, estas se pasan al objeto EJB3Configuration mediante el método addProperties().

```
objetoEJB3.addProperties(propiedades);
```

- Luego se agregan una a una todas las entidades usando el método addAnnotatedClass().

```
objetoEjb3.addAnnotatedClass(entidad);
```

- Por último se crea la instancia del EntityManager:

```
objetoEjb3.getEjb3Configuration().buildEntityManagerFactory().createEntityManager();
```

- **Lenguaje de consulta JPQL**[20]

El lenguaje de consulta de persistencia de Java JPQL *(Java Persistence Query Language*) se usa para ejecutar consultas sobre objetos persistentes independientes del mecanismo usado para el almacenamiento de los mismos. Como tal, JPQL es "portable" y no está restringido a algún motor de base de datos específico. Este lenguaje permite ejecutar consultas tipo SELECT, DELETE y UPDATE, operaciones de JOIN, agregaciones, proyecciones, y subconsultas. Las consultas pueden ser declaradas estáticamente en metadatos o pueden ser construidas dinámicamente en el código.

[20]ORACLE "JPQL Language Reference". {En linea}. {05 octubre de 2013} disponible en: (http://docs.oracle.com/cd/E11035_01/kodo41/full/html/ejb3_langref.html).

Tabla 1: Ejemplo de una consulta en JPQL y su equivalente en SQL.

Sentencia SELECT en JPQL	Sentencia equivalente en SQL
SELECT c FROM Clientes c WHERE c.cliente_id=:567897	SELECT * FROM Clientes WHERE cust_id = 567897.

Como se puede observar en la consulta JPQL los elementos que son seleccionados son los objetos como tal que cumplan las condiciones especificadas después de la sentencia FROM en lugar de seleccionar tuplas de una tabla.

Nota: En la sentencia *SELECT* en JPQL de la Tabla 1 se tiene la expresión Clientes c entre las cláusulas *FROM y WHERE*. Esto corresponde a la declaración de c como variable rango de tipo Clientes.

Para crear consultas utilizando el lenguaje JPQL la interfaz EntityManager ofrece dos métodos: `EntityManager.createQuery` y `EntityManager.createNamedQuery.`

El método `createQuery` es usado para crear consultas dinámicas las cuales son definidas directamente en la lógica de negocio de las aplicaciones. Ejemplo:

```
Public List buscarClientePorNombre(String nombre){

return em.createQuery(“ SELECT  c FROM Cliente c WHERE c.nombre LIKE :nombreCliente “)//Se establece la consulta

.setParameter(“nombreCliente”, nombre) //Se le da el valor al parámetro “nombreCliente” declarado en la consulta como :nombreCliente

.getResultList(); //Se ejecuta la consulta y se retorna la lista de objetos tipo Cliente

}
```

Las consultas creadas mediante el método createNamedQuery son consultas estáticas que son definidas como anotaciones en las entidades mediante la anotación @NamedQuery

```
@NamedQuery(
    name="buscarTodosLosClientesPorNombre",
    query="SELECT c FROM Cliente c WHERE c.nombre LIKE
:nombreCliente"
)
@Entity
Cliente
...
```

Al momento de ejecutar la consulta se llama de la siguiente forma:

```
public EntityManager em;
...
clientes = em.createNamedQuery("buscarTodosLosClientesPorNombre
")
    .setParameter("nombreCliente", "Alberto")
    .getResultList();
...
```

El lenguaje JPQL soporta parámetros por nombre (como el parámetro nombreCliente en el ejemplo anterior) o por posición como se muestra a continuación:

```
Public List buscarClientePorNombre(String nombre){

return em.createQuery(“ SELECT  c FROM Cliente c WHERE c.nombre
LIKE ?1 “)//Se establece la consulta

.setParameter(1, nombre) //Se le da el valor al parámetro
“nombreCliente” declarado en la consulta como :nombreCliente

.getResultList(); //Se ejecuta la consulta y se retorna la lista
de objetos tipo Cliente

}
```

Consultas tipo SELECT:

Una consulta de tipo SELECT en la sintaxis JPQL se representa con la notación BNF de la siguiente forma:

```
QL_statement ::= select_clause from_clause

[where_clause][groupby_clause][having_clause][orderby_clause]
```

Una consulta SELECT tiene 6 cláusulas: SELECT, FROM, WHERE, GROUP BY, HAVING y ORDER BY. Las cláusulas SELECT y FROM son obligatorias, pero las demás son opcionales.

La cláusula SELECT define el tipo de objetos o valores que retorna la consulta.

La cláusula FROM define el ámbito de la consulta al declarar una o más variables rango, las cuales pueden ser referenciadas en las cláusulas SELECT y WHERE.

Una variable de identificación puede representar a una entidad, un miembro de una colección producto de relaciones uno a muchos y muchos a muchos, o una entidad externa relacionada como uno a uno.

La cláusula WHERE representa una expresión condicional que restringe los objetos o valores que retorna la consulta.

La cláusula GROUP BY agrupa los resultados de la consulta de acuerdo a un conjunto de propiedades.

La cláusula HAVING es usada junto con la cláusula GROUP BY para restringir los resultados de la consulta mediante expresiones condicionales.

La cláusula ORDER BY ordena los objetos o valores retornados por la consulta en un orden específico.

Consultas tipo UPDATE y DELETE:

En notación BNF se representan de la siguiente forma:

```
update_statement :: = update_clause [where_clause]
delete_statement :: = delete_clause [where_clause]
```

El lenguaje JPQL permite operaciones masivas de borrado y actualización mediante estos dos tipos de consulta. El uso opcional de la cláusula SELECT puede contribuir a restringir el alcance de las consultas de tipo UPDATE y DELETE.

Para nuestro este fue fundamental el estudio detallado de la sintaxis del lenguaje JPQL para implementar la función de autocompletado del editor JPQL.

- **Hibernate.**

Es una herramienta de Mapeo objeto-relacional (ORM) 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 de las entidades que permiten establecer estas relaciones.

Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.

Al ser Hibernate una implementación del JPA ofrece las herramientas antes mencionadas junto con más características disponibles para que el programador haga uso del ORM en sus proyectos. El Lenguaje de consultas de Hibernate (HQL) es la implementación del JPQL. Al ser HQL una implementación del JPQL, toda consulta realizada en JPQL funciona de igual manera en HQL pero el recíproco no siempre es válido ya que el lenguaje HQL ofrece otras funcionalidades que no se encuentran en el lenguaje JPQL (ejemplo: HQL soporta sentencias INSERT de una forma similar al lenguaje SQL).

Hibernate funciona asociando a cada tabla de la base de datos *un Plain Old Java Object (POJO)* el cual viene dado por medio de una clase con métodos getter y setter para acceder y modificar la información contenida en el mismo.

