Universidad de las Ciencias Informáticas

Título: "Componente de acceso a datos para soluciones de emisión de documentos de identificación"

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas.

AUTOR: **Leandro Regueiro Martínez**

TUTOR: **Ing. Irving D. Cao**

TUTOR: **Ing. Yoan Suárez Blanco**

Ciudad de La Habana, Cuba.

Junio de 2009

Nuestra recompensa se encuentra en el esfuerzo y no en el resultado. Un esfuerzo total es una victoria completa.

Mahatma Gandhi.

DECLARACIÓN DE AUTORÍA

Declaro ser el único autor del trabajo titulado:

Componente de acceso a datos para soluciones de emisión de documentos de identificación y autorizo a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

Leandro Regueiro Martínez	Ing. Irving D. Cao	Ing. Yoan Suárez Blanco
_____	_____	_____
Firma del Autor	Firma del tutor	Firma del tutor

DEDICATORIA

A los seres que más quiero en la vida: Elba, Elián y a mis abuelas.

AGRADECIMIENTOS

A la Universidad de las Ciencias Informáticas por formarme como profesional y persona de bien.

A toda mi familia por brindarme toda su fuerza, amor y apoyo incondicional.

A los tutores y todos mis amigos.

Y aquellas personas que de una forma u otra me ayudaron en la realización de este trabajo de diploma.

RESUMEN

El Centro de Identificación y Seguridad Digital de la Universidad de las Ciencias Informáticas desea desarrollar una aplicación informática, sobre la base del software libre, para automatizar los procesos involucrados en la emisión de documentos de identificación. Uno de los aspectos por definir es la estrategia que se establecerá para implementar el acceso a datos teniendo en cuenta que durante el proceso de construcción de este sistema, se producirán cambios en los modelos de datos obligando a modificar las capas de acceso a datos referenciadas por los programadores de la solución. En el presente trabajo de diploma se expone el desarrollo de un componente que permitirá implementar el acceso a datos del Sistema de Emisión de Documentos de Identificación (SEDI), con el menor costo posible en cuanto al tiempo y el esfuerzo empleados por los programadores para terminar o modificar las capas de acceso a datos del SEDI.

El componente de acceso a datos utilizará las capacidades del Framework Hibernate 3 para gestionar el acceso a la base de datos, además se utilizará AspectJ (extensión del lenguaje Java para programar orientado a aspectos) para modularizar el código dedicado a las transacciones de base de datos. Las principales funcionalidades del componente serán: gestionar las conexiones para acceder a la base de datos, gestionar las transacciones de base de datos, exponer métodos de soporte para que el programador de persistencia implemente las interfaces de acceso a datos del SEDI.

Palabras claves: Centro de Identificación y Seguridad Digital, Sistema de Emisión de Documentos de Identificación, componente de acceso a datos, Framework Hibernate 3, AspectJ.

ÍNDICE

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

En la actualidad la mayoría de las aplicaciones empresariales guardan y recuperan información que deben perdurar una vez finalizada la ejecución de las mismas. Toda la información que poseen las empresas, generalmente está almacenada en forma de tablas que comparten datos y conforman bases de datos relacionales. El acceso a esta información toma un papel importante en las operaciones rutinarias de cada empresa, por lo que la forma de explotar estos datos tiene una repercusión directa en la eficiencia de la misma. Dada la trascendencia que tiene el acceso a dicha información, se puede afirmar que la forma de realizar consultas a las bases de datos es tan importante como el diseño de sus tablas.

El Centro de Identificación y Seguridad Digital de la Universidad de las Ciencias Informáticas pretende desarrollar una solución informática, sobre la base del software libre, que permita automatizar los procesos involucrados en la emisión de documentos de identificación. Este sistema no es una excepción a la necesidad de almacenar información en una base de datos relacional. La construcción de este sistema formará parte de un proceso de desarrollo iterativo e incremental, por esta razón en muchas ocasiones durante el proceso de desarrollo del sistema se pueden presentar cambios en el modelo de datos. Esto obliga a modificar las capas de acceso a datos, referenciadas por los desarrolladores de la solución para interactuar con la base de datos. Todos estos cambios requieren de tiempo y recursos humanos cuyos costos pudieran reducirse, de contar con un componente que agilice este proceso, libre de pagos y de portabilidad notable.

Partiendo de lo anteriormente planteado, se presenta como **situación problemática** que los posibles cambios que se producen en las bases de datos relacionales durante el desarrollo de soluciones informáticas para los procesos de emisión de documentos de identificación, provocan que los programadores reestructuren las capas de acceso a datos tantas veces mientras se modifiquen estas bases de datos. El esfuerzo y tiempo que emplean estos programadores para terminar una capa de acceso a datos resulta en ocasiones costoso. Por esta razón, surge la necesidad de crear un componente que minimice la complejidad de esta tarea.

Debido a esta situación se plantea como **problema a resolver,** disminuir el esfuerzo y tiempo de desarrollo del acceso a datos empleados en la construcción del sistema informático para la emisión de documentos de identificación.

Por tanto el presente trabajo centra su **objeto de estudio** en el acceso a datos, derivándose de este como **campo de acción** el acceso a datos en soluciones de emisión de documentos de identificación.

Para dar solución al problema existente, se ha tomado como **objetivo general**, desarrollar un componente de acceso a datos que facilite el trabajo de los programadores de la capa de persistencia en soluciones de emisión de documentos de identificación. Para dar cumplimiento al objetivo general se han trazado como **objetivos específicos:**

- Seleccionar una herramienta de persistencia que permita eliminar la diferencia de impedancia entre el mundo relacional de las base datos y el mundo orientado a objetos.
- Manejar transacciones de base de datos utilizando programación orientada a aspectos.
- Realizar diseño e implementación del componente.
- Realizar pruebas unitarias al componente de acceso a datos.

Se considera como **idea a defender** que, el desarrollo de un componente de acceso a datos permitirá minimizar esfuerzo y tiempo de desarrollo que se consumen en la realización de capas de acceso a datos en las soluciones de emisión de documentos de identificación.

Las **tareas de la investigación** a desarrollar para cumplir los objetivos son las siguientes:

- Revisión bibliográfica del tema.
- Descripción de los lenguajes, tecnologías y herramientas, a utilizar en la realización de la propuesta de solución.
- Estudio y análisis de las tendencias actuales de acceso a datos. Patrones, técnicas y mecanismos.

- Estudio de la programación orientada a aspectos. Específicamente el lenguaje AspectJ.
- Estudio de la metodología nombrada: Desarrollo Basado en Rasgos (FDD por sus siglas en inglés).
- Desarrollo del componente de acceso a datos.

Para el desarrollo de las tareas de investigación se utilizaron métodos científicos que permitieron mediante la deducción lógica, análisis y síntesis, llegar a conclusiones a partir de todos los elementos que conforman el objeto de investigación, en este caso el acceso a datos. Por lo que se utilizó el **método inductivo** y como métodos de soporte para el razonamiento lógico el **método analítico**: que permitió estudiar y analizar de forma separadas los mecanismos, la técnica de mapeo objeto-relacional, herramientas que existen para acceder a una base de datos y problemas implícitos a la hora de implementar el acceso a datos. Como otro método de soporte se utilizó el **método sistémico**, que permitió sobre la base del análisis anterior la reunión y selección de los elementos estudiados para conformar el componente de acceso a datos.

Producto del conocimiento directo de acceso a datos y experiencia en este campo se utilizó como **método empírico**, el **método observacional**, y se aplicó también el **método de simulación** (método experimental controlado), al utilizar el componente para implementar el acceso a datos a una base de datos experimental.

El presente documento está estructurado en cuatro capítulos:

En el capítulo uno se incluye el estado del arte acerca del acceso a base de datos. Se presentan las tendencias actuales para el almacenamiento de la información, los mecanismos estudiados para acceder a esta información así como una de las técnicas de acceso a datos de gran utilidad para acceder a base de datos relacionales. En este capítulo también se presentan las tecnologías y herramientas utilizadas para el desarrollo del componente, así como la metodología de desarrollo.

En el capítulo dos se identifican las características y comportamiento (lista de rasgos) del componente de acceso a datos a partir de la realización de los dos primeros procesos que plantea la metodología de desarrollo utilizada. También se planifica la lista de rasgos construida para posteriormente diseñar e implementar el componente en base a esta lista.

En el capítulo tres se describe la arquitectura utilizada para desarrollar el componente, así como el patrón de diseño aplicado y la definición de un aspecto para resolver el problema de código disperso de las transacciones de base de datos. También se muestran los diagramas de clases del diseño a partir de la lista de rasgos y los diagramas de secuencia de algunos rasgos pertinentes, además del modelo de despliegue.

En el capítulo cuatro se describen las pruebas unitarias realizadas durante la implementación y después de desarrollado el componente.

.

CAPÍTULO I. FUNDAMENTACIÓN TEÓRICA

1.1. Introducción

Son diversas las soluciones que han surgido para el almacenamiento y el acceso a la información en el desarrollo de un sistema informático. Las soluciones para simplificar el acceso a datos desde los lenguajes de programación más usuales han sido muchas y de complejidad variable, generalmente dependiendo de un cierto fabricante y, en muchas ocasiones ligadas a una determinada plataforma, paradigma, sistema operativo, o incluso lenguaje. En este capítulo se hace un estudio de mecanismos, una técnica y soluciones que implementan esta técnica más utilizada en la actualidad para el acceso a la información en sistemas de base de datos. También se presenta la metodología de desarrollo, las herramientas y tecnologías empleadas para el desarrollo del componente.

1.2. Conceptos asociados al dominio del problema

Para mejor entendimiento y desenvolvimiento de las áreas temáticas que se abordarán en el presente y posteriores capítulos, se mostrarán una serie de conceptos identificados durante la investigación realizada.

1.2.1. Diferencia de impedancia

Para la mayoría de las aplicaciones, almacenar y recuperar información implica alguna forma de interacción con una base de datos relacional. El acceso desde un lenguaje de programación orientado a objeto a una base de datos relacional presenta un problema fundamental para los desarrolladores porque el diseño de datos relacionales y el orientado a objetos comparten estructuras de relaciones muy diferentes dentro de sus respectivos entornos, ya que las bases de datos relacionales están estructuradas en una configuración tabular (tablas, columnas, registros)

y el modelo orientado a objetos normalmente está relacionado en forma de árbol. A la disyuntiva planteada anteriormente se le denominará:"diferencia de impedancia" entre el modelo relacional y el orientado a objeto. (CEYUSA, 2006)

1.2.2. Transacciones de base de datos

Cada operación de acceso a datos, como crear, actualizar o borrar datos está asociada a una transacción de base de datos. Una transacción: es una unidad atómica de ejecución, es decir, es un grupo de instrucciones que se ejecutan con éxito en su totalidad, o no se ejecuta ninguna. La transacción una vez iniciada, puede terminar de dos formas: transacción aceptada, cuando las operaciones asociadas a una transacción se ejecutan con éxito y se hacen persistentes los cambios realizados, en la base de datos o transacción rechazada, cuando ocurre una intervención en la ejecución de las operaciones asociadas a una transacción y se "vuelve atrás" todos los cambios que puedan haberse realizado, imposibilitando de esta manera la inconsistencia en los datos de la base de datos. (MSDN, 2005)

1.3. Sistemas de base de datos

Los sistemas de base de datos constituyen una de las herramientas más ampliamente difundidas en la actual sociedad de la información, son utilizadas para la recuperación y almacenamiento de la información que manejan las empresas y diversos organismo ya sean dedicado a la cultura, educación o las ciencias en general. Surgen con el objetivo de resolver los problemas que planteaban los sistemas de ficheros, tales como la redundancia de datos y la dependencia entre programas y datos.

Una base de datos puede definirse como: "*un conjunto exhaustivo no redundante de datos estructurados organizados independientemente de su utilización y su implementación en máquina accesibles en tiempo real"* (HYDE, 2002). Para facilitar el trabajo con la base de datos existen los Sistemas de Gestión de Base de Datos (SGBD), estos sistemas representan: *"un tipo de software*

muy específico, dedicado a servir de interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan. Se compone de un lenguaje de definición de datos, de un lenguaje de manipulación de datos y de un lenguaje de consulta". (VALDÉS, 2007)

Es decir que un SGBD es un software que permite definir, crear y mantener las bases de datos, además que proporciona un acceso controlado a estas y un lenguaje de manejo de datos para la inserción, actualización, eliminación y consulta de información en estas bases de datos.

Los SGBD se dividen en tres generaciones. La primera generación de SGBD comprende a los sistemas jerárquicos y de red. El modelo relacional da surgimiento a la segunda generación, los sistemas de bases de datos relacionales, siendo este el más utilizado y extendido en la actualidad. La tercera generación de los SGBD se encuentra representada por el modelo relacional extendido y el modelo orientado a objetos.

Estos sistemas presentan una serie de ventajas. Algunas de estas ventajas son el control de la redundancia, la consistencia de datos, la mejora en los aspectos de seguridad y la integridad. Algunos de sus inconvenientes son su coste al requerir más capacidad de almacenamiento donde en la mayoría de los casos se dedica exclusivamente una computadora para el SGBD; al estar todo la información centralizada hace que presente vulnerabilidad ante los fallos que puedan producirse. (VALDÉS, 2007)

1.3.1. Sistema de base de datos relacional

Hasta el momento se han mencionado las diferentes generaciones o tipos de sistema de gestión de base de datos, cada una de estas generaciones trata de resolver los problemas que planteaban los sistemas de base de datos predecesores. El SGBD más extendido y de mayor uso en la actualidad es el Sistema de Gestión de Base de Datos Relacional (SGBDR), como su nombre lo indica este sistema se encarga de administrar las bases de datos relacionales. Es decir aquellas bases de datos que almacenan la información en forma de tablas, donde estas tablas se organizan en filas y columnas; donde cada fila representa un registro (conjuntos de

datos acerca de elementos separados) y las columnas definen los campos del registro (atributos particulares de un registro).

Los SGBDR permiten realizar búsquedas utilizando los datos de columnas especificadas de una tabla para encontrar datos adicionales en otra tabla, también proveen herramientas para evitar la duplicidad de registros, garantizan que al eliminar un registro sean eliminados todos los registros relacionados dependientes y además representan un modelo más comprensible y aplicable. Los SGBDR presentan deficiencia en cuanto al almacenamiento de datos gráficos, y puede dificultar el desarrollo de ciertas aplicaciones que manejan datos de tipos multimedia como audio y video.

Los Sistemas de Gestión de Base de Datos Orientadas a Objetos (SGBDOO) se propusieron con el objetivo de satisfacer las necesidades de las aplicaciones anteriormente mencionadas. Sin embargo, este sistema de gestión de base de datos no logra sustituir a los SGBDR, ya que la composición de este modelo, del orientado a objeto, no está del todo claro ya que aún no existe algún modelo de datos para los SGBDOO que esté aceptado mundialmente, los modelos de datos orientado a objetos no están totalmente desarrollados ya que carecen de una teórica matemática coherente que le sirva de base, algo que no le sucede a los SGBDR. También SGBDOO no disponen un alto nivel de desarrollo comercial y disponen de muy poca experiencia en su uso. Por tanto, actualmente los SGBDR son los más conocidos, extendidos y los que presentan mayor desarrollo comercial.

1.3.1.1. Lenguaje de Consulta Estructurado. SQL

El origen del Lenguaje de Consulta Estructurado (SQL, por sus siglas en inglés) está ligado al origen de la base de datos relacional. Representa un estándar para los lenguajes relacionales y todos los SGBDR hacen uso de este estándar, aunque cada SGBD desarrolla su propio SQL con pequeñas invariantes a partir del SQL estandarizado. El SQL se considera un lenguaje declarativo ya que las sentencias SQL describen qué información se quiere procesar pero no como llevarla a cabo, el cómo procesar la información que se quiere lo determina el SGBD. (MUKHAN, 2002)

SQL permite realizar consultas a la base de datos y también definir la estructura de una base de datos relacional. Para definir la estructura de la base de datos, SQL cuenta con un lenguaje de definición de datos conocido por las siglas en inglés: DDL, Data Description Language. El DDL establece comandos (Tabla 1.1) que permite definir, borrar o modificar las tablas de una base de datos relacional y las relaciones entre estas tablas.

Comandos DDL	
Comando	**Descripción**
CREATE	Utilizado para crear nuevas tablas, campos e índices.
DROP	Empleado para eliminar tablas e índices.
ALERT	Utilizado para modificar las tablas agregando campos o cambiando la definición de los campos.

Tabla 1.1: Comandos de definición de datos SQL.

Para realizar consultas a la bases de datos se utiliza el lenguaje de manipulación de datos conocido como DML (Data Manipulation Language, por sus siglas en inglés). Para consultar los datos de la base de datos el DML cuenta con ciertos comando (Tabla 1.2) que permiten recuperar los datos almacenados en la base de datos y actualizar la base de datos añadiendo nuevos datos, suprimiendo datos antiguos o modificando datos previamente almacenados.

Comandos DML	
Comando	**Descripción**
SELECT	Utilizado para consultar registros de la base de datos que satisfagan un criterio determinado.
INSERT	Utilizado para cargar lotes de datos en la base de datos en una única operación.
UPDATE	Utilizado para modificar los valores de los campos y registros especificados.

DELETE	Utilizado para eliminar registros de una tabla de una base de datos.

Tabla 1.2: Comandos de definición de datos SQL.

Como se ha visto para manejar datos de una base de datos relacional y definir su estructura es imprescindible el uso de SQL. Es válido mencionar que este estándar no representa un lenguaje de programación. Si un programador quiere que su aplicación obtenga datos de una tabla o que cree nuevas tablas y relacionarlas en una base de datos relacional, sólo tiene que incluir en su código instrucciones SQL, siempre que el lenguaje que utiliza soporte sentencias SQL.