Para poder asociar el POJO a su tabla correspondiente en la base de datos, Hibernate usa ficheros hbm.xml para poder mapearlo con la base de datos. En el fichero XML se declaran las propiedades del POJO y sus correspondientes nombres de columna en la base de datos, asociación de tipos de datos,

referencias, relaciones con otras tablas, etc. Para garantizar la portabilidad del código en Java y las consultas, Hibernate maneja un parámetro donde se especifica el dialecto de la base de datos representado por medio de una clase que se encargará de comunicarse con la base de datos en el lenguaje SQL que esta entiende.

- **Janino.** [21]

Es una herramienta java rápida y liviana, útil para compilar código Java en tiempo de ejecución soportando elementos del lenguaje de programación java entre los cuales están:

- Declaración de paquetes, declaraciones tipo `import`
- Herencia (`extends e implements`)
- Declaración de clases, campos, métodos y ejecución de estos. (codehaus).

Janino tiene una limitación importante es que es compatible con versiones de java inferiores a la cuarta (1.4), por lo tanto con él no se pueden usar características extra que tenga Java desde esta versión en adelante, como por ejemplo declaración de tipos parametrizados, clases enumeradas, anotaciones, *for each* entre otras.

En nuestro caso fue usado para compilar los bloques de código escritos en el editor que creamos para código java de la siguiente manera:

Tenemos una cadena de caracteres en donde se encuentra la declaración de una clase llamada Parámetros. Dentro de esta clase se encuentra declarado un método estático que retorna un mapa de objetos con los objetos java declarados por el usuario en el editor Java. Las declaraciones tipo *import* que el usuario haya declarado se situarán en la cadena de caracteres justo antes de la declaración de la clase parámetros.

[21] CODEHAUSE "what is Janino". {En linea}. {12 septiembre de 2013} disponible en: (http://docs.codehaus.org/display/JANINO/Home)

```
   ➔ Aquí van las declaraciones de tipo import
public class Parametros {
     public static void java.util.Map getParametros(){
          java.util.Map parametros = new java.util.HashMap();
   ➔ Aquí se agregan al mapa todos los objetos creados por el
     usuario en la consola Java mediante el método
     put(nombreParametro, valor);
          ...
     }
}
```

Finalmente, teniendo una cadena de caracteres que declara a una clase llamada `Parametros`, esta clase es compilada usando el Janino y cargada. Luego se accede al método `getParametros` de la clase cargada mediante el uso del API *Reflection* de Java.

Mediante *Reflection* se crea una referencia a la clase.

```
Class                    claseParametros                    =
compiler.getClassLoader().loadClass("Parametros");
```

Se ejecuta el método estático donde se crea el mapa con los objetos creados y que servirán de parámetros en la consulta JPQL.

```
Method metodo;
metodo = claseParametros.getDeclaredMethod("getParametros");
Map<String, Object> parametros;
parametros = (Map<String, Object>)
metodoGetParametros.invoke(claseParametros);
```

2.1.6 Herramientas de desarrollo

2.1.6.1 Enterprise Architect[22]

Es una herramienta desarrollada por *Sparx Systems* para crear los diagramas UML en la etapa de análisis y diseño.

2.1.6.2 NetBeans IDE [23]

Es un entorno de desarrollo integrado libre, hecho principalmente para el lenguaje de programación Java. Existe además un número importante de módulos para extenderlo. NetBeans IDE es un producto libre y gratuito sin restricciones de uso.

2.1.6.3 Subversion (SVN)[24]

Es un sistema de control de versiones diseñado específicamente para reemplazar al popular CVS. Es software libre bajo una licencia de tipo Apache/BSD y se le conoce también como *SVN* por ser el nombre de la herramienta utilizada en la línea de comando.

2.1.6.4 Google Code [25]

El alojamiento de proyectos en Google Code es un servicio de alojamiento de software libre rápido, fiable y sencillo.

Permite:

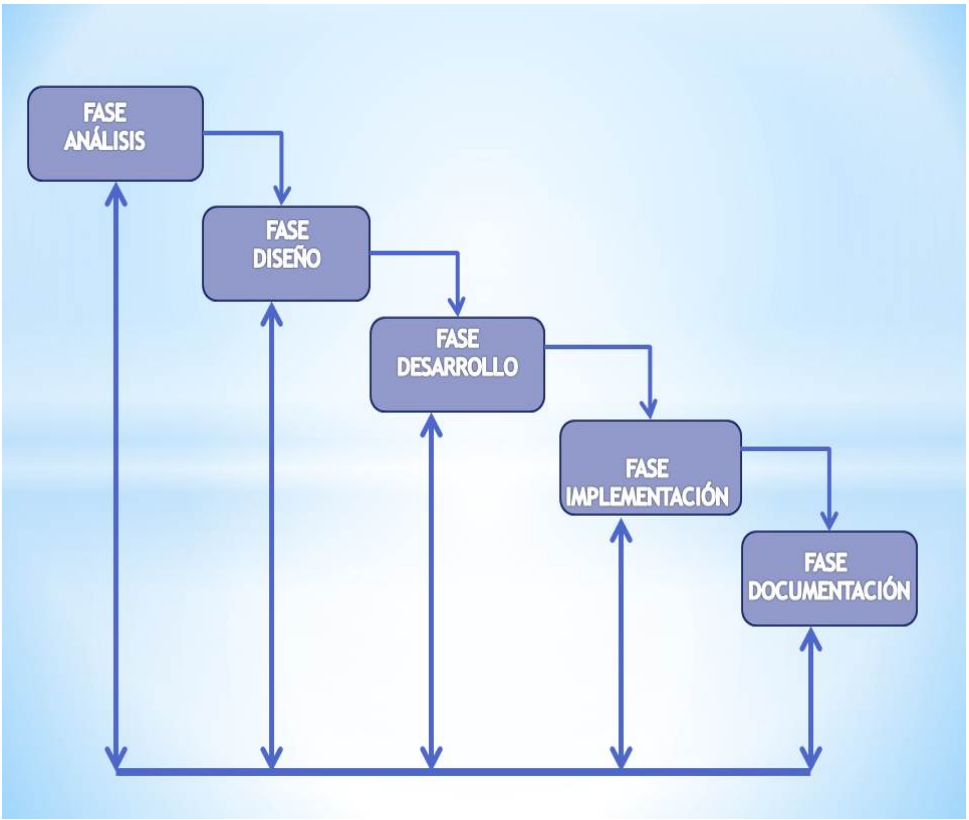
- Crear proyectos instantáneos sobre cualquier tema.
- Alojar código de Subversion con un 1 gigabyte de espacio de almacenamiento y admitir alojamiento para descargas con 2 gigabytes de espacio de almacenamiento.

[22] http://www.sparxsystems.com/
[23] https://netbeans.org/
[24] http://subversion.apache.org/
[25] https://code.google.com/intl/es/

- Consultar código fuente integrado y utilizar herramientas de revisión de código para facilitar la visualización de código, la revisión de contribuciones y el mantenimiento de una base de código de gran calidad.
- Realizar un seguimiento de problemas y búsquedas wiki de proyectos sencillas, pero flexibles y potentes, que pueden adaptarse a cualquier proceso de desarrollo.
- Marcar como destacados y actualizar flujos que facilitan el seguimiento de los proyectos y los desarrolladores que te interesan.

2.1.7 Metodología de desarrollo

Ilustración 12: metodología de desarrollo



Como metodología de desarrollo utilizamos el modelo en cascada el cual nos brindó tres beneficios

- El primero y principal es que los requerimientos están muy bien definidos.
- El segundo por su simplicidad.
- El tercero porque nos permitió avanzar siempre y cuando se cumplieran los objetivos de la fase anterior permitiéndonos así ver el progreso del proyecto por medio de una revisión al finalizar cada fase. Para la validación de los objetivos de cada una de las fases como los documentos entregables de las mismas, se realizaron mediante reuniones con los interesados reduciendo así las falencias o errores y por consiguiente asegurando el certero desarrollo del proyecto.

3. Desarrollo de la aplicación

3.1 Plan de trabajo

3.1.1 Análisis

3.1.1.1 Analizar las necesidades de los interesados detalladamente

Se realizaron reuniones constantes en las cuales los interesados expresaron el problema de no contar con una herramienta para probar consultas JPQL de manera ágil y con un entorno agradable, además con funcionalidades que ofrezcan comodidad al desarrollador.

Luego de un análisis conjunto entre interesados y desarrolladores, se procede con propuestas de posibles soluciones, aclarando el camino y evitando futuras discrepancias

3.1.1.2 Definición de los requisitos y alcance del sistema.

Consistió en dejar muy clara la manera en que se cumplirían los objetivos buscando siempre la conformidad de los interesados. El resultado de esta definición se encuentra plasmado en el acta de requisitos la cuál define las funciones de la aplicación a desarrollar.

3.1.1.3 Construcción del acta de requisitos.

Teniendo claros los requisitos y alcances se realizó un acta para la cual seleccionamos un modelo de "lista de deberes" con el fin de brindar claridad.

La herramienta debe:

- Permitir al usuario incluir las entidades (tablas mapeadas) a manejar en las consultas, empaquetadas en un archivo .jar.
- Tener un explorador de las entidades incluidas por el usuario, en el que se mostrarán agrupadas por archivo .jar en una estructura tipo árbol y se deberá mostrar los atributos persistentes de cada entidad.

- Permitir al usuario configurar una conexión con el servidor de base de datos para cualquiera los SGBD soportados por Hibernate.

- Tener un editor de código JPQL con las siguientes funcionalidades:
  - Guardar en un archivo las consultas creadas en el editor.
  - Abrir archivos guardados.
  - Copiar, cortar, y pegar texto en el editor.
  - Formatear y exportar las consultas para su uso directo en código JAVA
  - Autocompletar tanto sintaxis como atributos de las clases cargadas.

- Tener un editor de código java para la creación de objetos que serán usados como parámetros en las consultas.

- Mostrar los resultados de las consultas en una grilla, que permita ordenar los registros por cualquiera de sus campos.

- Mostrar los errores cometidos por el usuario.

- Implementar un documento en línea de ayuda sobre el manejo de las opciones de la herramienta.

3.1.1.4 Casos de uso documentación y diagrama.

Actores

Tabla 2: actores casos de uso

Actor	**Usuario**
Casos de uso	Establecer conexión, Agregar entidades, Ejecutar consulta, Eliminar entidades, Gestionar consulta.
Tipo	Primario
Descripción	Es el actor principal, un desarrollador que desee probar sus consultas JPQL sobre una base de datos relacional.

Casos de uso

Tabla 3: caso de uso iniciar aplicación.

Caso de uso	**Iniciar aplicación**
Actores	Usuario.
Tipo	Básico
Propósito	Hacer las revisiones necesarias para el correcto inicio de la aplicación antes de que la *Pantalla principal* esté disponible para el usuario.
Resumen	Revisa si existen las carpetas del sistema (jars, drivers, log y conf) ya que sin ellas la aplicación no puede seguir ejecutándose. También revisa, si existe un archivo de configuración y si hay jars en la carpeta jars.
Precondiciones	Ninguna
Flujo principal	1. Se revisa si existen las carpetas del sistema (jars, drivers, log y conf). 2. Se abre la *Pantalla principal* (P-1). 3. Si existen *jars* en la carpeta se cargan los *jars,* se llama al caso de uso *Configurar conexión*. De lo contrario se llama al caso de uso *Administrar Entidades.*
Subflujos	-
Excepciones	E1. Si las carpetas no existen se notifica al usuario y se cierra la aplicación. E2. Si no existe un archivo con los datos de conexión o si existe pero está dañado, se eliminará el archivo de conexión en caso de que exista y se le pedirá al usuario que ingrese los datos de la conexión. E3. Si la carga de un archivo *jar* falla, entonces se muestra la causa del error y se elimina el archivo *jar* de la carpeta de *jars* de la aplicación.

Tabla 4: caso uso configurar conexión

Caso de uso	**Configurar conexión**
Actores	Usuario
Tipo	Básico
Propósito	Establecer la conexión con la base de datos relacional.
Resumen	El usuario inicia el caso de uso el cuál le permite conectarse a la base de datos al especificar la URL de la conexión, el sistema gestor de la base de datos, el usuario y la contraseña de la base de datos, para establecer la conexión.
Precondiciones	Se debe haber cumplido el flujo principal del caso de uso Iniciar aplicación.
Flujo principal	1. Se muestra la pantalla *Configurar conexión (P-3)* al usuario con los campos URL de conexión, sistema de gestor de base de datos, usuario y contraseña. Si ya se estableció una conexión anteriormente entonces se cargará la información de esa conexión incluida la contraseña. 2. Si el usuario presiona el botón "conectar" entonces se establece la conexión con el *Servidor de base de datos.* 3. Se cierra la pantalla *Conexión con servidor (P-3),* se regresa a la *Pantalla principal (P-1),* habilitando la ejecución del caso de uso *Ejecutar consulta.*
Subflujos	
Excepciones	E1: Si la información de conexión es incorrecta se muestra un mensaje de error y se regresa a la misma pantalla (*Conexión con servidor)* para que el usuario introduzca una información válida.