1.4. Mecanismos, técnicas y patrones de acceso a datos

Respecto a los mecanismos de acceso a los datos, el nivel de evolución es notable en la actualidad. Antes del surgimiento de mecanismos para acceder a base de datos, el acceso a un servidor, requería el conocimiento puntual de librerías específicas de cada servidor, lo que originaba que una aplicación desarrollada con estas librerías, fuera inútil si se cambiaban los datos de servidor. Dado este problema surgen mecanismos para acceder a la base de datos; a continuación se muestran los mecanismo encontrados durante la investigación.

1.4.1. Mecanismos

ODBC. Conectividad abierta a base de datos

ODBC es la abreviatura en inglés que indica la conectividad abierta de base de datos. Es un estándar desarrollado por Microsoft que se utiliza para acceder a base de datos a través de consultas SQL, posibilita el acceso a cualquier dato desde cualquier aplicación, sin importar qué sistema de base de datos almacena la información. ODBC ofrece gran independencia entre las aplicaciones y las base de datos, ya que inserta un manejador de base de datos que traduce las consultas de datos enviadas por la aplicación en comandos que el sistema de base de datos entienda. Para utilizar este mecanismo la aplicación y el sistema de base de datos deben ser

compatibles con ODBC, es decir, que la aplicación sea capaz de producir comandos ODBC y que el sistema de base de datos sea capaz de responder ante ellos, esto se cumple en su totalidad ya que actualmente la mayoría de los sistemas de bases de datos soportan comandos ODBC.
Este es un mecanismo de gran utilidad ya que ofrece una interfaz de acceso universal que permite no tener que aprender a usar multiples interfaces de programación de aplicación. Sin embargo, ODBC tiene sus inconvenientes producto de que es una interfaz escrita en el lenguaje C y al no ser este un lenguaje portable le resta escalabilidad a la aplicación desarrollada con ODBC, haciéndola dependiente del sistema operativo. Y además, ODBC tiene el inconveniente de que se debe instalar manualmente en cada máquina donde se ejecute la aplicación que utilice comandos ODBC para acceder a la base de datos. (SKINNER, 2002)

ADO.Net

ADO.Net es un conjunto de componentes que brindan servicios de acceso a datos. Todos estos componentes forman parte de la biblioteca de clases básicas incluidas en el Framework .NET. Generalmente es utilizado para acceder y modificar la información guardada en un SGBDR, aunque también puede ser usado para acceder a fuentes de datos no relacionales, tales como XML, Excel y otros.
ADO.Net consiste en dos partes primarias:

- Un proveedor de datos (DataProvider), que es un conjunto de clases que proporcionan acceso a una fuente de datos específicos. Mediante un proveedor se pueden realizar conexiones con la fuente de datos, enviar consultas SQL y almacenar datos en un objeto DataSet. Cada proveedor presenta cuatros objetos básicos que son:
    - Connection, para conectar con una base de datos y administrar las transacciones en una base de datos.
    - Command, para emitir sentencias SQL a una base de datos.
    - DataAdapter, para insertar datos en un objeto DataSet y reconciliar datos de la base de datos.

  - DataReader, proporciona una forma de leer una secuencia de registros de datos sólo hacia delante desde un origen de datos. (SKINNER, 2002)

- Un DataSet, que representa en memoria una estructura análoga a una base de datos relacional, permite almacenar un conjunto de datos obtenidos mediante un objeto DataAdapter. Un DataSet es un objeto independiente que contiene un conjunto de registros de forma desconectada a la fuente de datos por lo que el DataSet no conoce en absoluto el origen y destino de los datos que contiene. En él no solo se pueden colocar datos provenientes de una base de datos relacional, permite manejar datos provenientes de un XML, código o información escrita explícitamente por un usuario. (SKINNER, 2002)

Una característica particular de ADO.Net es que se puede desarrollar el acceso a bases de datos relacionales manteniendo una conexión física permanente con la base de datos durante todo el proceso de consultas o actualizaciones sobre los dato, pero también permite desarrollar el acceso a bases de datos relacionales de manera desconectada debido a la existencia del DataSet, ya que en este objeto se copian los datos obtenidos de la base de datos y luego se pueden consultar y actualizar sin contar con una conexión abierta. Luego si se desea se puede establecer una conexión con la base de datos, mediante el objeto DataAdapter, para sincronizar los cambios efectuados por el DataSet y actualizar los datos en la base de datos. Es válido mencionar que no es obligatorio usar el objeto DataSet para insertar, actualizar o eliminar datos en una base de datos relacional, se puede ejecutar comandos SQL directamente en la base de datos para realizar inserciones, actualizaciones y eliminaciones.

JDBC. Conectividad de base de datos de Java

JDBC (Java Database Connectivity, por su siglas en inglés) es una Interfaz de Programación de Aplicación (API), que permite la ejecución de operaciones sobre bases de datos relacionales desde el lenguaje de programación Java, independientemente del sistema operativo donde se

ejecute la aplicación o de la base de datos a la cual se accede. Antes de que un programa Java pueda emplear el lenguaje SQL para acceder a los datos almacenados en una base de datos, es necesario establecer una conexión con la base de datos y enviar las sentencias SQL a ejecutar en el servidor de base de datos, todo esto se puede realizar en Java utilizando JDBC.

Esta API cuenta con una serie de interfaces escritas en Java que proporcionan un acceso estándar a bases de datos relacionales. Las interfaces existentes para trabajar con base de datos son:

- DriverManager, para cargar un manejador de base de datos.
- Connection, para establecer conexiones con la base de datos.
- Statement, para enviar consultas SQL al SGBDR.
- ResultSet, para almacenar y trabajar con el resultado de la consulta SQL enviada al SGBDR.

JDBC permite acceder a distintos sistemas de base de datos cambiando únicamente el manejador de base de datos proporcionado por el fabricante. Estos manejadores de base de datos implementan las interfaces de acceso a datos de JDBC permitiendo que los comandos JBDC sean entendibles por el sistema de base de datos que se utilice. (MUKHAN, 2002)

Según su arquitectura existen cuatro tipos de manejadores de base de datos para JDBC (Anexo A):

- Los de tipo 1: proporciona un puente entre el API JDBC y el API ODBC. Su objetivo es traducir los comandos JDBC a comandos ODBC para acceder a la base de datos. Usando este manejador no se necesita un manejador específico de cada base de datos de tipo ODBC, pero tiene como inconveniente que muchas plataformas no lo tienen implementado y resulta un proceso lento ya que se tiene que traducir las llamadas JDBC a ODBC (y viceversa) y acceder a la base de datos mediante comandos OBDC. (MUKHAN, 2002)

- Los de tipos 2: traducen los comandos JDBC a comandos que los manejadores de cada fabricante de base de datos proporciona. Es similar al de tipo 1 ya que este utiliza métodos nativos específicos de cada sistema de base de datos. (MUKHAN, 2002)
- Los de tipo 3: se utiliza un software intermedio que actúa como un servidor de acceso a datos que permite la conexión de múltiples clientes Java a los servidores de base de datos. Los comandos JDBC son enviados al software intermedio y este se encarga de traducir las llamadas JDBC en comandos entendibles para los manejadores de base de datos de tipo 2 que proporciona el gestor de base de datos que se utilice. (MUKHAN, 2002)
- Los de tipo 4: este tipo de driver representa una alternativa a los de tipo 2. Al estar desarrollado completamente en Java, presenta independencia de la plataforma que se utilice, las llamadas JDBC se envían directamente a la base de datos a través de un Socket. Al no tener que realizar alguna traducción intermedia, ni software intermedio que instalar son más fáciles de desplegar y realizan el proceso de consulta a la base de datos más rápido que los demás tipos de manejadores de base de datos. (MUKHAN, 2002)

Lenguaje SQLJ

SQLJ es un estándar para embeber sentencias SQL en programas Java. Al contrario de JDBC, SQLJ no es una API sino una extensión del lenguaje Java. Los programas escritos con SQLJ representan programas en Java con código SQL estático, ya que a diferencia de JDBC donde las consultas son almacenadas en cadenas de caracteres y enviadas a la base de datos; con SQLJ el código SQL se incluye directamente en el programa Java y este código es conocido en tiempo de compilación. Como principal objetivo de SQLJ es permitir una mayor integración entre el lenguaje Java y las bases de datos relacionales.

Los programas SQLJ son más fáciles de escribir y de mantener, además pueden resultar en ocasiones más cortos que los programas equivalentes escritos solamente en JBDC. SQLJ tiene como inconveniente que actualmente muchos Entornos de Desarrollo Integrados (IDE, por sus siglas en inglés) no proporcionan soporte para SQLJ, no existe soporte para la mayoría de frameworks de persistencia comunes, y la mayoría de los fabricantes de soluciones de base de datos tampoco ofrecen soporte para SQLJ; motivo por el cual el SQLJ no es el preferido para implementar el acceso a datos en Java. (MUKHAN, 2002)

1.4.2. Patrón de diseño DAO

Un patrón de diseño es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores, debe ser reusable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Los patrones de diseño pretenden evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente, proporcionando un catálogo de elementos reusables en el diseño de sistemas de software. Formaliza un lenguaje común entre diseñadores, estandarizando también el modo en que se realiza el diseño.

En la tarea de desarrollar el acceso a datos de un sistema informático existe un patrón de diseño que su utilización se ha hecho indispensable, este es el patrón de diseño DAO (Data Access Object, por sus siglas en inglés) es uno de los patrones de diseño de mayor uso en el desarrollo del acceso a datos de un sistema informático.

Se utiliza para separar las operaciones de bajo nivel de acceso a datos, de las operaciones de alto nivel de la lógica del negocio. Una implementación típica de DAO contiene:

- Una interfaz DAO. Que expone operaciones de acceso a datos, estas operaciones no están vinculadas con alguna tecnología de persistencia. Esta característica permite la

abstracción del mecanismo que se emplee para acceder a los datos, permitiendo desarrollar diversas implementaciones a una misma interfaz DAO.

- Una clase concreta que implemente la interfaz DAO. Esta clase contiene la lógica de acceso a datos a una fuente específica de información.
- Una clase fabricadora. Responsable de instanciar las interfaces DAO.
- Objetos de transferencias de datos. Posibilitan transferir la información que se obtiene desde una fuente determinada.

Este patrón permite implementar las operaciones definidas por una interfaz DAO de diversas maneras utilizando diferentes tecnologías, sin producir cambios en el código que utiliza las operaciones expuestas por esta interfaz, ya que cada implementación concreta de una interfaz DAO es transparente a quien utilice dicha interfaz. (LAGO, 2007)

1.4.3. Técnica Mapeo Objeto-Relacional

El mapeo objeto-relacional, conocido por su nombre en inglés: Object-Relational Mapping (ORM), es una técnica de acceso a datos que trata de disminuir la diferencia de impedancia entre el modelo relacional y el modelo orientado a objetos. Esta técnica consiste en transformar las representaciones de los datos de un modelo de objetos en un modelo de datos con un esquema relacional y viceversa, donde esta transformación se realiza transparentemente al programador. Una solución de este tipo está compuesta de cuatro partes:

- Una API encargada de realizar las operaciones de creación, recuperación, actualización y eliminación (operaciones CRUD) para los objetos de las clases que se corresponden con las tablas de la base de datos. Dichas operaciones se corresponden con instrucciones de lenguajes de manipulación de datos en el Sistema Gestor de Base Datos (SGBD). (QUINTERO, 2008)
- Un lenguaje de consulta a la base de datos, generalmente orientado a objetos, o API para especificar consultas que se refieran a las clases y atributos de estas clases. Estos

lenguajes en su mayoría son similares al SQL, con la diferencia principalmente de que es orientado a objeto como se ha comentado anteriormente. (QUINTERO, 2008)

- Una utilidad para especificar metadatos de mapeo. Generalmente mediantes archivos XML se realizan la correspondencia entre las tablas de la base de datos y las clases que representan estas tablas. También se hacen corresponder los atributos de las clases con los campos de la tabla. (QUINTERO, 2008)
- Técnicas de optimización de transacciones como el chequeo sucio automático, que permite detectar cambios en los objetos y actualizar su estado en la base de datos sin necesidad de escribir alguna sentencia de actualización; técnicas estratégicas para la recuperación de las colecciones que presentan los objetos (producto de las asociaciones uno-muchos, muchos-uno), tales como:
    - Carga perezosa (Lazy Loading), donde se cargan primero el objeto de la tabla maestra y posteriormente, si se necesitan, sus colecciones de objeto que representan los datos de la tabla detalle.
    - Carga proactiva (Eager Loading). Se cargan de una vez el objeto de la tabla maestra y sus colecciones de objetos de la tabla detalle. (QUINTERO, 2008)

Con el uso de la técnica mapeo objeto-relacional el programador de acceso a datos no tiene que convertir explícitamente los datos resultantes de una consulta a la base de datos en objetos de transferencias. El acceso a datos mediante el uso de esta técnica se realiza completamente orientado a objetos, en cierta medida se ignora la existencia del SGBDR.

1.5. Herramientas que utilizan el mapeo objeto-relacional

Anteriormente se describió una técnica que permite eliminar la diferencia de impedancia entre el modelo relacional, que es el modelo propio de los SGBDR, y el modelo orientado a objeto. En la práctica esta técnica permite crear una base de datos orientada a objetos virtual sobre la base de datos relacional. Esto posibilita el uso de las características propias de la programación orientada

a objetos, básicamente herencia y polimorfismo. En la actualidad hay herramientas disponibles de uso libre que implementan el mapeo objeto-relacional. Entre las más utilizadas se encuentran:

1.5.1. IBatis

IBatis es un framework de código libre basado en capas desarrollado por Apache, que se ocupa de la capa de persistencia, se sitúa entre la capa de negocio y la capa de la base de datos. Simplifica la implementación del patrón de diseño DAO y la persistencia de objetos en bases de datos relacionales.

Los principales componentes propuesto por IBatis son:

- Data Mapper, que proporciona una forma sencilla de interacción de datos entre los objetos y bases de datos relacionales.
- DAO, abstracción que oculta la persistencia de objetos en la aplicación y proporciona un API de acceso a datos al resto de la aplicación.

IBatis no es una herramienta ORM en su totalidad, es decir, no cumple con todas las características que debe tener una solución de mapeo objeto-relacional. No cuenta con un lenguaje propio de consulta, utiliza el SQL. Este framework en esencia se basa en asociar objetos con sentencias SQL o procedimientos almacenados mediante ficheros XML. (PÉREZ, 2009)

IBatis presenta grandes ventajas en cuanto a su simplicidad. Es relativamente fácil de utilizar y de dominarlo, utiliza programación declarativa (uso de ficheros XML) por lo que separa las sentencias SQL del código de la aplicación. IBatis se puede utilizar en aplicaciones desarrolladas en el lenguaje Java o en la plataforma .NET. Además este framework es muy tolerante si se utiliza en modelos de datos poco normalizados.

También este framework presenta una serie de inconvenientes que hacen que en ocasiones no sea la herramienta más idónea. Una de sus principales desventajas es que no es una solución totalmente transparente, ya que el programador de acceso a datos tiene que escribir sentencias SQL explícitamente en archivos XML.

1.5.2. Hibernate

Hibernate es una herramienta ORM completa para la plataforma Java, actualmente es una de las herramientas ORM de mayor reputación en la comunidad de desarrollo, representa claramente el producto de código libre líder en soluciones de mapeo objeto-relacional. Este framework parte de la filosofía de corresponder clases simples, que contengan solos los métodos “get” y “set” (clases POJO), con las tablas de la base de datos mediante archivos XML, por lo que proporciona un modelo de programación natural permitiendo diseñar objetos persistentes que podrán incluir polimorfismo, relaciones, colecciones.

Las cinco principales interfaces que se usan en casi todas las aplicaciones donde se utiliza Hibernate son:

- **Session**: es la principal interfaz que se utiliza para implementar la gestión de acceso a datos con Hibernate. Una instancia de esta interfaz se puede equiparar a grandes rasgos al concepto de conexión de JDBC y cumple un papel muy parecido, es decir, sirve para realizar una o varias operaciones de acceso a datos relacionadas dentro de un proceso de negocio, desmarcar una transacción y aporta algunos servicios adicionales como manejo de caché para evitar interacciones innecesarias con la base de datos. (BAUER, 2005)
- **SessionFactory**: como su nombre en inglés lo indica, es una fábrica de sesión. Contrario a la interfaz Session, una instancia de SessionFactory es costosa de crear y normalmente existe una sola instancia para toda la aplicación, creada generalmente en el arranque de la aplicación. Una SessionFactory representa a una base de datos, por lo que debe existir una SessionFactory por cada base de datos a la que se acceda. (BAUER, 2005)
- **Configuration**: usada para configurar Hibernate. En una instancia de Configuration se especifican las propiedades necesarias para conectarse a una base de datos, tales como la ubicación de la base de datos, usuario y contraseña, dialecto del SGBDR entre otras

propiedades y además la ubicación de los archivos de mapeo. Mediante esta interfaz se crea una SessionFactory. (BAUER, 2005)

- **Transaction**: es una interfaz de uso opcional. Puede que muchas aplicaciones no hagan uso de esta interfaz y definan las transacciones de base de datos por otros medios. El uso de esta interfaz para manejar las transacciones de base de datos ayuda a mantener portable las aplicaciones desarrolladas con Hibernate entre los distintos entornos de ejecución, ya sea un entorno gestionado o no gestionado. (BAUER, 2005)
- **Query y Criteria**: Query permite realizar consultas a la base de datos y controlar la forma en que la consulta sea ejecutada. Las consultas son escritas en HQL (Lenguaje de Consulta de Hibernate) o en el SQL nativo del SGBDR. La interfaz Criteria es similar a la Interface Query, con la diferencia que las consultas a la base de datos que se realizan no son mediante un lenguaje de consulta como HQL o SQL, sino mediante mecanismo orientados a objetos. (BAUER, 2005)

Hibernate presenta grandes ventajas en cuanto a sus prestaciones y flexibilidad al realizar la correspondencia entre tablas de la base de datos y las clases que representan estas tablas. Hibernate soporta diversos tipos de asociaciones, se pueden establecer asociaciones de uno a muchos, de uno a uno, de muchos a muchos y hasta asociaciones de herencia. Unos de los atractivos al utilizar Hibernate es que cuenta con herramientas que permiten generar automáticamente los archivos de mapeo y las clases persistente a partir de un esquema de base de datos existente, eliminando el proceso de crearlos manualmente por parte del programador. Este framework presenta su propio lenguaje de consulta, el HQL. Las consultas construidas con este lenguaje se realizan en base a las clases y sus atributos, con el uso de este lenguaje se establece gran portabilidad hacia los distintos SGBDR, ya que a partir de este lenguaje el propio Hibernate genera el código SQL nativo del SGBDR que se utilice.