Tabla 5: Tabla 5: caso de uso administrar entidades

Caso de uso	**Administrar entidades**
Actores	Usuario
Tipo	Básico
Propósito	Administrar los archivos *.jar* en los que se encuentran empaquetadas las entidades.
Resumen	El usuario inicia el caso de uso en el cual podrá importar y eliminar archivos *.jar* que contienen las entidades.
Precondiciones	Se debe haber cumplido el flujo principal del caso de uso iniciar aplicación.
Flujo principal	1. Se abre la pantalla *Administrar entidades* (P-2) (Se muestra una pantalla con un listado de archivos .jar importados al proyecto que contienen las entidades y el usuario tiene la posibilidad de eliminarlos o agregar más. Si el usuario pulsa el botón Agregar Entidades entonces se ejecuta el subflujo S-1. Si el usuario pulsa el botón Eliminar entidades entonces se ejecuta el subflujo S-2. Si el usuario pulsa Aceptar se guardan cambios en lista de entidades, se cierra esta pantalla y se muestra la *Pantalla principal* (P-1). Si el usuario pulsa Cancelar se ejecuta el subflujo S-3.
Subflujos	S-1: Se llama el caso de uso Agregar entidades. S-2: Se llama el caso de uso Eliminar entidades. S-3: Si el usuario pulsa Cancelar entonces todos los cambios realizados a la lista entidades ya sea que se haya eliminado o agregado un archivo *.jar*, son revertidos, se cierra la pantalla

	de *Administrar entidades (P-2)* y se regresa a la *Pantalla principal (P-1)*.
Excepciones	

Tabla 6: caso de uso agregar entidades

Caso de uso	**Agregar entidades**
Actores	Usuario
Tipo	Extensión
Propósito	Agregar las entidades que se encuentran empaquetadas en archivos .jar
Resumen	El usuario inicia el caso de uso mediante el cual podrá localizar en algún medio de almacenamiento local las entidades (empaquetadas en un archivo .jar) que desee importar a la aplicación.
Precondiciones	Se debe haber cumplido el flujo principal del caso de uso iniciar aplicación
Flujo principal	1. Se abre un selector de archivos donde el usuario podrá elegir el archivo *.jar* a importar y tiene dos opciones: si selecciona *Aceptar,* entonces el archivo *.jar* se importa en el proyecto momentáneamente, se cierra la *Pantalla de selección de archivo jar* (P-3) y se regresa al flujo principal del caso de uso Administrar Entidades. Si el usuario selecciona *Cancelar* entonces regresa al flujo principal del caso de uso Administrar Entidades.
Subflujos	

Excepciones	E1: Si se intenta agregar un archivo *.jar* con el mismo nombre que uno que ha sido previamente cargado, se notifica al usuario que el archivo ya existe y se pregunta si se desea sobrescribir el archivo *.jar* existente. Si la respuesta es sí, se sobrescribe el archivo *jar*, de lo contrario se regresa al flujo principal del caso de uso Administrar Entidades.

Tabla 7: caso de uso eliminar entidades

Caso de uso	**Eliminar entidades.**
Actores	Usuario.
Tipo	Extensión
Propósito	Eliminar un archivo *jar* de entidades que haya sido importado al proyecto.
Resumen	Se elimina un archivo *jar* de entidades existente de la aplicación.
Precondiciones	El usuario debió haber ejecutado el subflujo 2 (S-2) del caso de uso *Administrar entidades*, y tener seleccionado como mínimo un archivo *jar* de la lista de *jars* actuales que se muestra en pantalla.
Flujo principal	Se elimina momentáneamente el *jar* de la lista de *jars* y se pone en cola para ser eliminado de la carpeta en caso de que se guarden los cambios realizados en el caso de uso *Administrar entidades*.
Subflujos	--
Excepciones	--

Tabla 8: caso de uso ejecutar consulta

Caso de uso	**Ejecutar consulta**
Actores	Usuario
Tipo	Básico
Propósito	Ejecutar una consulta en la consola JPQL del proyecto.
Resumen	Toma la consulta de la consola JPQL del proyecto y la ejecuta
Precondiciones	El usuario debe tener entidades cargadas al proyecto y debe haber establecido una conexión con la base de datos.
Flujo principal	1. El usuario oprime el botón de *ejecutar consulta* (o realiza un acceso rápido mediante una combinación de teclas) que está en el panel de herramientas de la aplicación. 2. Se ejecuta todo el código contenido en el editor JAVA y esta retorna los objetos creados y los envía al editor JPQL para que sean usados como parámetros en la consulta. 3. Se ejecuta la consulta JPQL que se encuentra en el editor JPQL. 4. Los resultados de la consulta se despliegan en la pantalla de resultados donde el usuario podrá ordenar los resultados por columnas de acuerdo al tipo de valor contenido en las mismas.
Subflujos	--
Excepciones	E1: Existe un error de sintaxis en el editor JAVA. Se muestra el error en la consola de errores del JPangolin. Y finaliza el caso de uso. E2: Existe un error de sintaxis en el editor JPQL. Se muestra el error en la consola de errores del JPangolin. Y finaliza el caso de uso.

Tabla 9: Caso de uso gestionar consulta

Caso de uso	**Gestionar consulta**
Actores	Usuario
Tipo	Básico
Propósito	Gestiona el guardado, el cargue y formateo de las consultas JPQL.
Resumen	Se encarga de guardar y cargar consultas realizadas previamente por el usuario, además de formatearlas para ser exportadas al proyecto en el que se usarán.
Precondiciones	Se debe haber cumplido el flujo principal del caso de uso *Iniciar aplicación.*
Flujo principal	Se muestran al usuario las siguientes opciones: - Cargar archivo de consulta. - Guardar consulta en archivo. - Exportar consulta. Si el usuario elige cargar archivo de consulta se ejecuta el subflujo *S-1*. Si el usuario elige guardar una consulta se ejecuta el subflujo *S-2*. Si el usuario selecciona exportar consulta se ejecuta el subflujo *S-3*.
Subflujos	S-1: Se mostrará un explorador de archivos en el cuál el usuario escogerá el archivo de consultas. Dicha consulta será insertada en la consola JPQL y si existe código Java, será insertado en la consola Java. S-2: Se mostrará un explorador de archivos para que elija el destino del archivo y proporcione el nombre del mismo. S-3: Se mostrará una ventana con la consulta formateada lista para ser copiada en el proyecto en que se usará.
Excepciones	E1: El archivo de consultas se encuentra dañado o no es

	válido. El caso de uso termina avisándole al usuario que no es posible cargar dicho archivo.

A continuación se mostrarán las pantallas que componen la aplicación para establecer la relación entre las mismas y los casos de uso mencionados anteriormente.

Pantalla (P1) principal:

Esta pantalla es la interfaz principal, es la que se va a abrir cuando arranca la aplicación, desde esta se podrá acceder a todos los menús, en ella se realizaran las consultas y se observaran las entidades organizadas en el árbol.

Pantalla (P2): Administrar Entidades:

Esta pantalla le brindara al usuario la opción de administrar sus entidades, contiene cuatro botones (agregar, eliminar, aceptar, cancelar) .El botón agregar mostrara un explorador de archivos para seleccionar los jars a cargar recordando la última ruta desde donde se cargaron jars, eliminar borrara temporalmente los archivos seleccionados y los botones aceptar y cancelar sencillamente guardaran o revertirán los cambios hechos en la lista.

Pantalla (P3) configurar conexión:

Esta pantalla recibirá los datos de conexión por parte del usuario, tendrá la funcionalidad de recordar datos de la última conexión los cuales serán cargados en el momento de arrancar la aplicación solo si hay la aceptación del usuario.

Pantalla (P4) Opciones:

Esta pantalla permitirá al usuario elegir su configuración preferida para el uso de la aplicación, elegirá teclas de acceso rápido, teclas de funcionalidades y opciones como seleccionar un directorio de usuario para guardar sus consultas.