1.6. Lenguajes, metodología y herramientas

1.6.1. Programación orientada a aspectos

La Programación Orientada a Aspectos (AOP, por sus siglas en inglés) es un modelo de programación que aborda un problema específico, ya que con este modelo se pueden capturar las partes de un sistema que los modelos de programación habituales como el orientado a objeto obligan a que estén repartidos a lo largo de distintos módulos del sistema. Estos fragmentos que afectan a distintas partes de un sistema informático son llamados aspectos y los problemas que solucionan, problemas cruzados.

Usando un lenguaje que soporte AOP, se puede encapsular estas dependencias o conceptos en módulos individuales, obteniendo un sistema independiente de ellos y podemos utilizarlos o no sin tocar el código del sistema básico, preservando la integridad de las operaciones básicas.

Dentro de los principales campo de acción de la programación orientada a aspectos están: la seguridad, medidas de tiempo y optimización, manejo de errores y la persistencia. Actualmente varios lenguajes se han creado o adaptado con nuevos componente para desarrollar utilizando este nuevo paradigma, alguno de ellos son: Aspect.Net, AspectC++, HyperJ, AspectJ. Este último será el lenguaje que se utilizará para el desarrollo del componente. (MORENO, 2006)

1.6.1.1. Lenguaje AspectJ

Representa una extensión orientada a aspectos del lenguaje de programación Java, es el más maduro y con mayor número de características de todos los frameworks AOP para Java. Permite la aplicación de aspectos a clases Java para la solución de los problemas cruzados. Al utilizar Java como lenguaje base, AspectJ proporciona todos los beneficios de Java y hace que sea sencillo que los desarrolladores Java entiendan el lenguaje AspectJ.

AspecJ permite aumentar o reemplazar el flujo de ejecución principal del programa de una manera que afecta a los distintos módulos, modificando el comportamiento del sistema. Por ejemplo, se puede especificar que una acción determinada sea ejecutada antes de la ejecución

de ciertos métodos o manejadores de excepciones en un conjunto de clases tan sólo especificando en un módulo separado los puntos de entrelazado y la acción a realizar cuando se alcanzan esos puntos. (MORENO, 2006)

También permite realizar modificaciones en la estructura estática de las clases, interfaces y aspectos del sistema. Se puede añadir nuevos atributos y métodos a clases e interfaces para definir estados comportamientos específicos. (MORENO, 2006)

1.6.2. Lenguaje Java

Este es un lenguaje basado en las tecnologías de software libre, es un lenguaje orientado a objetos, se basa mucho en los lenguajes de programación como C y C++, pero presenta un modelo de objetos mucho más simple que estos últimos.

Debido a su característica de ser orientado a objeto soporta las tres características propias del paradigma de la orientación a objetos como son el encapsulamiento, herencia y polimorfismo.

También ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos, como son la aritmética de punteros y la herencia múltiple que se permitían en lenguajes como C y C++. Los programas en Java pueden ejecutarse sobre cualquier sistema operativo, sin necesidad de hacer cambios en estos programas, ya que el compilador de Java genera bytecodes, que no es más que un formato intermedio indiferente al sistema operativo diseñado para transportar el código eficientemente a múltiples plataformas hardware y software, el resto de los problemas los soluciona el intérprete de Java, manteniéndose para todas las plataformas la misma sintaxis y las mismas bibliotecas de clases estándares del lenguaje Java. (MOREA, 2005)

1.6.3. Lenguaje de modelado UML

UML es un lenguaje para visualizar, especificar, construir y documentar los elementos que componen un sistema desarrollado con la tecnología orientada a objetos. Se ha convertido en el estándar internacional para definir, organizar y visualizar los elementos que configuran la

arquitectura de una aplicación orientada a objetos. Este lenguaje pretende unificar las experiencias acumuladas sobre técnicas de modelado e incorporar las mejores prácticas en un acercamiento estándar.

UML permite la creación de los diferentes modelos que ofrecen las vistas necesarias para la construcción de un software de calidad y permite la compresión del sistema que se quiere realizar tanto por parte de los usuarios finales, como de los desarrolladores que implementarán la solución.

UML como lenguaje para modelar un sistema de software se ha convertido en un estándar con las siguientes características:

- Permite modelar sistemas utilizando técnicas orientadas a objetos (OO).
- Permite especificar las decisiones de análisis y diseño, construyéndose modelos precisos y completos.
- Está compuesto por diversos elementos gráficos que se combinan para conformar diagramas, además cuenta con reglas para combinar dichos elementos.
- Es independiente del lenguaje de programación y de las características de los proyectos, ya que fue diseñado para modelar cualquier tipo de proyecto.
- Integra las mejores prácticas de los lenguajes de modelación existentes.
- A pesar de ser un lenguaje potente, es fácil de aprender y de usar.
- Permite documentar los artefactos de un proceso de desarrollo.
- Capaz de modelar toda la gama de sistemas que se necesite construir.

1.6.4. Metodología de Desarrollo Basado en Rasgos. FDD

En la actualidad la cantidad y variedad de los procesos de desarrollo de software han aumentado en gran medida. Se han desarrollado dos corrientes en lo referente a los procesos de desarrollo, están los métodos pesados y los métodos ligeros o ágiles. Las metodologías tradicionales,

pesadas generalmente, se centran especialmente en el control del proceso, mediante una exhaustiva documentación; definiendo roles, actividades, artefactos, herramientas y notaciones para el modelado y una documentación detallada. Estas metodologías son muy efectivas y necesarias en proyectos grandes.

Las metodologías ágiles, dan mayor valor al individuo, a la colaboración con el cliente y al desarrollo incremental del software con iteraciones muy cortas. Con cortos documentos centrados en lo esencial.

La metodología de Desarrollo Basado en Rasgos, FDD por sus siglas en inglés: Feature Driven Development. Es un enfoque ágil para el desarrollo de sistemas, dicho enfoque no hace énfasis en la obtención de los requerimientos sino en cómo se realizan las fases de diseño y construcción. Está dirigido para proyectos con tiempo de desarrollo cortos, es decir, menos de un año, e iteraciones cortas, aproximadamente dos semanas, que a medida que se itera se va generando software funcional, estableciendo de esta forma entregas tangibles del sistema que se construye, estas entregas representan pequeñas partes del software con significado para el cliente. (CALABRIA, 2003)

Un proyecto desarrollado por esta metodología se divide en 5 procesos de desarrollo (Figura 1). Los tres primeros procesos se hacen al principio del proyecto, y los dos últimos se realizan de forma iterativa.

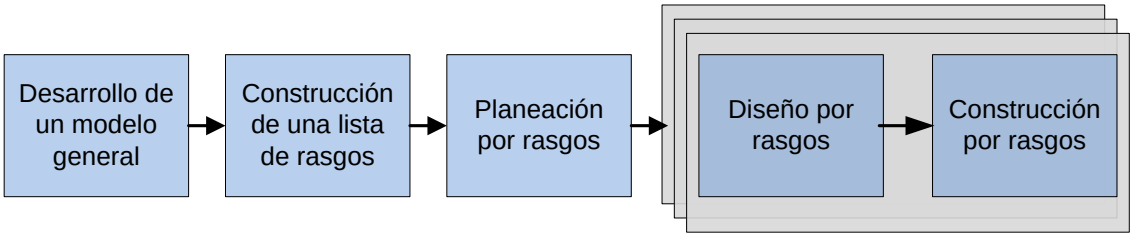


Figura 1: Procesos de desarrollo de la metodología FDD.

- **Desarrollo de un modelo general.**

  Al iniciar esta fase, se tiene una idea del contexto y visión del sistema. Se presenta un ensayo del dominio (walkthrough) en el cual los miembros del equipo son informados a través de una descripción del sistema que se quiere construir. El dominio global es dividido en diferentes áreas y se realiza un ensayo del dominio detallado para cada una de las áreas del dominio. A partir de estas descripciones del contexto y visión del sistema se realizan un modelo de objetos para cada área de dominio y simultáneamente, se construye un modelo global del sistema a partir de los modelos por áreas. (CALABRIA, 2003)

- **Construcción de una lista de rasgos.**

  Se identifican los rasgos que resumen las características y comportamiento del sistema que se desea construir. El resultado de esta fase es una lista de rasgos categorizada jerárquicamente. La lista de rasgos se compone por áreas temáticas, actividades del negocio que comprenden estas áreas temáticas y por los rasgos que representan los pasos para cumplimentar cada actividad del negocio. Estos rasgos son pequeños funcionalidades útiles a los ojos del cliente y los rasgos que requieran de más de diez días se descomponen en otros más pequeños que se puedan cumplimentar en el tiempo máximo de dos semanas. (CALABRIA, 2003)

- **Planeación por rasgos.**

  Se incluye la creación de un plan de alto nivel, en el que los conjuntos de rasgos se ponen en secuencia conforme a su prioridad y dependencia. Esta planificación permite establecer que rasgos se incluyen en cada iteración de los dos últimos proceso que establece esta metodología. (CALABRIA, 2003)

- **Diseño por rasgos y construcción por rasgos.**

  Este proceso se realiza de forma iterativa. En cada iteración se selecciona un pequeño conjunto de rasgos del conjunto. Se identifican las clases involucradas por cada rasgo y se

diseñan e implementan estas clases a partir del conjunto de rasgos que están en la iteración. Se procede luego iterativamente hasta que se producen todos los rasgos identificados. Una iteración puede tomar de unos pocos días a un máximo de dos semanas. El proceso iterativo está definido por los hitos: un ensayo del dominio del rasgo a desarrollar, diseño de las clases, inspección del diseño, codificación, pruebas unitarias, inspección de código y por último promoción del rasgo que se construyó. (CALABRIA, 2003)

1.6.5. Visual Paradigm para UML

Visual Paradigm para UML es una herramienta que utiliza UML como lenguaje de modelado, es una herramienta libre de costo que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. Permite dibujar todos los tipos de diagramas de clases, generar código desde diagramas y generar documentación. Es una herramienta colaborativa, es decir, soporta múltiples usuarios trabajando sobre el mismo proyecto; genera la documentación automáticamente en varios formatos como Web o .Pdf, y permite control de versiones.

Con el uso de esta herramienta la productividad en el desarrollo de software aumenta ya que se reduce el costo de las mismas en términos de tiempo y de dinero, ayudando en todos los aspectos del ciclo de vida de desarrollo del componente en tareas como el proceso de realizar el diseño de este, documentación o detección de errores.

1.6.6. Eclipse Europa 3.3

Es un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés) de código abierto y multiplataforma. Presenta como característica clave su extensibilidad, producto que tiene una arquitectura de plug-ins que permite integrar diversos lenguajes e introducir otras aplicaciones accesorias. Un plug-in es la mínima unidad de la plataforma que puede ser desarrollado por

separado y que le aporta una nueva funcionalidad al IDE. Este IDE presenta otras características como:

- Tiene un editor visual con sintaxis coloreada.
- Realiza compilación incremental de código.
- Modifica e inspecciona valores de variables.
- Avisa de los errores cometidos mediante una ventana secundaria.
- Depura código que resida en una máquina remota.

Es soportado por los principales sistemas operativos: Linux, Windows, Solaris 8, Mac OSX-Mac/Carbon. Permite mantener el registro de todo el trabajo y los cambios en los ficheros que forman un proyecto, dado que presenta un sistema de control de versiones, de los principales están: CVS y Subeclipse. Con esta característica, diferentes desarrolladores pueden colaborar en la implementación de un mismo proyecto. Además presenta funciones útiles de programación como:

- Información de Javadoc, al posicionar el puntero sobre el nombre de una clase.
- Buen auto-completamiento de código en el editor Java.
- Detención de errores en tiempo de compilación.
- Conjunto de plantillas de código.
- Formateo del código

Con el uso de este IDE, aumenta la productividad y la calidad del software. También reduce el coste de las aplicaciones informáticas al ser libre de pago por su uso. (MONTERO, 2007)

1.6.7. JUnit

JUnit es un Framework de código libre desarrollado para crear y ejecutar pruebas unitarias de forma automática en Java. Con esta herramienta se determinan casos de pruebas, estos casos de pruebas son clases que disponen de métodos para probar el funcionamiento de una clase o módulo específico, por lo que para cada clase que se quiera probar se define su correspondiente clase de caso de prueba. También se pueden definir suite de pruebas que permiten organizar los

casos de pruebas de forma que cada suite agrupa los casos de pruebas que estén funcionalmente relacionados. De las herramientas para realizar pruebas unitarias JUnit es una de la más utilizada en Java, producto que no tiene coste alguno, la mayoría de los IDE soportan la utilización de la herramienta, el resultado de las pruebas realizadas con JUnit se obtiene inmediatamente y se pueden presentar junto con el código para validar el trabajo realizado. (USAOLA, 2006)

1.7. Conclusiones

En este capítulo se realizó un estudio sobre las tendencias actuales para almacenar la información, los mecanismos para acceder a las bases de datos y las soluciones existentes que implementan la técnica de mapeo objeto-relacional. Se pudo identificar que las bases de datos relacionales son las más utilizadas actualmente y que para acceder a estas desde los lenguaje de programación tradicionales existen diferentes mecanismo, que a pesar de sus diferencias todos tienen como objetivo principal implementar y abstraer los protocolos de comunicación entre las aplicaciones informáticas y las bases de datos, simplificando de cierta forma la tarea de consultarlas. También se describió la técnica de mapeo objeto-relacional como una solución a la diferencia de impedancia entre el modelo relacional y el orientado a objeto y se presentaron dos soluciones que implementan esta técnica, considerándose una vez analizadas, que el framework Hibernate representa una solución ORM muy completa, de mucha madures y de gran utilidad en toda la comunidad de desarrollo.

En este capítulo se presentaron además los lenguajes, la metodología y herramientas que se utilizaron para el desarrollo del componente. Se utiliza la metodología de Desarrollo Basado en Rasgos debido a que es más factible realizar entregas de código funcional en cortos periodos de tiempo, además de que esta metodología expone reglas para alcanzar el éxito en el desarrollo del componente generando los artefactos necesarios para la construcción y documentación del componente. Como lenguaje de modelado se utiliza UML y como herramienta Case, Visual Paradigm por las ventajas que proporciona esta herramienta, ya que de las herramientas CASE libres de costo representa la más madura y de mayor prestaciones para el modelado visual. El lenguaje utilizado es Java y su extensión para programar orientado a aspectos, AspectJ. La herramienta que se consideró más adecuada para la implementación es Eclipse Europa 3.3 por la rapidez de compilado y su fácil extensibilidad en cuento a añadir nuevas características y funcionalidades mediante plug-ings. Para realizar las pruebas unitarias al componente se utiliza el framework JUnit.

CAPÍTULO II. CARACTERÍSTICAS DEL COMPONENTE

2.1 Introducción

En este capítulo teniendo en cuenta las estrategias que se pueden establecer a la hora de definir como implementar el acceso a datos en una aplicación informática, se determina la principal característica del componente, que es utilizar la técnica de mapeo objeto-relacional. También se identifican qué dificultades están presentes al utilizar el framework Hibernate; y a partir de estas dificultades, y el objetivo que se persigue con el desarrollo del componente de acceso a datos, se construye una lista de rasgos, que representará las características del componente. Una vez creada la lista de rasgos se planifica cada rasgo para la posterior fase iterativa de diseño y construcción por rasgos.

2.2 Estrategias de acceso a datos

Establecer la manera de realizar el acceso a datos en una aplicación informática es una de las decisiones de arquitectura más importantes ya que en una aplicación estándar un gran por ciento del código generado está relacionado con el código para guardar y recuperar la información inherente a su dominio. Una estrategia inmediata sería utilizar directamente alguna API de acceso a datos, ya sea ADO.Net para el desarrollo en la plataforma .Net o JDBC para el desarrollo en la plataforma Java. Bajo esta estrategia la mayor parte del código se centra en recuperar datos de una base de datos y eventualmente modificarlos para posteriormente aplicar los cambios contra la base de datos relacional. Pero esta estrategia desde el punto de vista orientado a objetos tienes sus desventajas:

- La información que se maneja es de forma tabular, no representa objetos del dominio de la aplicación.

- Representa las relaciones entre tablas de la base de datos y no los distintos tipos de asociaciones que existen entre los objetos del dominio.
- Es extremadamente sensible a los cambios que puedan surgir en el esquema de la base de datos.
- El código generado para la gestión del acceso a datos tiende a ser engorroso y relativamente difícil de mantener.

Además se puede añadir que estas APIs de acceso a datos representan una capa liviana sobre los sistemas de bases de datos relacionales, y como tal se preocupan solamente de abstraer e implementar los mecanismos y protocolos de comunicación; y todo el resto de los aspectos relacionados con la interacción con las bases de datos debe ser manejado por los desarrolladores.

Una estrategia recomendada y compatible con las buenas práctica de diseño es crear un modelo de objeto del dominio que represente el cien por ciento de la información que maneja la aplicación y utilizar la técnica de mapeo objeto-relacional para guardar y recuperar estos objetos contra la base de datos. Esta estrategia ofrece las siguientes ventajas:

- Ya no se maneja información tabular sino puros objetos. La implementación del acceso a datos se realiza con un enfoque orientado a objeto. Donde se puede establecer herencia, se puede mapear una jerarquía de clases a una sola tabla o crear una tabla por cada clase concreta. También soporta el mapeo de todo tipo de relaciones que pueden existir entre los objetos de dominio, como asociaciones de uno a mucho, uno a uno, muchos a muchos.
- Se disminuye las veces que se accede a la base de datos, ya que se guardan en memoria los objetos que son accedidos varias veces.
- Soporta múltiples dialectos SQL, se puede independizar por completo la aplicación del SGBDR que se utilice. La aplicación puede guardar sus datos en SQL Server, MySQL,

PostgreSQL, y demás gestores con solo cambiar la configuración correspondiente para cada uno de ellos.

- Se aprovechan todas las capacidades que brinda la técnica de mapeo objeto-relacional. En cuanto a simplificación del acceso a base de datos relacionales.

Teniendo en cuenta estas dos posibles estrategias, se utilizará como mecanismo de acceso a datos en la construcción del componente, la técnica de mapeo objeto-relacional, y en este caso se optará por alguna solución de mapeo objeto-relacional existente. Entre las herramientas ORM analizadas durante la investigación, el framework Hibernate resultó el más idóneo, por sus prestaciones, estabilidad, utilidad a nivel mundial y su flexibilidad para realizar el mapeo entre clases y tablas de la base de datos.

2.3 Construcción del modelo general

Esta sección es una transcripción de una sesión de modelaje en el mes de abril del 2009. Describe el primer proceso que establece la metodología utilizada. Se muestra el ensayo de dominio que describe el contexto y la visión de lo que se desea desarrollar y se presenta el diagrama de objetos del sistema.

2.3.1 Ensayo del dominio

Para el desarrollo del Sistema de Emisión de Documentos de Identificación (SEDI), se necesita un mecanismo que ayude a minimizar el tiempo y el esfuerzo dedicado a la tarea de implementar el acceso a datos, para lograrlo se quiere desarrollar un componente de acceso a datos. Para la construcción del SEDI se utilizará el lenguaje de programación Java y para el almacenamiento de los datos un SGBDR.

Para la implementación del acceso a datos del SEDI el desarrollador debe definir un conjunto de operaciones para cada interfaz DAO que utilice este sistema. Existen ciertas operaciones de

acceso a datos que son imprescindibles y constantes en cada interfaz DAO por lo que se hace necesario que el componente de acceso a datos brinde e implemente de forma genérica todas aquellas operaciones que se consideren comunes o básicas.

El desarrollador de acceso a datos además de definir un conjunto de operaciones básicas para cada interfaz DAO, puede dotar a alguna Interfaz DAO de otras funciones de acceso a datos que abarquen aspectos de búsquedas más especializados. Estas búsquedas requieren el uso de algún tipo de lenguaje de consulta de acceso a datos para lograr el nivel de detalle que se necesite, por lo que el componente debe contener funciones que permitan realizar búsquedas utilizando el lenguaje de consultas HQL.

En Hibernate el manejo del sistema de transacciones suele ser engorroso y repetitivo de forma que hay que introducir muchas instrucciones que se repiten una y otra vez, también el programador tiene que preocuparse en cuando abrir y cerrar una sesión de Hibernate provocando que este no se concentre en la lógica de acceso a datos e imposibilitando de esta forma que le dedique el tiempo necesario para optimizar las consultas creadas, por lo que se desea que todas estas dificultades que se presentan a la hora de utilizar el framework Hibernate sean resueltas por el componente a desarrollar.

2.3.2 Modelo general

El modelo general consiste en la construcción de un diagrama de clases que representa los tipos de objetos más importantes dentro del dominio del problema y las relaciones entre ellos. Este diagrama de clases es de carácter estructural y luce como el tradicional diagrama entidad-relación de las bases de datos relacionales, pero presenta dos grandes diferencias ya que puede incluir relaciones de herencia, generalización y especialización, y las operaciones no reflejan conveniencias de programación sino que se describen en formas de rasgos especificando cómo debe comportarse el objeto. (PALMER, 2002)

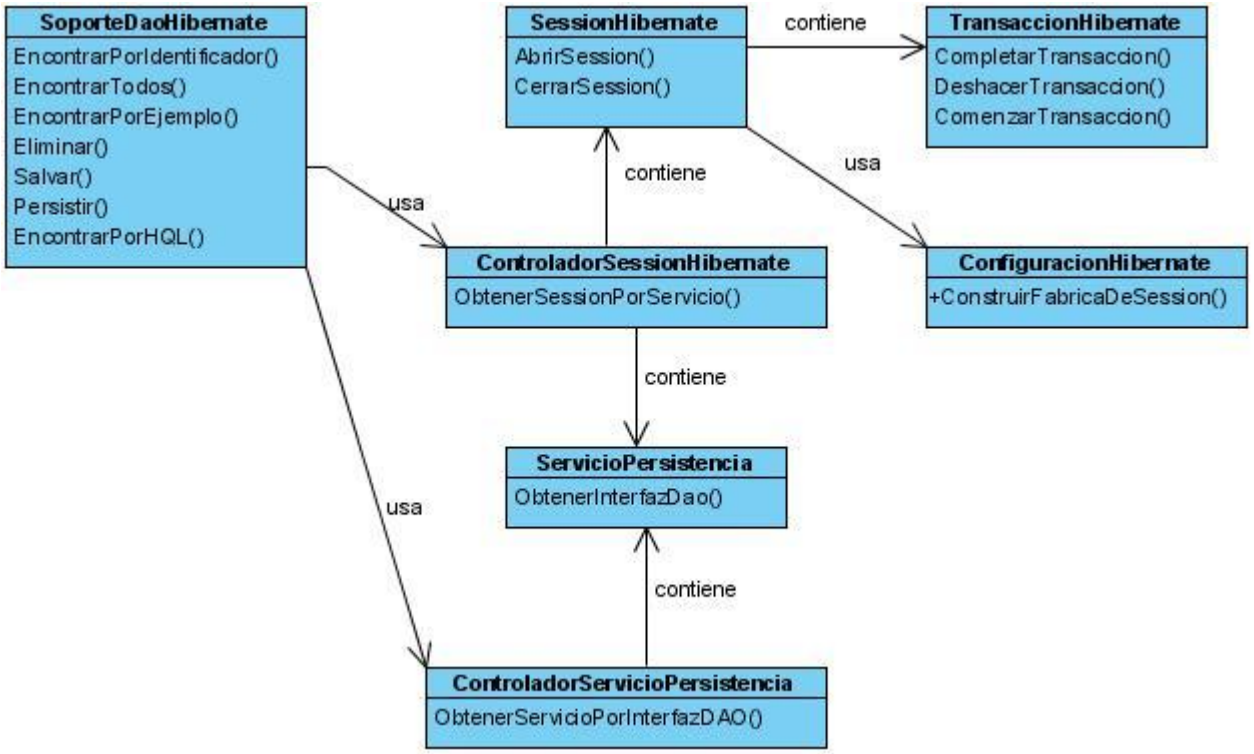


Figura 2: Modelo general del componente de acceso a datos.

2.4 Construcción de la lista de rasgos

A partir del ensayo del dominio y el modelo general se pudieron identificar un conjunto de rasgos que representan las características del componente de acceso a datos y que define la base para lo que se debe diseñar e implementar. La lista de rasgos se estructuró de forma jerárquica, donde se definen áreas temáticas (en este caso una sola), las actividades del negocio comprendidas en cada área temática y los rasgos que permiten cumplimentar la actividad de negocio a la que pertenecen. En la Tabla 3 se muestran los rasgos identificados con sus correspondientes actividades del negocio dentro del área temática determinada.

Lista de rasgos		
Área Temática	**Actividad del negocio**	**Rasgos**
Acceso a datos.	Establecer operaciones de soporte.	• Encontrar un objeto á partir de su identificador en la base de datos. • Encontrar todos los objetos de una tabla. • Encontrar objetos a partir de una consulta por ejemplo. • Eliminar un objeto de la base de datos. • Salvar un objeto a la base de datos. • Actualizar un objeto de la base de datos. • Buscar objetos a partir de consultas HQL. • Determinar interfaz de operaciones comunes para el acceso a datos.
Acceso a datos.	Gestionar las sesiones de hibernate.	• Abrir sesión de hibernate. • Cerrar sesión de hibernate. • Obtener sesión hibernate ligada a un servicio de persistencia. • Construir fábrica de sesión de hibernate.
Acceso a datos.	Gestionar las transacciones de hibernate.	• Comenzar transacción de hibernate. • Completar transacción de hibérnate. • Deshacer transacción de hibernate. • Separar codificación de transacciones de hibernate dentro

		de las operaciones de acceso a datos.
Acceso a datos.	Establecer un servicio de persistencia.	• Obtener instancia de interfaz Dao. • Obtener el servicio de persistencia que instanció una interfaz Dao.

Tabla 2.3: Lista de rasgos del componente de acceso a datos.

2.5 Planeación de la lista de rasgos

El último proceso que se realizó antes de empezar a diseñar e implementar el componente fue el de planear las lista de rasgos identificada. Esta planeación se realizó teniendo en cuenta todos los rasgos identificados en el proceso anterior y basándose en los hitos que se establecen durante los procesos iterativos de diseño y construcción por rasgos. A la hora de planear cada rasgo se tuvo en cuenta que dentro de una iteración no pueden existir rasgos que dependan de otros rasgos que no han sido implementados, con la excepción de que puede depender de rasgos no implementado, pero que todos estos se encuentre en la misma iteración. Otro aspecto que se tuvo en cuenta y que definió que se realizara una sola iteración para diseñar e implementar el componente de acceso a datos fue el de elegir los rasgos de tal forma que la implementación de estos formen una pieza funcional completa. A continuación se muestran en forma de tablas la planeación de cada rasgo, la fecha de planeación mostrada representa la culminación de cada hito establecido en la iteración.

Área temática: Acceso a datos.

Establecer operaciones de soporte.						**Rasgos**: 8
Descripción	**Ensayo**	**Diseño**	**Ins.Diseño**	**Código**	**Ins.Código**	**Prom.**
	Plan	Plan	Plan	Plan	Plan	Plan

Encontrar un objeto á partir de su identificador en la base de datos.	05/5/09	05/7/09	04/7/09	04/7/09	04/8/09	04/8/09
Encontrar todos los objetos de una tabla.	05/5/09	05/7/09	04/7/09	04/7/09	04/8/09	04/8/09
Encontrar objetos a partir de una consulta por ejemplo.	05/5/09	05/7/09	04/7/09	04/7/09	04/8/09	04/8/09
Eliminar un objeto de la base de datos.	05/5/09	05/7/09	04/7/09	04/7/09	04/8/09	04/8/09
Salvar un objeto a la base de datos.	05/5/09	05/7/09	04/7/09	04/7/09	04/8/09	04/8/09
Persistir un objeto de la base de datos.	05/5/09	05/7/09	04/7/09	04/7/09	04/8/09	04/8/09
Buscar objetos a partir de consultas HQL.	05/5/09	05/7/09	04/7/09	04/8/09	04/9/09	04/9/09
Determinar interfaz de operaciones comunes en el acceso a datos.	05/4/09	05/5/09	05/5/09	05/5/09	05/5/09	05/5/09

Tabla 2.4: Planeación de los rasgos para establecer operaciones de soporte.

Gestionar las sesiones de hibernate.					**Rasgos**: 4	
Descripción	**Ensayo**	**Diseño**	**Ins.Diseño**	**Código**	**Ins.Código**	**Prom.**
	Plan	Plan	Plan	Plan	Plan	Plan
Abrir sesión de hibernate.	04/27/09	04/28/09	04/28/09	04/29/09	04/29/09	04/30/09
Cerrar sesión de hibernate.	04/27/09	04/28/09	04/28/09	04/29/09	04/29/09	04/30/09
Obtener sesión hibernate ligada a un servicio de persistencia.	04/30/09	05/2/09	05/2/09	05/4/09	05/4/09	05/4/09
Construir fábrica de sesión de hibernate.	04/27/09	04/28/09	04/28/09	04/29/09	04/29/09	04/30/09

Tabla 2.5: Planeación de los rasgos para gestionar las sesiones de hibernate.

Gestionar las transacciones de hibernate.					**Rasgos**: 4	
Descripción	**Ensayo**	**Diseño**	**Ins.Diseño**	**Código**	**Ins.Código**	**Prom.**
	Plan	Plan	Plan	Plan	Plan	Plan
Comenzar transacción de hibernate.	04/27/09	04/28/09	04/28/09	04/29/09	04/29/09	04/30/09
Completar transacción de hibernate.	04/27/09	04/28/09	04/28/09	04/29/09	04/29/09	04/30/09
Deshacer transacción de hibernate.	04/27/09	04/28/09	04/28/09	04/29/09	04/29/09	04/30/09
Separar manejo	05/9/09	05/10/09	05/10/09	05/11/09	05/11/09	05/12/09

de transacciones de hibernate dentro de las operaciones de acceso a datos.						

Tabla 2.6: Planeación de los rasgos para gestionar las transacciones de hibernate.

Establecer servicio de persistencia.					**Rasgos**: 2	
Descripción	**Ensayo**	**Diseño**	**Ins.Diseño**	**Código**	**Ins.Código**	**Prom.**
	Plan	Plan	Plan	Plan	Plan	Plan
Obtener instancia de interfaz de acceso a datos.	04/30/09	05/2/09	05/2/09	05/4/09	05/4/09	05/4/09
Obtener el servicio de persistencia que instanció una interfaz Dao.	04/30/09	05/2/09	05/2/09	05/4/09	05/4/09	05/4/09

Tabla 2.7: Planeación de los rasgos para establecer un servicio de persistencia.

2.6 Conclusiones

En este capítulo se presentó la realización de los tres primeros procesos secuenciales para el desarrollo del componente usando la metodología de desarrollo basado en rasgos. Con la ejecución de estos procesos se logró determinar las principales clases del sistema a construir y mediante la construcción de una lista de rasgos se determinó que funcionalidades debe cumplir el componente para lograr el objetivo que se persigue con el desarrollo de este. También se definió que la construcción completa del componente se realizará en una sola iteración teniendo en cuenta que el resultado de una iteración es la presentación de una pieza funcional completa.

CAPÍTULO III. DISEÑO Y CONSTRUCCIÓN DEL COMPONENTE

3.1 Introducción

En el presente capítulo se presentará una visión de la estructura de alto nivel que tendrá el componente y su interacción con elementos externos. También se mostrará que patrón de diseño fue aplicado para el desarrollo del componente así como la utilización del paradigma de programación orientado a aspectos aplicado al componente, utilizando el lenguaje AspectJ. Se presentará el diagrama de clases del diseño del componente, así como los diagramas de secuencia para aquellos rasgos de menor entendimiento a la hora de realizar la implementación de estos. También se presentará el diagrama de despliegue del componente de acceso a datos.

3.2 Descripción de la arquitectura del componente

La arquitectura de software es un aspecto muy importante a tener en cuenta para el desarrollo de cualquier producto informático, pues esta define la estructura o forma del sistema en general, y en ella están comprendidas las propiedades visibles de los componentes del software y las relaciones entre ellos.

Este componente está basado en un modelo de dos capas donde que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior. La capa inferior del componente se encarga de gestionar el acceso a la base de datos, esta capa estará compuesta por el framework Hibernate. La capa superior representará una capa de abstracción para el programador de persistencia, que permitirá acceder a la base de datos utilizando los mecanismos de Hibernate de forma sencilla y transparente. Esta capa superior expone métodos de conveniencias para que el programador consulte la base de datos y además permitirá que el programador no tenga que manejar las conexiones a la base de

datos (sesiones de Hibernate), ni que tenga que escribir código dedicado a las transacciones de base de datos. (Figura 3).

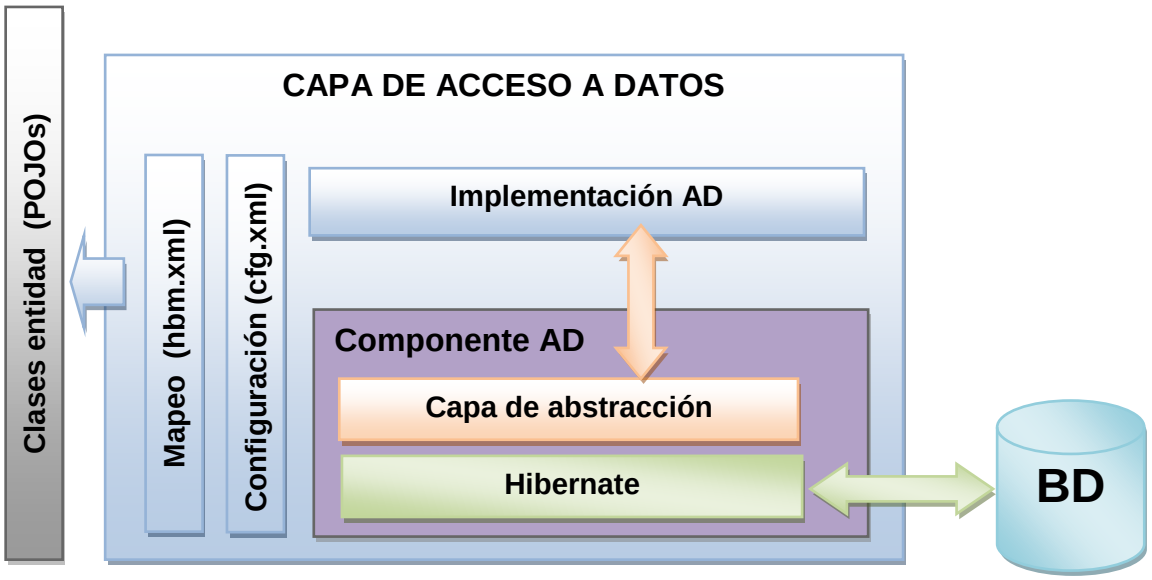


Figura 3: Arquitectura del componente y sus interacciones con elementos externos.

En el diagrama de bloques anterior se muestra gráficamente la arquitectura y los elementos que interactúan con el componente dentro de una capa de acceso a datos.