Ilustración 13: (P-1) prototipo pantalla principal

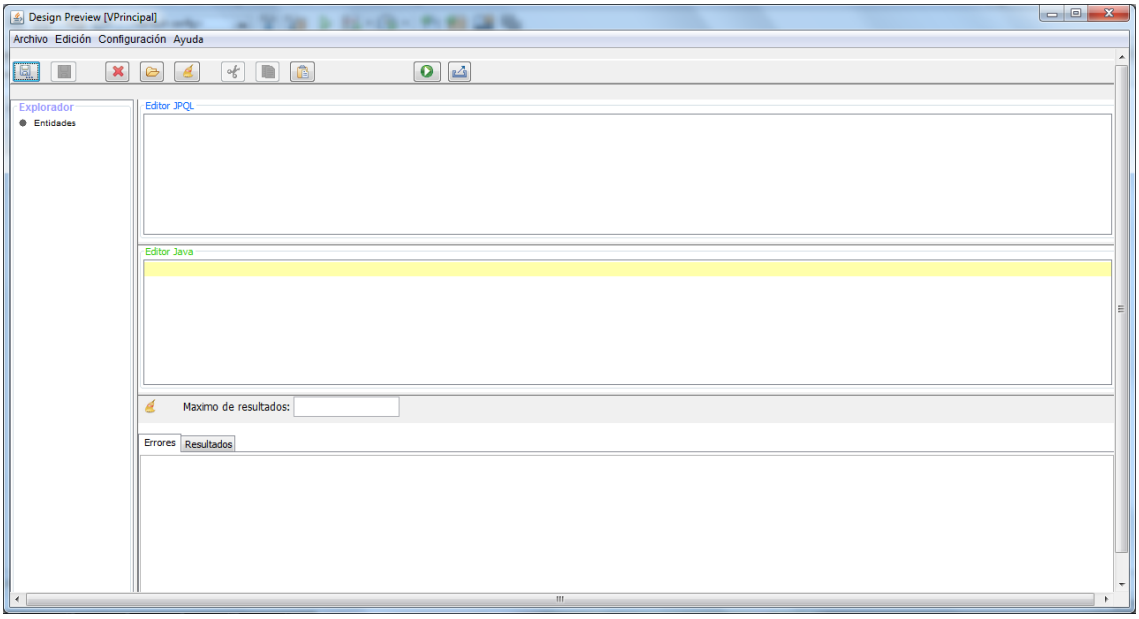


Ilustración 14: (P-2) prototipo pantalla administrar entidades

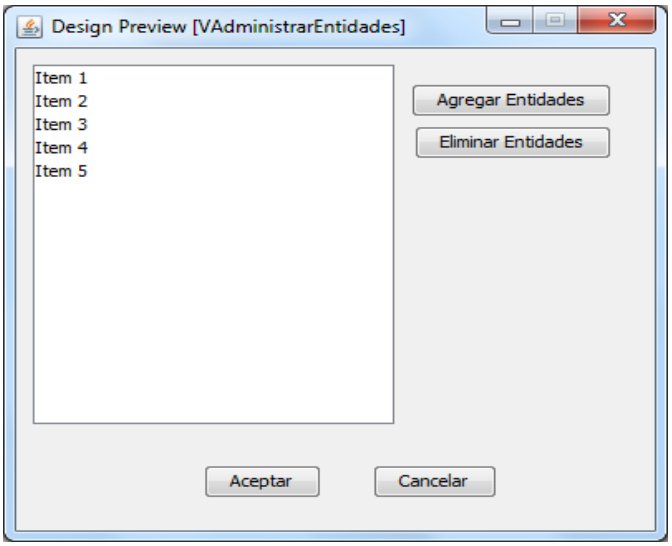


Ilustración 15: (P-3) prototipo pantalla configurar conexión

Ilustración 16: (P-4) prototipo pantalla opciones

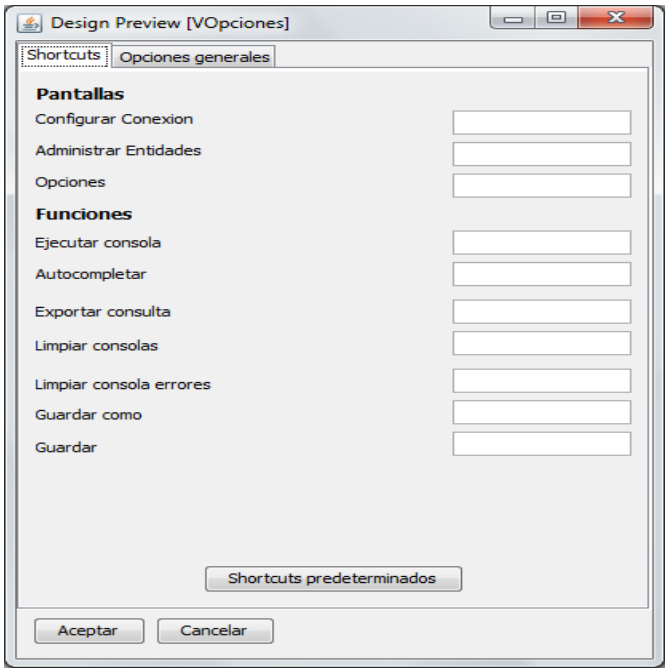
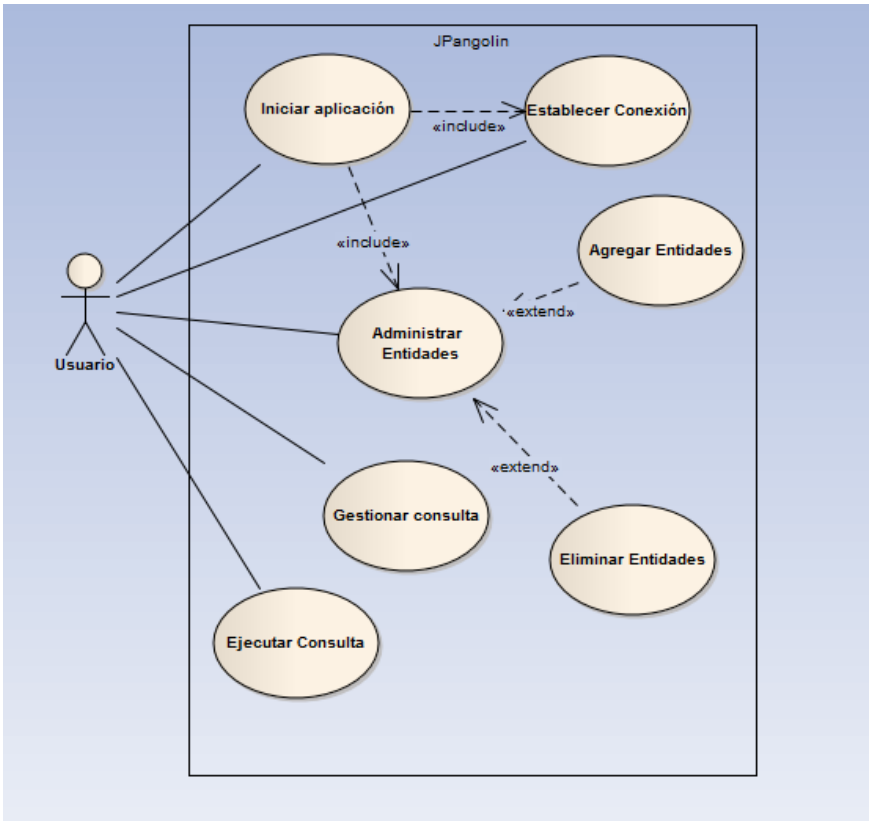


Ilustración 17: Diagrama de casos de uso



3.1.1.5 Selección de las herramientas a utilizar.

Para cada una de las herramientas a continuación veremos las razones por las cuales las elegimos.