Las clases entidad (POJOs) son compartidas por toda la aplicación, estas clases son el modelo de objeto que representa la información que maneja la aplicación, esencialmente son contenedores de datos. Se mantienen los archivos de mapeo y configuración de hibernate de igual manera que si se estuviera utilizando solamente este framework, el componente no elimina la forma tradicional de Hibernate para realizar el mapeo de las clases entidad contra la base de datos, aprovechando que existen herramientas que generan estos archivos automáticamente a partir de un esquema de base de datos existentes.

3.3 Patrón de diseño instancia única

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Sólo es utilizado un patrón de diseño en esta solución, el patrón instancia única (Singleton), diseñado para restringir la instanciación de objetos a partir de una clase a un único objeto, garantizando que sólo tenga una instancia y proporcionando un punto de acceso global a ella. El patrón Singleton es utilizado específicamente en este componente para proveer una única instancia de la clase que maneja la configuración de Hibernate para crear una fábrica de sesiones (Figura 4). Esto garantiza que el objeto accedido, la fábrica de sesión, es siempre el mismo sin importar el punto o el momento de acceso, evitando además que este deba ser pasado constantemente como parámetro a clases que lo necesiten. La implementación de este patrón se hace agregando en la clase un método estático encargado de crear una instancia de la misma sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada de otra forma que no sea el uso de este método, se regula el alcance del constructor.

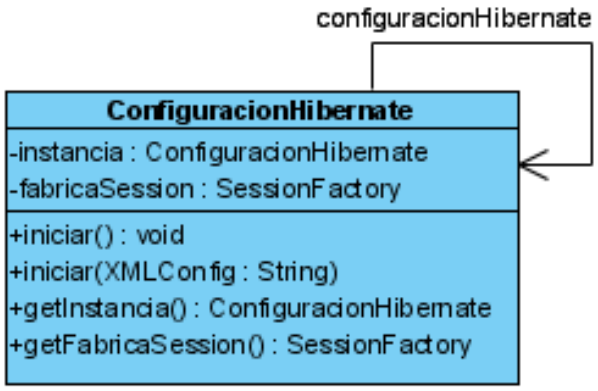


Figura 4: Aplicación del patrón Singleton a la clase ConfiguracionHibernate.

3.4 Aplicación de aspectos

Dentro de los lenguajes orientados a aspectos, se utilizó AspectJ que representa una extensión del lenguaje Java para programar orientado a aspectos. AspectJ permite adicionar comportamiento en ciertos puntos bien definidos durante la ejecución de los programas (modificación dinámica) y también permite definir nuevas propiedades y operaciones sobre clases existentes (modificación estática).

Mediante este lenguaje se definió las diferentes partes del código que son afectadas por el concepto de transacción de base de datos encapsulado por el aspecto AspectoTransaccionHibernate. Estas partes del código afectadas por la codificación repetitiva y engorrosa de las transacciones de base de datos son las propias operaciones de acceso a datos que expone una interfaz Dao. Para lograr encapsular el concepto de transacción de base de datos en un aspecto se definió que cuando se alcance cierto punto de ejecución, en este caso la llamada de una operación de acceso a datos desde cualquier parte de la aplicación, el aspecto se encargue de entrelazar alrededor de la operación de acceso a datos el código dedicado a las transacciones de base de datos.

Para entrelazar el aspecto con el código de las operaciones de acceso a datos se utilizó los siguientes constructores que especifica AspectJ:

Puntos de enlace (JoinPoint), que no es más que un sitio identificable en la ejecución de un programa, que podría ser la instanciación de una clase, la llamada a un método o la asignación de una valor a un atributo de una clase, entre otras más.

Puntos de corte (PoinCut), en este constructor se especifican los JoinPoint y se capturan la información referente al contexto, o sea se pueden capturar el objeto que llamó a un método (en el caso de que el JoinPoint determine la llamada a tal método), y también capturar los argumentos pasados al método.

Otros de los constructores que permitió escribir el código de las transacciones de base de datos que se entrelazaría con el código de las operaciones de acceso a datos fue:

Avisos (Advice), que permite escribir código que se quiere entrelazar en los JoinPoint que se especifican en un PoinCut. Un Advice se puede ejecutar tanto antes (Before) de la ejecución de un método (si el PointCut que utiliza el Advice determina la ejecución de tal método), después (After) de la ejecución de un método o alrededor (Around) de la ejecución de un método.

Teniendo en cuenta estos constructores y definiendo las partes de código afectadas por el código disperso de las transacciones de base de datos se creó el aspecto, AspectoTransaccionHibernate.

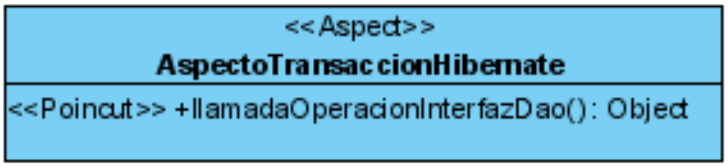


Figura 5: Aspecto que encapsula el concepto de transacción de base de datos.

En este aspecto de definió el **PoinCut *llamadaOperacionInterfazDao()*** para capturar las llamadas desde cualquier parte del código a la operaciones de las interfaces de acceso a datos que se implementen utilizando el componente de acceso a datos. Además se utiliza un **Advice** que contiene el PoinCut mostrado anteriormente. Este Advice es de tipo **Around** y dentro de su cuerpo encapsula la lógica para la transacción de base de datos que debe mezclar con la operación de acceso a datos que se capture.

3.5 Modelo de clases del diseño

3.5.1. Diagrama de clases del diseño

Un diagrama de clases del diseño es un diagrama que muestra un conjunto de interfaces, colaboraciones y sus relaciones. Los diagramas de clases del diseño se utilizan para modelar

principalmente la vista de diseño estática de un sistema. Esto incluye modelar el vocabulario del sistema, las colaboraciones o esquemas. Los diagramas de clases, son importantes, no sólo para visualizar, especificar y documentar modelos estructurales, sino también para construir sistemas ejecutables, aplicando ingeniería directa e inversa. A continuación se muestra el diagrama de clases del diseño del componente de acceso a datos.

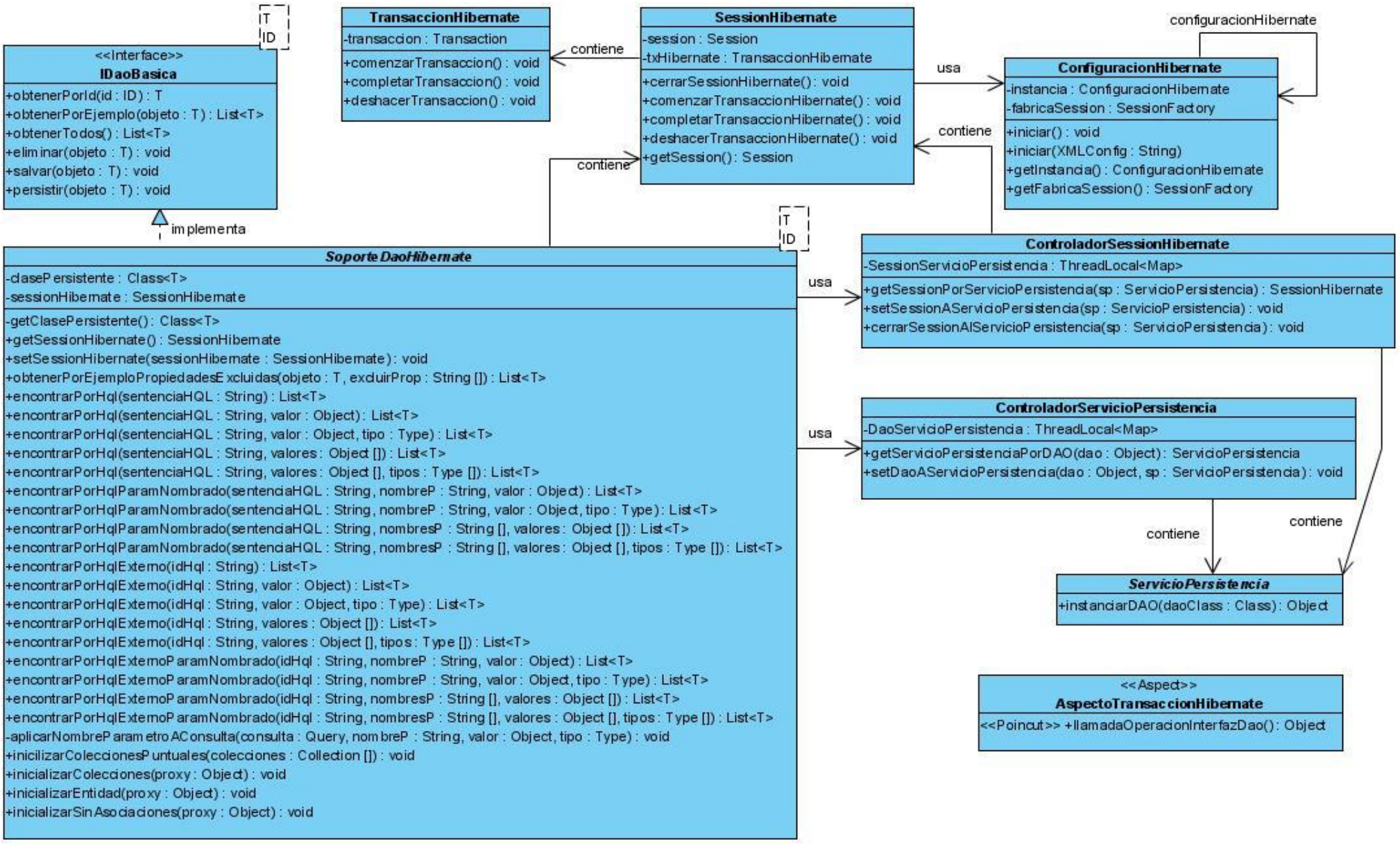


Figura 6: Diagrama de clases del diseño del componente de acceso a datos.

3.5.2. Descripción de las clases del diseño

En esta sección se describen todas las clases del diseño presentadas anteriormente en el diagrama de clase del diseño, para cada una de ellas se muestra su nombre y tipo, una breve explicación de su responsabilidad o función, y el nombre y tipo de cada uno de sus atributos. Se expone además efímeramente el objetivo y forma de funcionamiento de todas las operaciones contenidas por la clase, exceptuando aquellas comúnmente conocidas como “get” y “set”.

IDaoBasica

Esta interfaz define un conjunto de operaciones de acceso a datos comunes, que debe tener cada interfaz creada por el desarrollador de acceso a datos para la persistencia del SEDI. Es una interfaz parametrizada que tiene dos argumentos:

- T, es la clase persistente para la que se implementará el acceso a datos.
- ID, es el tipo del identificador de la clase persistente.

Los métodos estarán definidos en base a esos parámetros y no están acoplados a ninguna tecnología de persistencia.

Nombre: IDaoBasica	
Interfaz	
Atributo	**Tipo**
Para cada responsabilidad:	
Nombre:	obtenerPorId(ID identificador)
Descripción:	Devuelve un objeto de la base de datos que tenga el identificador pasado por parámetro.
Nombre:	obtenerPorEjemplo(T objetoEjemplo)
Descripción:	Devuelve los objetos que coincidan con los valores de los atributos del objeto pasado por parámetro. No se incluyen las propiedades con valor nulo, las que

	especifican el identificador en la base de datos y las que establecen asociaciones.
Nombre:	obtenerTodos()
Descripción:	Devuelve todos los objetos de una tabla de la base de datos.
Nombre:	salvar(T objeto)
Descripción:	Salva un objeto nuevo en la base de datos
Nombre:	eliminar(T objeto)
Descripción:	Elimina de la base de datos el objeto pasado por parámetro.
Nombre:	persistir(T objeto)
Descripción:	Actualiza el objeto pasado por parámetro si está asociado con la base de datos o salva el objeto si se creó por primera vez.

Tabla 3.8: Descripción de la interfaz IDaoBasica.

SoporteDaoHibernate

Es una clase de soporte para la implementación del acceso a datos. Contiene métodos de conveniencias para su uso por parte del programador de acceso a datos. Esta clase implementa las operaciones definidas por interfaz **IDaoBasica**, además de contener otros métodos de conveniencias para consultar la base de datos utilizando el lenguaje HQL.

SoporteDaoHibernate es una clase abstracta y parametrizada, con 2 argumentos:

- T, es la clase persistente para la que se implementará el acceso a datos.
- ID, define el tipo del identificador de la clase persistente.

Los métodos estarán definidos en base a esos argumentos y se implementan usando los mecanismos de acceso a datos de Hibernate.

Nombre: SoporteDaoHibernate	
Soporte	
Atributo	**Tipo**
1. clasePersistente 2. sessionHibernate	1. Class<T> 2. SessionHibernate
Para cada responsabilidad:	
Nombre:	obtenerPorEjemploPropiedadesExcluidas(T objeto, String[] excluirProp)
Descripción:	Devuelve los objetos que coincidan con los valores de los atributos del objeto pasado por parámetro. No se incluyen en la búsqueda las propiedades con valor nulo, las que representan el identificador, las que establecen asociaciones ni las especificadas en el parámetro *excluirProp*.
Nombre:	encontrarPorHql(String sentenciaHql)
Descripción:	Devuelve objetos a partir de una consulta HQL estática (no espera valores de entrada) pasada por parámetro.
Nombre:	encontrarPorHql(String sentenciaHql, Object valor)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica (espera valores de entrada) pasada por parámetro.
Nombre:	encontrarPorHql(String sentenciaHql, Object valor, Type tipo)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica (con valor de entrada y tipo de valor) pasada por parámetro.
Nombre:	encontrarPorHql(String sentenciaHql, Object[] valores)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica (con más de un valor de entrada) pasada por parámetro.
Nombre:	encontrarPorHql(String sentenciaHql, Object[] valores, Type[] tipos)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica (con valores de entrada y tipos de valores) pasada por parámetro.
Nombre:	encontrarPorHqlParamNombrado(String sentenciaHql, String nombreP, Object valor)
Descripción:	Devuelve objetos a partir de una consulta HQL con valores de parámetros con

	nombres. Una consulta HQL con valores de parámetros nombrados se define de la siguiente forma: *From Clase c Where c.atributo =* ***:valorNombrado***
Nombre:	encontrarPorHqlParamNombrado(String sentenciaHql, String nombreP, Object valor, Type tipo)
Descripción:	Devuelve objetos a partir de una consulta HQL con valores de parámetros con nombres (espera también el tipo de dato del valor nombrado).
Nombre:	encontrarPorHqlParamNombrado(String sentenciaHql, String[] nombresP, Object[] valores)
Descripción:	Devuelve objetos a partir de una consulta HQL con valores de parámetros con nombres (espera varios valores).
Nombre:	encontrarPorHqlParamNombrado(String sentenciaHql, String[] nombresP, Object[] valores, Type[] tipos)
Descripción:	Devuelve objetos a partir de una consulta HQL con valores de parámetros con nombres (espera varios valores y sus tipos de datos correspondientes).
Nombre:	encontrarPorHqlExterno(String idHql)
Descripción:	Devuelve objetos a partir de una consulta HQL estática escrita en un archivo XML. El parámetro IdHql representa el identificador de la consulta en el archivo XML.
Nombre:	encontrarPorHqlExterno(String idHql, Object valor)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica (espera un valor) escrita en un archivo XML.
Nombre:	encontrarPorHqlExterno(String idHql, Object valor, Type tipo)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica (espera un valor y su tipo de datos) escrita en un archivo XML.
Nombre:	encontrarPorHqlExterno(String idHql, Object[] valores)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica (espera varios valores para su ejecución) escrita en un archivo XML.
Nombre:	encontrarPorHqlExterno(String idHql, Object[] valores, Type[] tipos)

Descripción:	Devuelve objetos a partir de una consulta HQL dinámica (espera varios valores y sus tipos de datos correspondientes para su ejecución) escrita en un archivo XML.
Nombre:	encontrarPorHqlExternoParamNombrado(String idHql, String nombreP, Object valor)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica escrita en un archivo XML con valores de parámetros con nombres (espera un valor). El parámetro *IdHql* representa el identificador de la consulta en el archivo XML.
Nombre:	encontrarPorHqlExternoParamNombrado(String idHql, String nombreP, Object valor, Type tipo)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica escrita en un archivo XML con valores de parámetros con nombres (espera un valor y tipo de valor).
Nombre:	encontrarPorHqlexternoParamNombrado(String idHql, String[] nombresP, Object[] valores)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica escrita en un archivo XML con valores de parámetros con nombres (espera valores).
Nombre:	encontrarPorHqlexternoParamNombrado(String idHql, String nombreP[], Object[] valores, Type[] tipos)
Descripción:	Devuelve objetos a partir de una consulta HQL dinámica escrita en un archivo XML con valores de parámetros con nombres (espera valores y sus tipos correspondientes).
Nombre:	getSessionHibernate()
Descripción:	Obtiene un objeto de la clase SessionHibernate.
Nombre:	getClasePersistente()
Descripción:	Devuelve la clase persistente a la que se le implementa el acceso a datos.
Nombre:	aplicarNombreParametroAConsulta(Query consultaObjeto, String nombreP, Object valor, Type tipo)
Descripción:	Método de ayuda para formar una consulta HQL dinámica que espera valores con parámetros nombrados.
Nombre:	inicializarColeccionesPuntuales(Collection[] colecciones)

Descripción:	Inicializa las colecciones pasadas por parámetro que no han sido cargadas por una sesión de hibernate.
Nombre:	inicializarColecciones(Object proxy)
Descripción:	Inicializa todas las colecciones que presentan el objeto pasado por parámetro.
Nombre:	inicializarEntidades(Object proxy)
Descripción:	Inicializa todos los objetos que representan una asociación de uno con otros objetos que tiene el objeto pasado por parámetro.
Nombre:	inicializarAsociacciones(Object proxy)
Descripción:	Inicializa todas asociaciones del objeto pasado por parámetro. Ya sean asociaciones de muchos o uno con otros objetos.
Nombre:	inicializarSinAsociacciones(Object proxy)
Descripción:	Inicializa un objeto proxy. Solo inicializa sus atributos atómicos, los atributos que establecen asociaciones no son inicializados.

Tabla 3.9: Descripción de la clase SoporteDaoHibernate.

SessionHibernate

Nombre: SessionHibernate	
Atributo	**Tipo**
1. session 2. txHibernate	1. Session 2. TransaccionHibernate
Para cada responsabilidad:	
Nombre:	cerrarSessionHibernate()
Descripción:	Cierra una sesión de la interfaz de Hibernate: Session
Nombre:	comenzarTransaccionHibernate()
Descripción:	Llama al método comenzarTransaccion () de la clase TransaccionHibernate
Nombre:	completarTransaccionHibernate()

Descripción:	Llama al método completarTransaccion () de la clase TransaccionHibernate
Nombre:	deshacerTransaccionHibernate()
Descripción:	Llama al método deshacerTransaccion () de la clase TransaccionHibernate
Nombre:	getSession()
Descripción:	Obtiene la sesión de la interfaz de Hibernate: Session

Tabla 3.10 : Descripción de la clase SessionHibernate.

TransaccionHibernate

Nombre: TransaccionHibernate	
Atributo	**Tipo**
1. transaccion	1. Transaction
Para cada responsabilidad:	
Nombre:	comenzarTransaccion ()
Descripción:	Comienza una transacción de base de datos mediante la interfaz de Hibernate: Transaction
Nombre:	completarTransaccion ()
Descripción:	Realiza el “commit” de una transacción de base de datos mediante la interfaz de Hibernate: Transaction
Nombre:	deshacerTransaccion ()
Descripción:	Realiza el “rollback” de una transacción de base de datos mediante la interfaz de Hibernate: Transaction
Nombre:	setTransaccion(Transaction transaccion)
Descripción:	

Tabla 3.11 : Descripción de la clase TransaccionHibernate.

ConfiguracionHibernate

Esta clase implementa el patrón de diseño instancia única, conocido por su nombre en inglés como Singleton. Permite crear una fábrica de sesión solo la primera vez que se inicia la configuración de Hibernate.

Nombre: ConfiguracionHibernate	
Atributo	**Tipo**
1. Instancia 2. fabricaSession	1. ConfiguracionHibernate 2. SessionFactory
Para cada responsabilidad:	
Nombre:	iniciar(String xmlConfiguracion)
Descripción:	Crea una fábrica de sesión. Se pasa por parámetro la ubicación del archivo de configuración de hibernate.
Nombre:	Iniciar()
Descripción:	Crea una fábrica de sesión. El archivo de configuración de hibernate debe estar en la raíz del sistema de paquete de la aplicación.
Nombre:	getFabricaSession()
Descripción:	
Nombre:	getInstancia()
Descripción:	Devuelve la instancia de ConfiguraciónHibernate creada por primera vez.

Tabla 3.12: Descripción de la clase ConfiguracionHibernate.

ControladorSessionHibernate

Nombre: ControladorSessionHibernate	
Controladora	
Atributo	**Tipo**

1. SessionServicioPersistencia	1. ThreadLocal<Map>
Para cada responsabilidad:	
Nombre:	getSessionPorServicioPersistencia(ServicioPersistencia sp)
Descripción:	Busca la instancia de la clase SessionHibernate ligada a la instancia del servicio de persistencia pasado por parámetro
Nombre:	setSessionAServicioPersistencia(ServicioPersistencia sp)
Descripción:	Liga una instancia de la clase SessionHibernate al servicio de persistencia pasado por parámetro.
Nombre:	cerrarSessionAlServicioPersistencia(ServicioPersistencia sp)
Descripción:	Elimina la instancia SessionHibernate que esté ligada al servicio de persistencia pasado por parámetro.

Tabla 3.6: Descripción de la clase ControladorSessionHibernate.

ServicioPersistencia

Nombre: ServicioPersistencia	
Atributo	**Tipo**
Para cada responsabilidad:	
Nombre:	instanciarDAO(Class daoClass)
Descripción:	Devuelve una instancia de la clase pasada por parámetro

Tabla 3.7: Descripción de la clase ServicioPersistencia.

ControladorServicioPersistencia

Nombre: ControladorServicioPersistencia
controladora

Atributo	**Tipo**
1. DaoServicioPersistencia	1. ThreadLocal<Map>
Para cada responsabilidad:	
Nombre:	getServicioPersistenciaPorDAO(Object dao)
Descripción:	Obtiene el servicio de persistencia que instanció la interfaz dao pasado por parámetro.
Nombre:	setDaoAServicioPersistencia(Object dao, ServicioPersistencia sp)
Descripción:	Liga la interfaz dao pasado por parámetro con el servicio de persistencia pasado por parámetro.

Tabla 3.8: Descripción de la clase ControladorServicioPersistencia.

AspectoTransaccionHibernate

Nombre: AspectoTransaccionHibernate	
aspecto	
PointCut	**Descripción**
1. llamadaOperacionInterfazDao()	1. Intercepta las llamadas a las operaciones de cualquier interfaz de acceso a datos creadas por el programador (estas interfaces deben extender de la interfaz IDaoBasica que proporciona el componente)
Advice	**Descripción**
1. around() : llamadaOperacionInterfazDao()	1. Adiciona código para el manejo de transacciones antes y después de la ejecución de la operación que interceptó el pointcut *llamadaOperacionInterfazDao()*

Tabla 3.9: Descripción del aspecto AspectoTransaccionHibernate.

3.6 Diagramas de secuencias

Los diagramas de secuencia se utilizan para estructurar los aspectos dinámicos de un sistema, conllevan a modelar instancias concretas o prototípicas de clases interfaces, componentes y nodos, así como los mensajes enviados entre ellos. Ilustran un comportamiento, describen la forma en que grupos de objetos colaboran para obtener un objetivo final y presenta una visión externa del sistema. Para mejor entendimiento de los aspectos dinámicos de aquellos rasgos que pudieran no estar claros a la hora de su implementación se crearon los siguientes diagramas de secuencias.

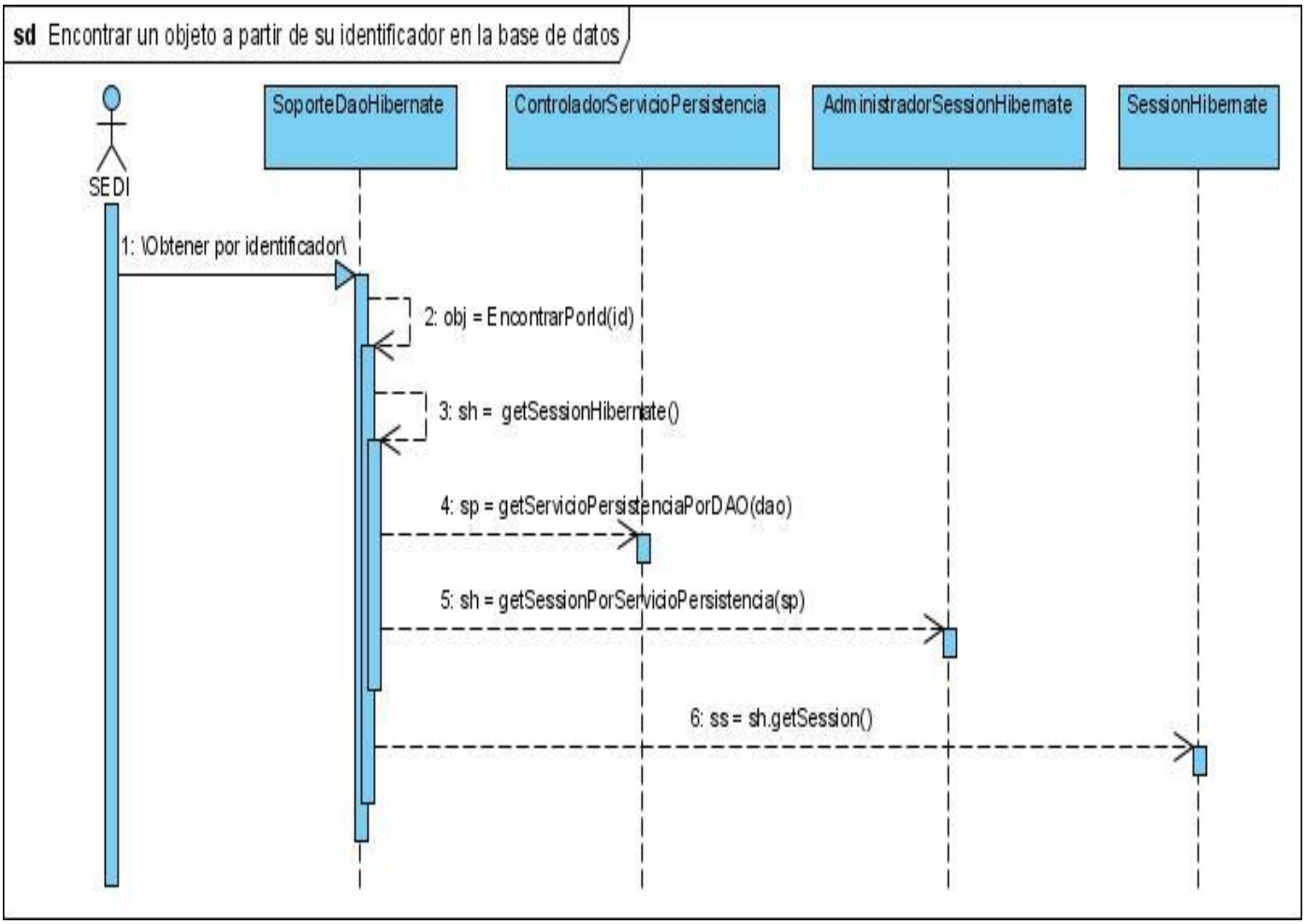


Figura 7: Diagrama de secuencia: Encontrar un objeto a partir de su identificador en la base de datos.

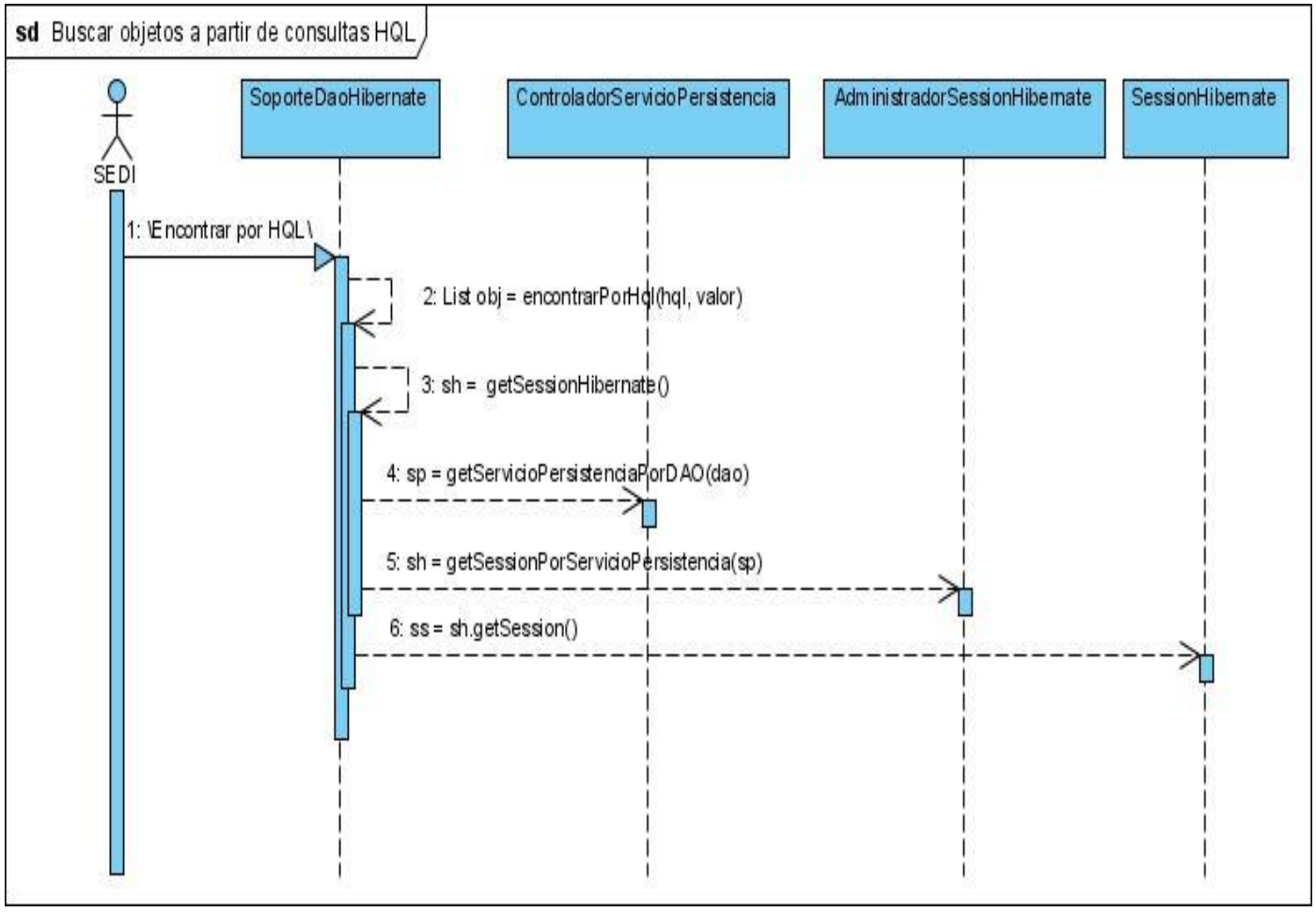


Figura 8: Diagrama de secuencia: Buscar objetos a partir de consultas HQL 1.

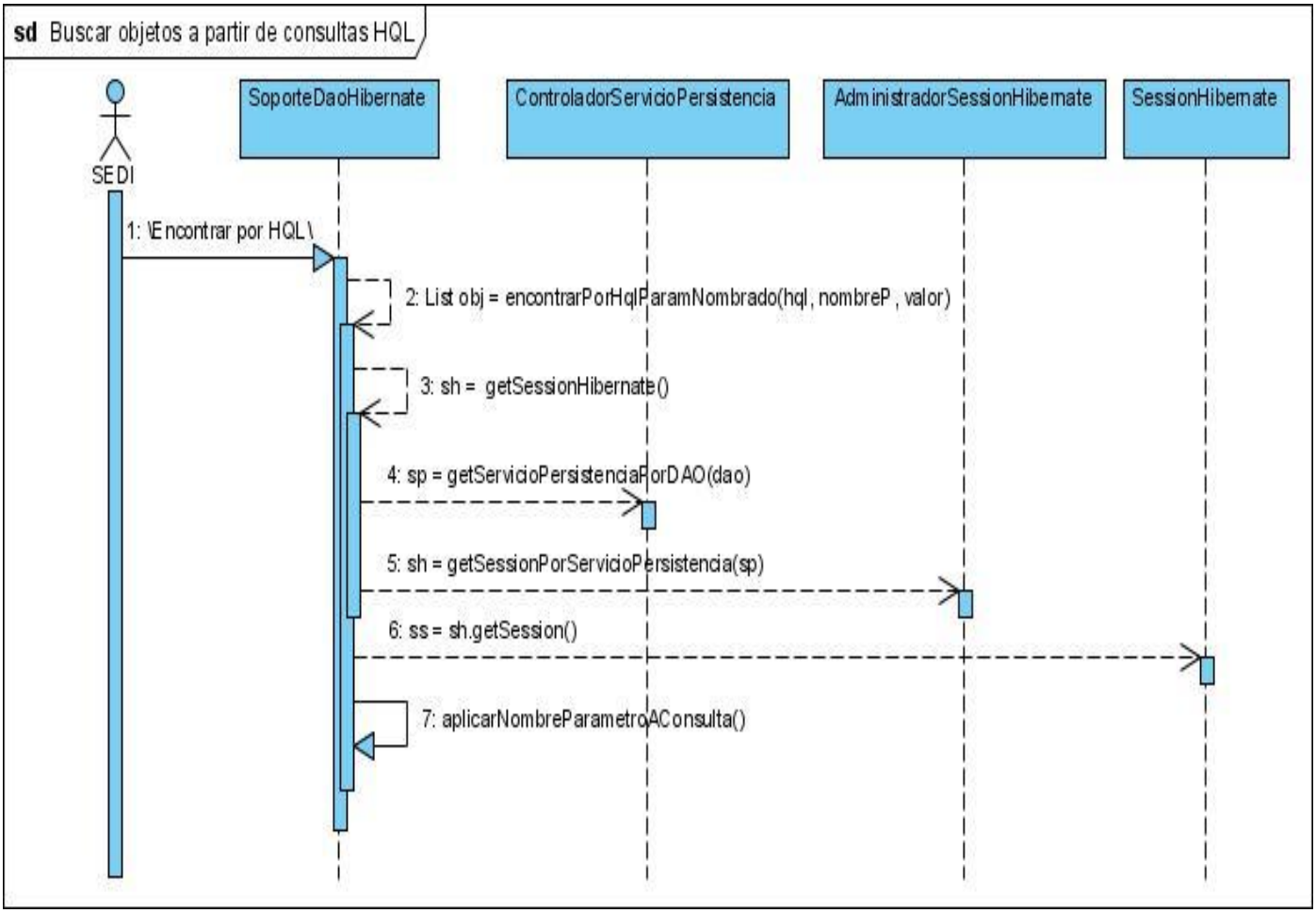


Figura 9: Diagrama de secuencia: Buscar objetos a partir de consultas HQL 2.