- Enterprise Architect:
  Esta herramienta la seleccionamos por la comodidad que nos brinda en el momento de realizar diagramas UML tanto para análisis como para diseño.

- Java SE 7
  Elegimos este lenguaje de programación las siguientes características:

  1. Orientado a objetos.
  2. Flexible
  3. Multiplataforma
  4. Gratuito

5. Abierto
6. Expandible
7. Disponibilidad de librerías
8. Soporte y documentación

- NetBeans IDE: esta herramienta fue elegida por que es un IDE en el cual ya teníamos algún grado de experiencia conociendo las facilidades que brinda para el desarrollo de software en java y además por brindarnos un constructor de interfaz gráfica (*GUI Builder*) para diseñar un prototipo al cual paso a paso le incorporamos funcionalidades.

- Subversion (SVN): elegido por ser una herramienta gratuita elegida para gestionar el control de versiones durante el desarrollo de la aplicación permitiendo, entre otras cosas, llevar un completo control de las adecuaciones que realizábamos cada uno de los desarrolladores. Además se acopla con NetBeans IDE desde el cual podemos gestionar las actualizaciones.

- Google Code: elegido por servirnos como repositorio para nuestro servidor Subversion (SVN) brindándonos disponibilidad 24/7, haciendo más cómodo el trabajo en equipo de los desarrolladores y también sin costo alguno.

3.1.1.6 Documentación, entrenamiento y capacitación con las herramientas.

A continuación se muestra una tabla en la que se especifican las fuentes de documentación, una descripción y su utilidad para nuestro proyecto.

Tabla 10: Fuentes de documentación.

Descripción	**Utilidad en JPangolin**
Tutorial que nos ofrece Oracle actualizado a cada versión de java	Como nuestra herramienta fue desarrollado en java, este tutorial fue una guía constante ante cualquier duda sobre el lenguaje.
Serie de tutorías sobre NetBeans	De vital ayuda para realizar las GUIs de nuestro proyecto
Libro sobre mapeo ORM	Útil para entender los conceptos de mapeo y utilizarlos a la hora de manipular entidades en la aplicación.
Libro sobre mapeo ORM con Hibernarte	Útil para entender conceptos de mapeo con Hibernarte como implementación JPA.
Referencia del lenguaje JQPL	Se utilizó para entender la sintaxis del lenguaje JPQL y así poder implementar la función de Autocompletar.
Stackoverflow sitio de preguntas y respuestas para programadores.	Para efectos de dudas y preguntas puntuales sobre Hibernarte, JPQL y Java.

3.1.2 Fase diseño.

3.1.2.1 Estudio detallado del acta de requerimientos para iniciar diseño

Este proceso consistió en añadir detalles al análisis y tomar decisiones buscando un buen diseño del sistema.

A continuación se muestran la organización de las clases a utilizar y una breve descripción de cada una.

Tabla 11: organización de las clases

	Clasificación	Clases encargadas
	Principal.	• LPrincipal.

JPangolin		• LPrincipal.
	Administrar entidades.	• LAdministrarEntidades. • VAdministrarEntidades.
	Configurar conexión.	• LConfigurarConexion. • VConfigurarConexion.
	Opciones de usuario.	• LOpciones. • VOpciones.
	Datos.	• ClaseEJB3
	Logica.	• Logica.
	Pantallas.	• Pantallas

Tabla 12: descripción clase JPangolin

Nombre: **JPangolin**
Descripción: **es la clase principal de la aplicación en ella se encuentra el método main y desde ahí se abre la pantalla principal**
Estereotipo: **control**

Tabla 13: descripción clase EJB3

Nombre: **Datos.EJB3**
Esta clase contiene una instancia de la clase EJB3Configuration que a su vez contiene todas las entidades cargadas y toda la configuración que Hibernate necesitará (datos de conexión, dialecto de la base de datos, etc.) para gestionar el mapeo y la comunicación con la base de datos. La aplicación sólo contiene una instancia de esta clase y es usada para crear el objeto EntityManager.
Estereotipo: **control**

Tabla 14: descripción clase Logica

Nombre: **Logica**
Descripción: **esta clase contiene los métodos y variables comunes para toda la aplicación. Todos los métodos y variables públicas son estáticos.**
Estereotipo: **control**

Tabla 15: descripción clase LAdministrarEntidades

Nombre: **LAdministrarEntidades**
Descripción: **contiene la lógica que da funcionalidad a la pantalla administrar entidades, en ella están establecidos los métodos que cargan y eliminan dichas entidades.**
Estereotipo: **control**

Tabla 16: descripción clase LConfigurarConexion

Nombre: **LConfigurarConexion**
Descripción: **contiene la lógica que da funcionalidad a la pantalla configurar conexión, en ella están los métodos encargados de gestionar las conexiones y sus respectivos archivos.**
Estereotipo: **control**

Tabla 17: descripción clase LPrincipal

Nombre: **LPrincipal**
Descripción: **contiene los métodos necesarios para que todos los componentes de la pantalla principal funcionen correctamente.**
Estereotipo: **control**

Tabla 18: descripción clase LOpciones

Nombre: **LOpciones**
Descripción: **encargada de la funcionalidad de los componentes de la pantalla opciones, gestionando todas las configuraciones hechas por el usuario.**
Estereotipo: **control**

Tabla 19: descripción clase Pantallas

Nombre: **Pantallas**
Descripción: **es la clase encargada de la gestión de pantallas (abrir, cerrar) con el fin de tener solo una instancia de cada pantalla a la vez.**
Estereotipo: **control**

Tabla 20: descripción clase VAdministrarEntidades

Nombre: **VAdministrarEntidades**
Descripción: **es la interfaz gráfica para administrar entidades**
Estereotipo: **borde**

Tabla 21: descripción clase VconfigurarConexion

Nombre: **VConfigurarConexion**
Descripción: **es la interfaz gráfica para configurar conexión**
Estereotipo: **borde**

Tabla 22: descripción clase VPrincipal

Nombre: **VPrincipal**
Descripción: **es la interfaz gráfica principal**
Estereotipo: **borde**

Tabla 23: descripción clase VOpciones

Nombre: **VOpciones**
Descripción: **es la interfaz gráfica de opciones**
Estereotipo: **borde**

3.1.2.2 diseño de pantallas para captura y presentación de datos

Con el fin de ofrecer más claridad al lector estas pantallas (P1, P2, P3, P4) se presentan en la sección 3.1.1.4 casos de uso documentación y diagrama

3.1.2.3 Definir el modelo funcional (menús del sistema).

Ilustración 18: menús del sistema

3.1.2.4 Confrontación de especificaciones de diseño con acta de requerimientos.

En la siguiente tabla podemos observar que requisitos se cumplen en cada menú verificando así la concordancia entre análisis y diseño.