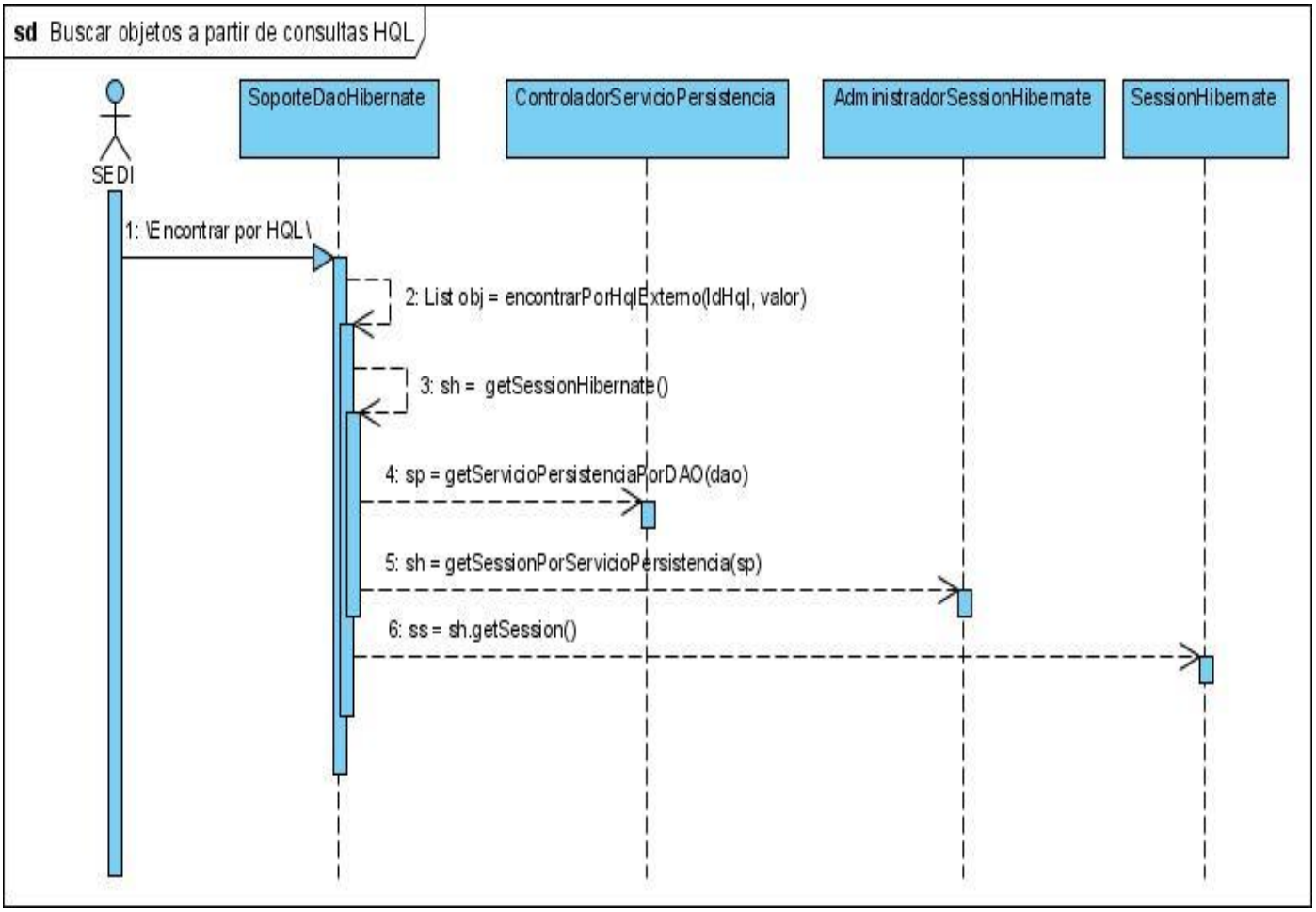


Figura 10: Diagrama de secuencia: Buscar objetos a partir de consultas HQL 3.

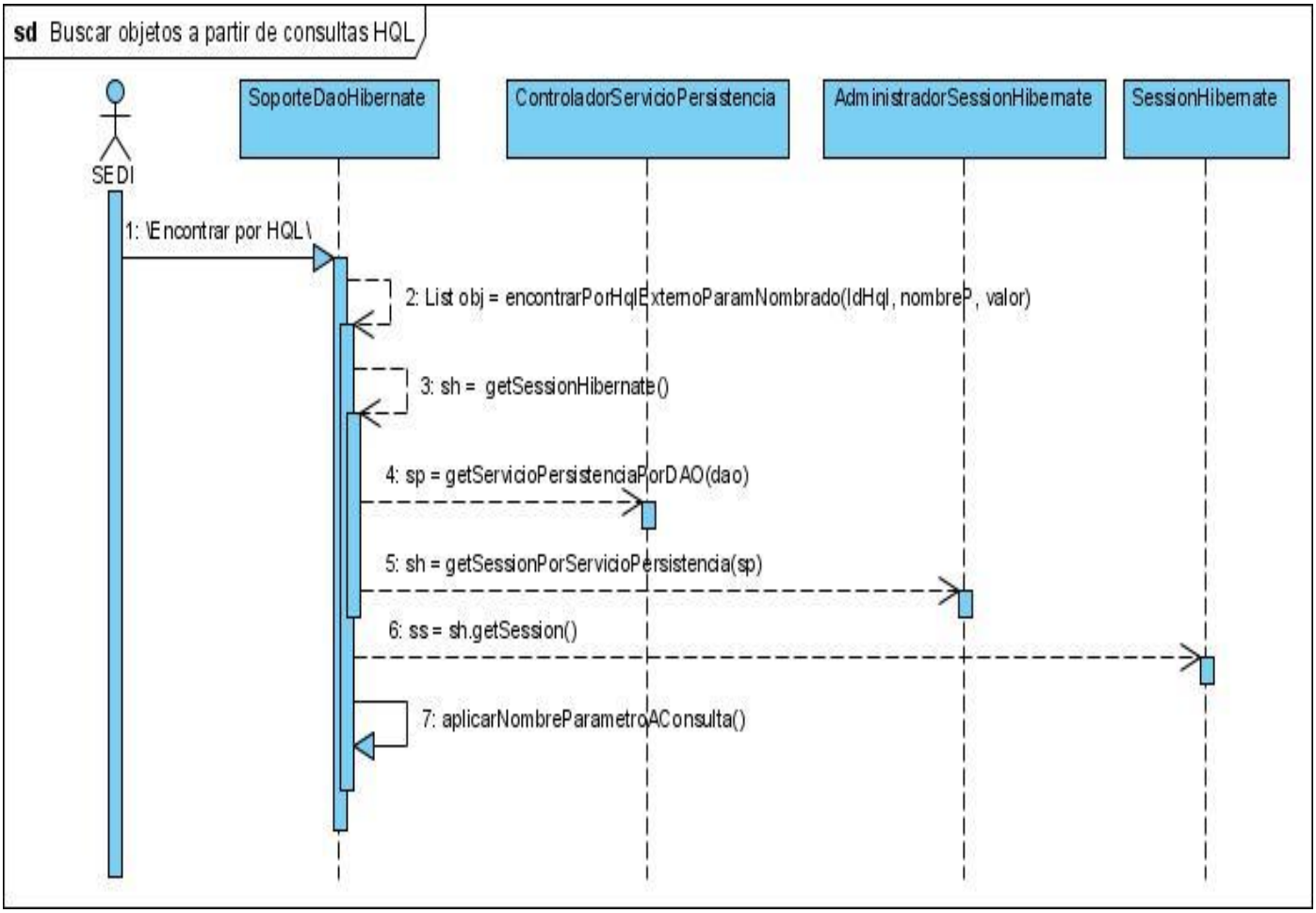


Figura 11: Diagrama de secuencia: Buscar objetos a partir de consultas HQL 4.

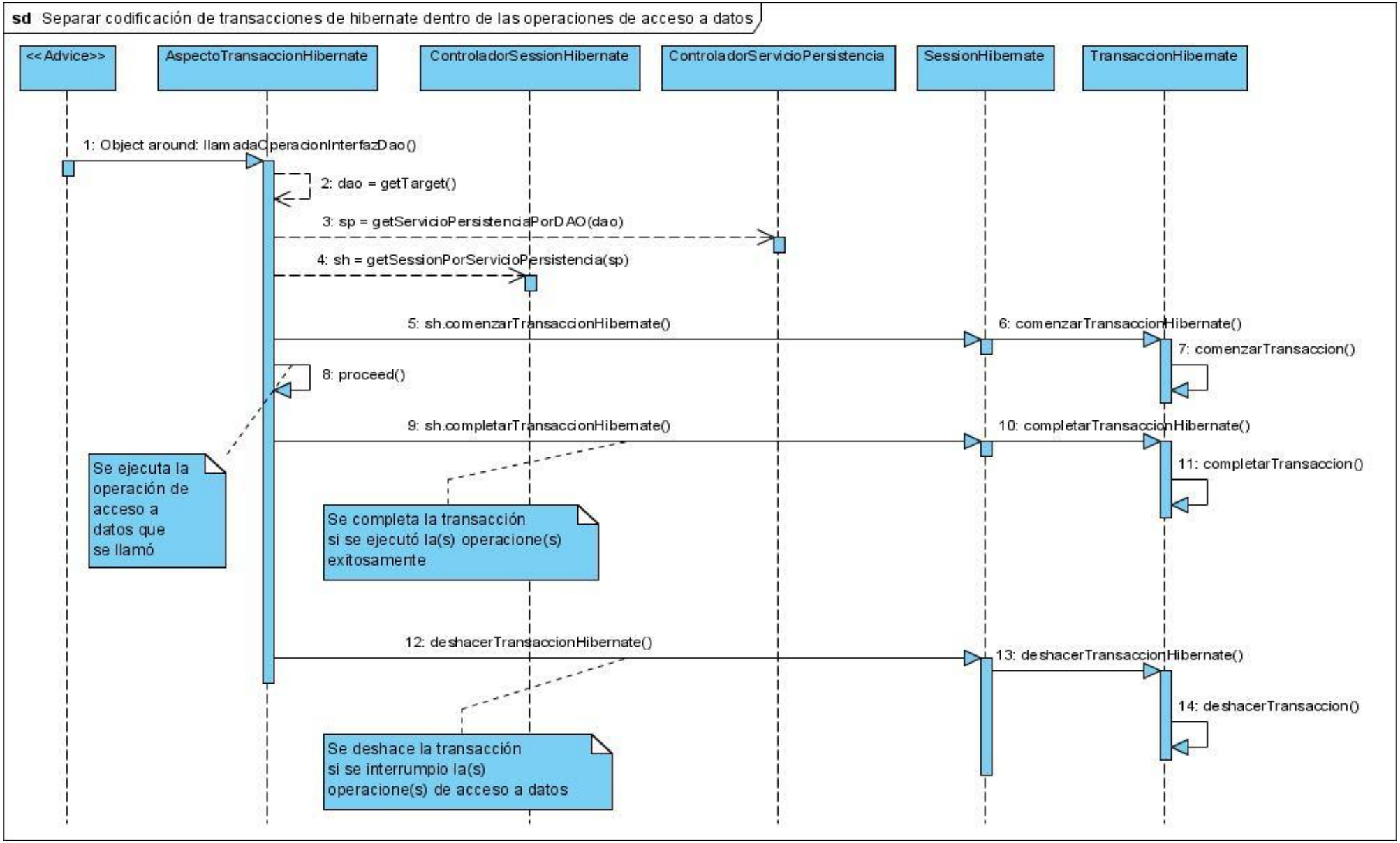


Figura 12: Diagrama de secuencia: Separar codificación de transacciones de hibernate dentro de las operaciones de acceso a datos.

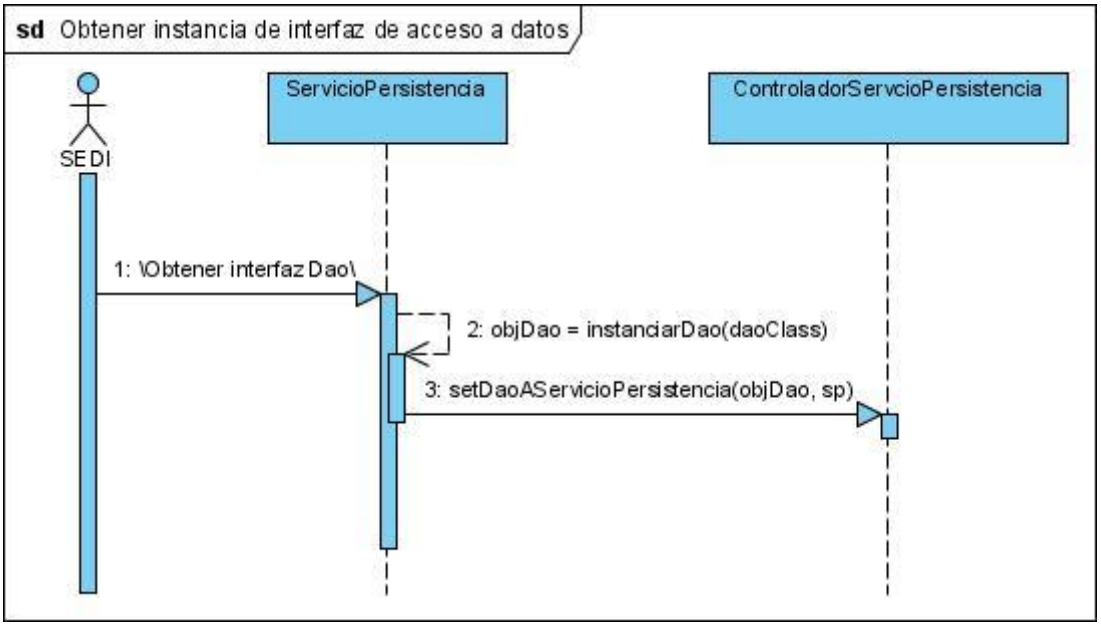


Figura 13: Diagrama de secuencia: Obtener instancia de interfaz de acceso a datos.

3.7 Diagrama de despliegue

El diagrama de despliegue es un modelo de objetos que describe la distribución física del sistema en términos de cómo se distribuye la funcionalidad del mismo entre los nodos de cómputo que utiliza. No es más que la representación física de todos los recursos que utiliza el sistema, por ejemplo, nodos con capacidad de procesamiento, módems e impresoras. (DISEÑO, 2005)

El siguiente diagrama de despliegue muestra la distribución física del componente de acceso a datos.

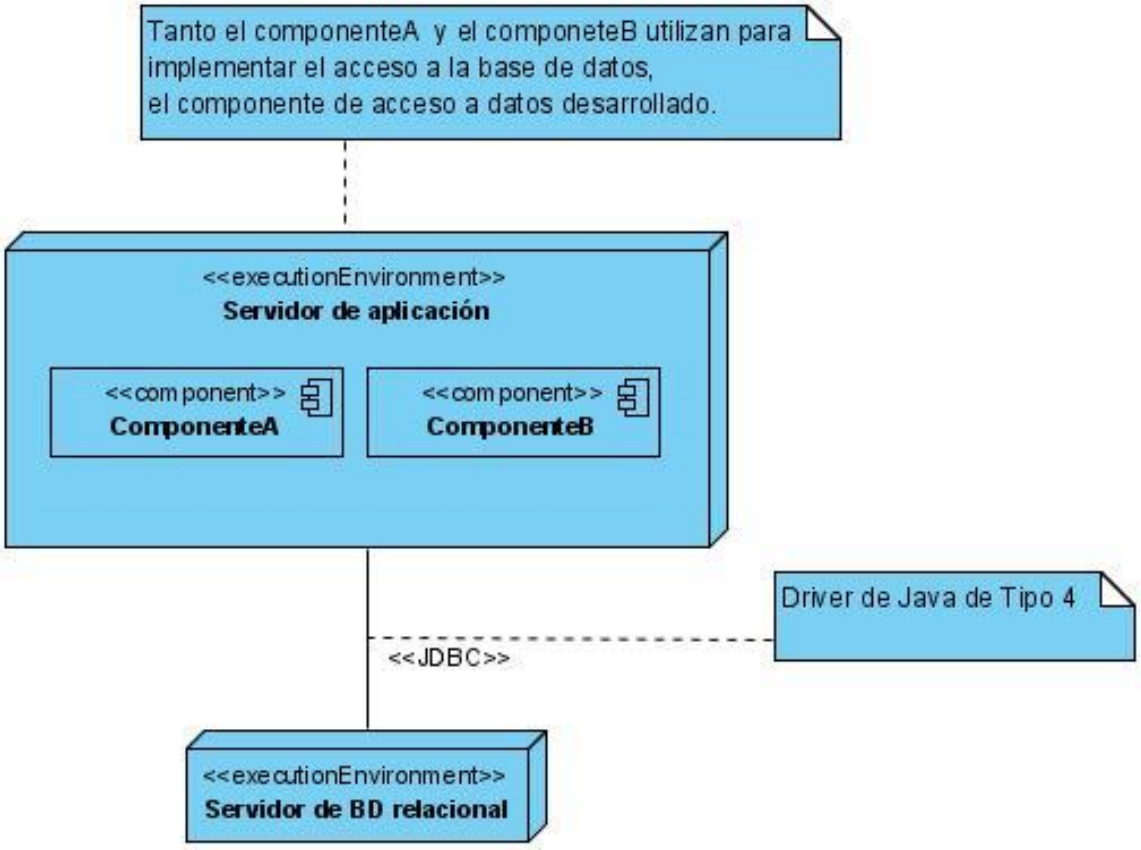


Figure 14: Diagrama de despliegue del componente.

El nodo Servidor de aplicación, representa un servidor de aplicación que utilice el Sistema de Emisión de Documentos de Identificación, el componente de acceso a datos se puede utilizar en cualquier servidor mientras que este se implemente con la tecnología Java. Dentro de este servidor se encuentran componentes que implementan las funcionalidades del Sistema de Emisión de Documentos de Identificación, donde cada componente desarrolla su capa de persistencia utilizando el componente de acceso a datos desarrollado.

El Nodo Servidor de BD relacional, representa un servidor de Base de Datos utilizado para el almacenamiento de los datos del Sistema de Emisión de Documentos de Identificación y para

lograr la conexión del sistema con la base de datos se utiliza JDBC como protocolo de comunicación.