Tabla 24: menús vs requerimientos que cumple

MENU	REQUERIMIENTOS QUE CUMPLE
Principal	✓ Tener un explorador de las entidades incluidas por el usuario, en el que se mostrarán agrupadas por archivo .jar en una estructura tipo árbol y se deberá mostrar los atributos persistentes de cada entidad. ✓ Tener un editor de código JPQL que permita autocompletar. ✓ Tener un editor de código Java. ✓ Mostrar los resultados de las consultas en una grilla que permita el ordenamiento por columnas. ✓ Mostrar los errores cometidos por el usuario. ✓ Ofrecer la opción de guardar, abrir y exportar consultas. ✓ Ofrecer un documento de ayuda en línea.

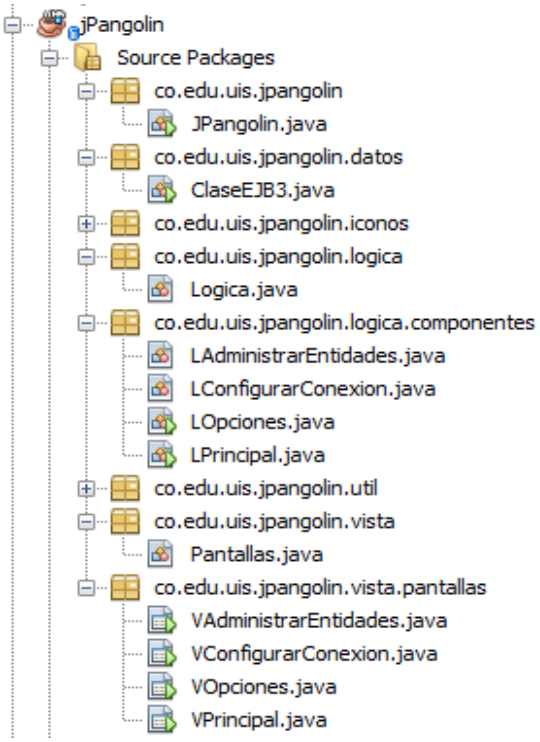
administrar entidades	✓ Permitir al usuario incluir y eliminar las entidades (tablas mapeadas) a manejar en las consultas, empaquetadas en un archivo *.jar.*
configurar conexión	✓ Permitir al usuario configurar una conexión con el servidor de base de datos para cualquiera los SGBD soportados por Hibernate.
configuraciones	✓ Permitir seleccionar opciones sobre combinación de teclas de acceso rápido

3.1.3 Fase de desarrollo

3.1.3.1 Estudio detallado de las especificaciones de diseño para iniciar el desarrollo.

En esta fase se crearon las clases con la organización que se muestra en la ilustración 5 (organización de las clases). Cumpliendo así las especificaciones de diseño, para esta tarea utilizamos nuestra herramienta NetBeans IDE, puede notarse además que ubicamos nuestras clases dentro de una organizada estructura de paquetes.

Ilustración 19: explorador de proyectos de NetBeans.



3.1.3.2 Desarrollar prototipo afín al diseño.

Esta fase consistió en darle funcionalidad a las clases del paquete *co.edu.jpangolin.vista* las cuales son *JFrame* encargados de la interacción con el usuario, dicha funcionalidad consiste en correlacionar todos los *JFrame* y asignarles eventos al momento de cerrar, abrir y oprimir botones sin ejecutar la lógica de la aplicación, solamente para efectos de visualización. El prototipo se encuentra en un medio digital adjunto a este documento.

3.1.3.3 Escritura de los programas que brindan la funcionalidad al prototipo.

Al llegar a esta etapa y teniendo en cuenta que ya se cuenta con un prototipo, y una estructura de clases organizadas se hace más fácil el trabajo pues se le puede dar funcionalidad teniendo en cuenta los requerimientos y cumpliéndolos uno por uno.

3.1.3.4 Realización de pruebas

Se ejecutaron dos tipos de pruebas: las realizadas por los desarrolladores en las se buscaban problemas de funcionamiento y pruebas realizadas por el todo el equipo de trabajo (director, codirector y desarrolladores) en las cuales se revisaba el correcto funcionamiento y el cumplimiento de los requerimientos. En seguida se presenta un listado de algunas de las pruebas más comunes.

- Probar menús y relaciones entre diferentes pantallas: fueron las primeras pruebas realizadas en las que se verificó que cada pantalla se abriera y se cerrara cuando se deseaba y mediante los eventos que las administran.
- Probar gestión de entidades (carga y eliminación de jars): consistió en realizar el cargue de varios archivos jar verificando que estos archivos fueran copiados a la carpeta *jars* de la aplicación una vez se efectuara la carga de los mismos. También se verificó que una vez el usuario decidiera eliminar uno o más archivos jar, estos fueran también eliminados de la carpeta *jars* al momento de efectuarse los cambios (cuando el usuario hace [clic] en el botón aceptar de la pantalla Administrar Entidades). Se detectaron y corrigieron errores al momento del cargue de un jar cuando este contiene una clase que hace referencia a otra clase (contenida en otro jar) que no se encuentra cargada aún al *classpath*, por lo tanto se definió un orden de cargue de los jars.
- Probar conexión con diferentes gestores de bases de datos: Se realizaron pruebas de conexión en las cuales se verificó que tanto los drivers como los dialectos estuviesen disponibles al momento de realizar la conexión y crear el EntityManager.
- Probar consultas (la función autocompletar del editor JPQL, salidas de resultados y errores): se ejecutaron consultas tipo SELECT, UPDATE y DELETE utilizando como referencia la documentación que ORACLE ofrece (ORACLE). Dicha documentación describe el lenguaje JPQL mediante la notación Backus-Naur (*Backus-Nauf Form*, BNF) todas las posibles combinaciones que se pueden generar con una consulta JPQL.

3.1.4 Fase de implementación

3.1.4.1 Realizar seguimiento detallado al desempeño.

Esta fase y la inmediatamente anterior se complementan pues el seguimiento de desempeño consiste en pruebas realizadas y en la documentación de los errores presentados en las mismas y su solución.

3.1.4.2 Dar solución a problemas presentados.

Para tramitar los problemas se creó una un formulario utilizado por los desarrolladores para tener un completo control sobre todos los problemas presentados con fecha de ocurrencia y fecha de solución.

Tabla 25: gestión de errores

Descripción del problema	Fecha de detección	Descripción de la solución	Fecha de la solución
Consumo excesivo de memoria al momento de la ejecución y el procesamiento de los resultados una consulta JPQL.	03/07/2013	Cambio del formato de la consulta por un equivalente que reduce significativamente la lógica necesaria para procesar los resultados.	10/07/2013
Falla en el borrado de un jar cargado al Classpath en el sistema operativo Windows.	04/07/2013	Creación de un Classloader al cuál se cargan las clases y el cuál se renueva cada vez que se efectúen cambios sobre las entidades en la carpeta jars.	13/07/2013
Referencia circular en las clases mapeadas	20/09/2013	Referencia rota al detectarse la repetición	25/09/2013

al momento de formatear la consulta previo a su ejecución.		en las clases.	
Problema de referencia en los jars cargados.	14/08/2013	Creación de un archivo properties que contenga el orden de carga de los jars.	16/08/2013
Listado de campos estáticos y con la anotación @Transient.	04/09/2013	Creación de un método que indica la validez de cada uno de los campos que se listan en el autocompletar.	06/09/2013
Problema de conexión con el motor de base de datos Informix.	20/08/2013	Carga de la clase del Driver del SGBD al classpath previo a la conexión con el mismo.	24/09/2013

3.1.4.3 Realizar ajustes necesarios.

Luego de la detección de problemas y estudio de estos se realizan los cambios necesarios los cuales quedan reportados tanto en la tarjeta estipulada para control de problemas como en los registros del servidor de versiones, en este caso Subversion SVN.

3.1.4.4 Realimentar el proyecto con las fases previas.

Esta fase consiste en una reunión entre interesados y equipo de trabajo en la que se analiza si el resultado final era lo que esperaban los interesados de no ser así se revisa el aspecto crítico y se retrocede en las fases para encontrar en donde estuvo la discrepancia y replantear.

3.1.5 Fase de documentación

3.1.5.1 Creación de la documentación en línea sobre el manejo de las opciones de la herramienta para el usuario final.

En la interfaz principal se encuentra un menú que llevara al usuario una página creada en código HTML como una página web sencilla, en la que se explica al usuario cada una de las funcionalidades de la herramienta. El documento se puede acceder desde la aplicación desde el menú Ayuda – Documento de Ayuda, o mediante la tecla F1. En seguida se abre el programa predeterminado del sistema operativo del usuario para leer archivos HTML.

El documento se titula Guía de Usuario y está dividido en diez secciones:

- Instalación: describe los pasos preliminares a la ejecución de la aplicación.
- Al iniciar la aplicación: describe el flujo de la aplicación al iniciar y los posibles errores que puedan presentarse.
- La interfaz de usuario: enumera y explica la interfaz de usuario contenida en la pantalla principal: los menús y botones que esta contiene.
- Las opciones de configuración de la aplicación: describe cómo puede el usuario cambiar algunas opciones predeterminadas en la aplicación.
- Agregar y eliminar jars: muestra cómo agregar y eliminar jars y los posibles errores que puedan existir.
- Establecer la conexión con BD: describe la conexión con el servidor desde la pantalla Configurar Conexión.
- El editor JPQL: describe el editor, la función autocompletar y lo que el usuario debe tener en cuenta al momento de escribir una consulta en el editor.
- El editor Java: describe el editor y la sintaxis de creación de los objetos.
- Ejecutar una consulta: muestra cómo ejecutar una consulta y dónde visualizar los resultados.
- Visualización de resultados: informa al usuario sobre la visualización de los resultados y el ordenamiento de los mismos.

CONCLUSIONES

- Actualmente ORM es una excelente opción para implementar una aplicación de software: por una parte, se hace uso del paradigma orientado a objetos, aprovechando las ventajas de flexibilidad, mantenimiento y reutilización. Por otra parte, se dispone una base de datos relacional, aprovechando su madurez y su estandarización.

- JPangolin se postula como una excelente herramienta complementaria para el desarrollo de aplicaciones que usen Java e Hibernarte como implementación de JPA pues brinda al desarrollador la función de centrarse por completo en la elaboración y ejecución de las consultas a la base de datos que implementará en la aplicación a desarrollar.

RECOMENDACIONES

- Todo software debe ser desarrollado en función de patrones y estándares, y reutilizando al máximo el código. Esto nos permite centrarnos en la lógica de negocio, reducir los tiempos de desarrollo, y aumentar la calidad del resultado.

- Se deben revisar periódicamente las versiones más recientes de las herramientas que hacen parte de JPangolin tales como Hibernate, Janino, entre otros; para proveer actualizaciones que contribuyan a mejorar el rendimiento.

- Se propone realizar las siguientes mejoras en la aplicación a futuro:
    - Creación de varios espacios de trabajo con conexiones a SGBD independientes para que el usuario pueda separar sus jars por proyectos.
    - Enriquecer la opción de exportar consulta.
    - Hacer que el documento de ayuda al usuario sea más dinámico, permitiendo que el usuario consulte un contenido específico mediante un panel de búsqueda.
    - Hacer una documentación de errores más exhaustiva.
    - Mejorar la herramienta de modo que no sólo pueda trabajar con clases mapeadas con anotaciones sino también mediante archivos XML.
    - Permitir que el usuario pueda escoger qué implementación de JPA usará JPangolin.

ANEXOS

El prototipo no funcional y los archivos de la ayuda en línea de la herramienta, se encuentran en una un CD aparte junto con un ejecutable de la herramienta desarrollada.

BIBLIOGRAFÍA

- BAUER, Christian; GAVIN, King. Java Persistente with Hibernate . (2007). Manning Publications.
- CODEHAUS. Proyecto Janino. {10 de julio 2013}. disponible en: http://docs.codehaus.org/display/JANINO/Home
- ENEKO GONZALES,Benito. *java hispano*. { 12 de septiembre de 2013}, disponible en: http://www.javahispano.org/storage/contenidos/reflection.pdf
- Joyanes Aguilar, Luis; Zahonero Martínez, Ignacio. Programación en JAVA 2: Algoritmos, estructuras de datos y programación orientada a objetos. Madrid, McGraw-Hill. 2002.
- *MSDN Library*. {10 de agosto de 2013} disponible en: http://msdn2.microsoft.com/es-es/library/ms233836(VS.80).aspx
- ORACLE. The java EE 7 tutorial. disponible en: http://docs.oracle.com/cd/E11035_01/kodo41/full/html/ejb3_langref.html
- Piattini Velthuis, Mario G. Análisis y diseño de aplicaciones informáticas de gestión: Una perspectiva de ingeniería del software. México, Alfaomega. 2007.
- Scott W, A. *Mapping objects to relational databases: O/R mapping in detail.* {en linea}. {12 deseptiembre de 2013}. Disponible en: http://www.agiledata.org/essays/mappingObjects.html
- viscuoso, German{10 septiembre de 2013 }.disponible en: http://www.db4o.com/espanol/db4o%20Whitepaper%20-%20Bases%20de%20Objetos.pdf