Protocolo de comunicación JDBC, es la API de Java que utiliza la herramienta Hibernate (y por consecuencia el componente) para lograr la persistencia de los objetos. JDBC es una especificación de un conjunto de clases y métodos de operación que permiten a cualquier programa Java acceder a sistemas de bases de datos de forma homogénea. La aplicación de Java debe tener acceso a un driver JDBC adecuado. Este driver, preferentemente de tipo 4 (Anexo A), es el que implementa la funcionalidad de todas las clases de acceso a datos y proporciona la comunicación entre el API JDBC y la base de datos real.

3.8 Conclusiones

Como resultado de este capítulo se conoce como fue diseñado el componente de acceso a datos, partiendo de las clases del diseño definidas. También se conoce que patrón de diseño fue aplicado para el desarrollo del componente y se presentó una vista de alto nivel de la estructura del componente y sus relaciones con elementos externos a él. Se mostró la utilización del lenguaje de programación AspectJ que permitió modularizar todo el código relacionado con las transacciones de base de datos. Además se conoció cómo se llevan a cabo las funcionalidades que presenta el componente y la distribución física de los nodos de procesamiento que se necesitan para su funcionamiento.

CAPÍTULO IV. VALIDACIÓN DEL COMPONENTE

4.1. Introducción

En el desarrollo del software, las posibilidades de error son innumerables. Pueden darse resultados inesperados en las respuestas y comportamiento de un sistema, ya sea por uso indebido de las estructuras de datos, errores al enlazar módulos, etc. La prueba y validación de los resultados no es un proceso que se realiza una vez desarrollado el software, sino que debe efectuarse en cada una de las etapas de desarrollo. Es fundamental medir la cobertura de las pruebas, es decir, la determinación de cuando se han realizado las suficientes pruebas. Si se siguen encontrando errores cada vez que se procesa el programa, las pruebas deben continuar. Durante el mantenimiento debe de existir documentación de pruebas que incluya casos de prueba y resultados esperados. Si se producen modificaciones en el programa, habrá que probar de nuevo todas las partes del programa afectadas por las modificaciones. Por lo dicho anteriormente, se desarrolló este capítulo, con el objetivo de especificar las pruebas que fueron hechas al componente de acceso a datos.

4.2. Descripción de las pruebas

Siempre que se desarrolla algún tipo de software, hay que tener en cuenta las pruebas a realizar, con esto se asegura que nuestro programa o librería funcionen correctamente. Unas de las pruebas más comunes son las “pruebas de unidad”. Estas son pruebas llevadas a cabo por los implementadores sobre las unidades mínimas desarrolladas por ellos, estas unidades pueden ser clases, métodos, propiedades o componentes. Estas unidades se prueban separadas unas de otras y básicamente se realizan estas pruebas durante la implementación del software. Para la realización de las pruebas se debe tener en cuenta que:

- La prueba es un proceso de ejecución de un programa con la intención de descubrir errores.
- Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
- Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Las pruebas de unidades son orientadas en la mayoría de los casos a nivel de clases, ya que las clases representan la unidad básica por excelencia en el desarrollo orientado a objetos. Por lo que se realizarán pruebas de unidad a las clases, como también se le realizarán pruebas de unidad al componente como un todo una vez que se termine su desarrollo.

4.3. Aplicación de pruebas de unidad a nivel de clases

En esta sección se probarán por separadas las clases que conforman el componente de acceso a datos. Se utilizará el framework JUnit para Java, este framework permite automatizar los procesos de prueba que se llevarán a cabo. Para cada clase que se le realizan las pruebas de unidad se crearon las clases de pruebas:

Clase PruebaConfiguracionHibernate

Mediante la clase PruebaConfiguracionHibernate se verifica si se inicia con éxito la configuración de Hibernate. PruebaConfiguracionHibernate presenta los métodos de prueba *testIniciarConfiguracion()* y *testSessionFactoryConstruida()*. El primero de estos métodos comprueba la ejecución del método *iniciar(String XMLConfig)* de ConfiguracionHibernate. De manera secuencial *testSessionFactoryConstruida(),* una vez iniciada la configuración verifica si la sessionFactory ha sido creada. A continuación se muestra un disparo de pantalla una vez finalizado el proceso de ejecución de esta prueba.

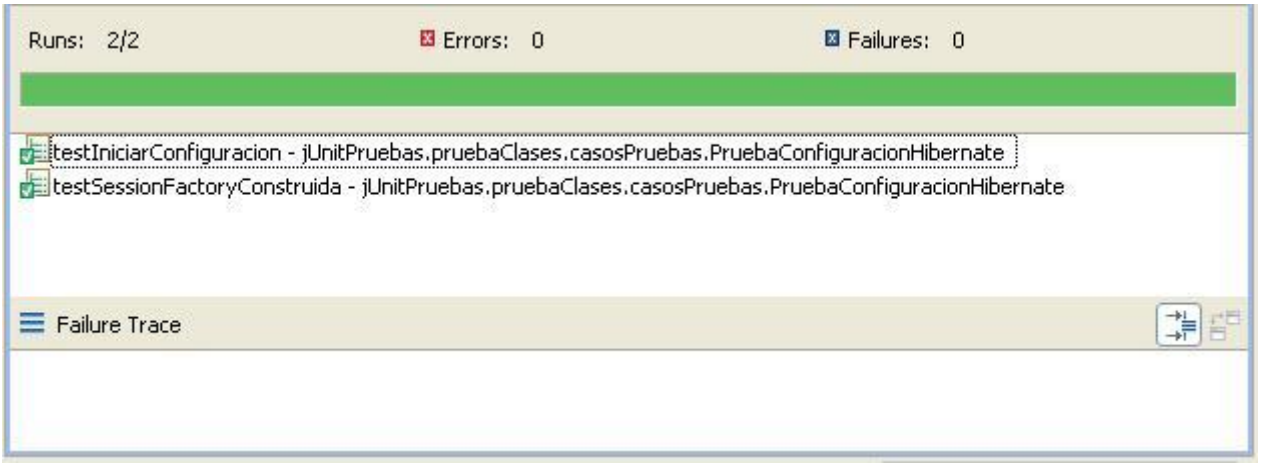


Figura 15: Ejecución de la clase PruebaConfiguracionHibernate.

En la figura anterior se aprecian los "test" ejecutados para comprobar que se configuró correctamente Hibernate usando la clase ConfiguracionHibernate. El icono verde que se ubica al lado del nombre del "test" indica que se completó con éxito. En caso de fallo el color se vuelve rojo. La barra de progreso en verde asegura que la totalidad de las pruebas se ejecutaron satisfactoriamente.

Clase PruebaSessionHibernate

La clase de prueba PruebaSessionHibernate contiene los métodos de pruebas *testSessionAbierta()*, *testManejoTransacciones()*, *testSessionCerrada()*. Antes de ejecutar estos métodos de prueba es necesario iniciar la configuración de hibernate e instanciar un objeto SessionHibernate para trabajar con él. El método *testSessionAbierta ()* verifica que esté abierta la sesión (de la interfaz Session de hibernate). Mediante las otras pruebas se verifica que se comienza, se completa y se deshace una transacción de base de datos con hibernate, y finalmente se prueba que se cierre la sesión abierta al inicio.

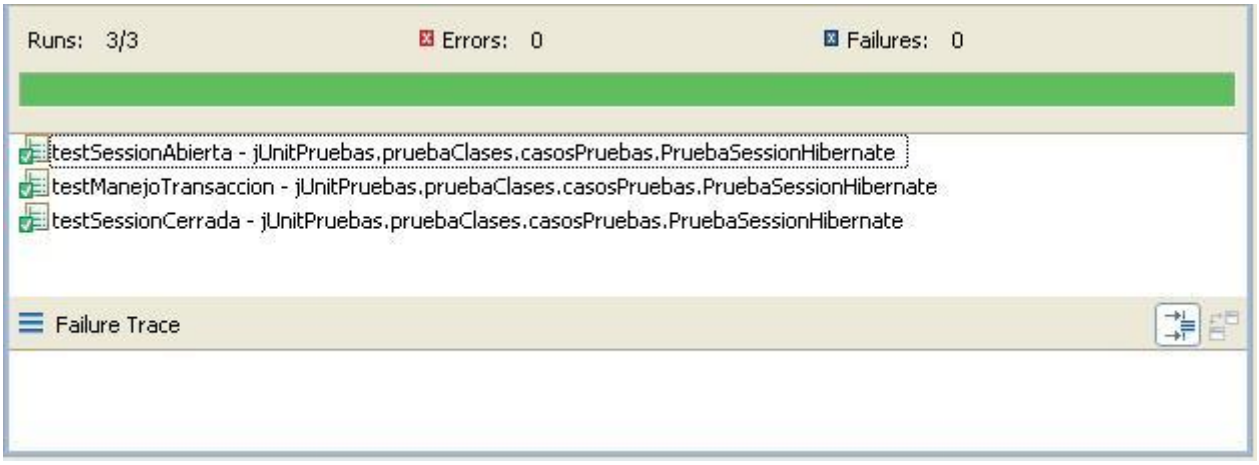


Figura 16: Ejecución de la clase PruebaSessionHibernate.

Clase PruebaServicioPersistencia

Para la ejecución de esta prueba es necesario definir una clase *ServicioPersistenciaPrueba* que extiende de *ServicioPersistencia* (clase abstracta que brinda el componente), además de crear una *InterfazDaoPrueba* y la clase concreta que la implementa *InterfazDaoPruebaImpl*. Una vez creada las condiciones necesarias se puede verificar que se instancia correctamente una interfaz de acceso a datos utilizando el método *instanciarDao(Impl.class)* de la clase *ServicioPersistencia*.

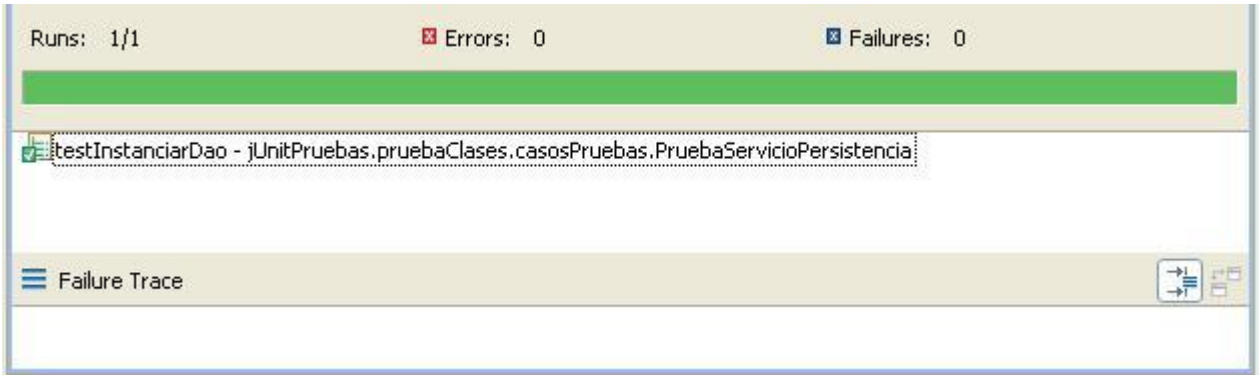


Figura 17: Ejecución de la clase PruebaServicioPersistencia.

4.4. Aplicación de pruebas de unidad a nivel de componente

Para probar el componente como un todo también se utilizó JUnit y se necesitó crear una base de datos experimental para implementar el acceso a esta utilizando el componente (Anexo B). Es válido aclarar que el diseño de esta base de datos se basa en una descripción de alto nivel de un proceso del SEDI para realizar una solicitud de pasaporte diplomático y que no se tuvieron en cuenta muchos detalles por lo que se proponen atributos a las tablas y relaciones entre estas tablas de manera lógica pero que no representa una base de datos fiel al proceso de solicitud de pasaporte, ni óptima para la producción.

Primeramente para desarrollar el acceso a la base de datos se crearon las clases entidades y los mapeos de estas clases con las tablas de la base de datos mediante archivos XML, también se creó el archivo de configuración de Hibernate necesario para conectarse con las base de datos (la creación de estos archivos son necesarios para poder utilizar el componente).

Posteriormente se creó la interfaz de acceso a datos IEntidadDao y la interfaz ISolicitudDao. Tanto IEntidadDao como ISolicitudDao al extender de la interfaz básica del componente IDaoBasica contendrán los métodos para buscar por un identificador, obtener todos los objetos de una tabla, buscar objetos a partir de una consulta por ejemplo, persistir un objeto de la base de datos y eliminarlo. Además la interfaz IEntidadDao se le adiciona un método para asignar personales a una entidad, a la interfaz ISolicitud una operación para crear una solicitud de pasaporte y dos operaciones que hacen uso del lenguaje HQL para consultar la base de datos (Anexo B).

Una vez realizados todos estos pasos se determinan dos clases de pruebas para probar las operaciones de la interfaz IEntidadDao e ISolicitudDao.

4.4.1. Definición de las clases de pruebas

Para probar las interfaces IEntidadDao e ISolicitudDao se crearon dos clases de pruebas. A continuación se presentarán estas dos clases de prueba y los casos de prueba que exponen cada una de ellas.

Clase EntidadPruebas

Nombre: EntidadPruebas	
Descripción general	
Permitirá probar la implementación de la interfaz que permite guardar, recuperar y consultar datos de la tabla *entidad* en la base de datos. Probará las operaciones definida en la interfaz que permite crear entidades en la base de datos, obtener todas la entidades de la tabla entidad de la base de datos, obtener entidades mediante una consulta por ejemplo, obtener una entidad por su identificador, eliminar una entidad y asignar personal a una entidad (ya sean directivos o titulares).	
Casos de pruebas:	
Nombre:	testCrearEntidades()
Descripción:	Permitió comprobar que el método *persistir(Entidad ObjetoEntidad)* funciona correctamente.
Nombre:	testObtenerTodasEntidades()
Descripción:	Prueba el funcionamiento del método *obtenerTodos()* que obtiene todos los objetos de la tabla *entidad.* En este caso de prueba se verifica si se obtienen todos las entidades creadas en el"test" anterior.
Nombre:	testObtenerEntidadPorEjemplo()
Descripción:	Se prueba que el método *obtenerPorEjemplo(Entidad ejemploEntidad)* devuelve lo que se espera comprobando de esta forma su buen funcionamiento.
Nombre:	testObtenerEntidadPorId()
Descripción:	Se comprueba que se obtiene una entidad a partir de su identificador en la base de datos. Si no existe se verifica que devuelva un objeto nulo.
Nombre:	testEliminarEntidad()
Descripción:	Se prueba que se elimine correctamente una entidad mediante el método

	eliminar(Entidad entidadEliminada).
Nombre:	testAsignarPersonal
Descripción:	Se prueba el método que permite asignarles personales a una entidad específica a partir de sus siglas. La operación de la interfaz que se prueba es *adicionarPersonal(String siglas, Personal[] personales).*

Tabla 4.13: Descripción de la clase de prueba EntidadPruebas.

A continuación se muestran en la siguiente figura la ejecución de la clase EntidadPruebas con el resultado de cada caso de prueba utilizando el framework JUnit.

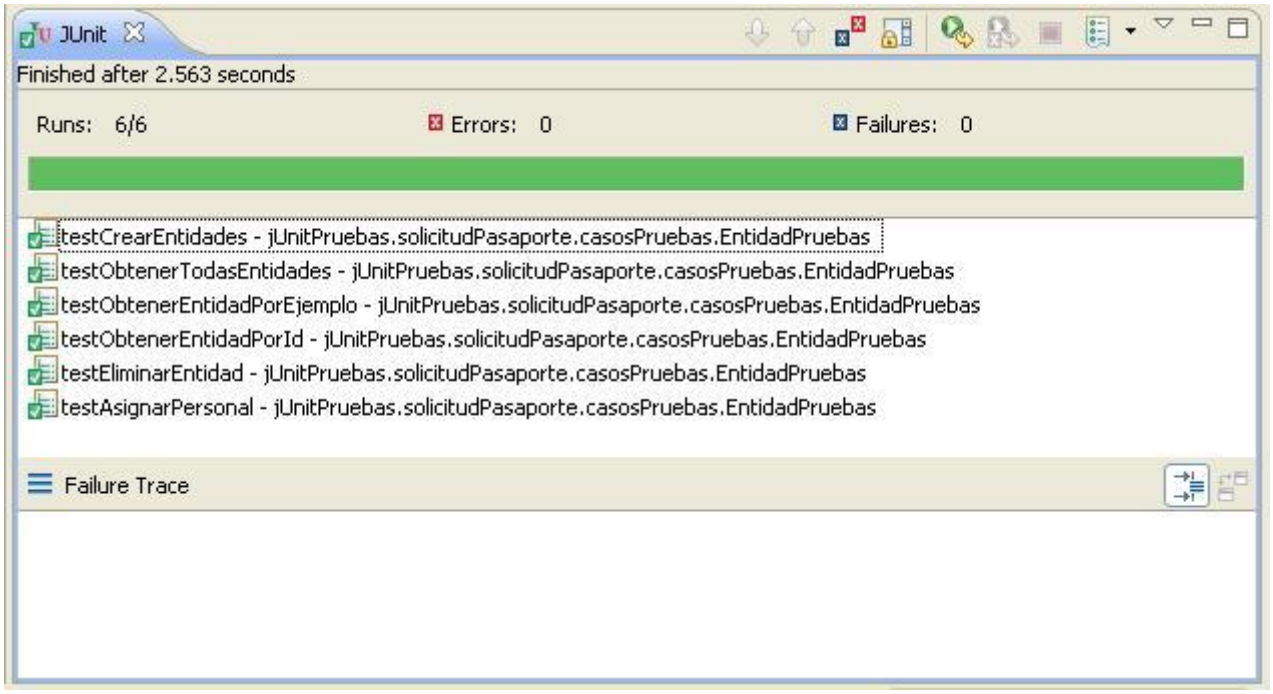


Figura 18: Ejecución de la clase EntidadPruebas

Clase SolicitudPruebas

Nombre: SolicitudPruebas	
Descripción general	
Permitirá probar la implementación de la interfaz que permite consultar datos de la tabla *solicitud_pasaporte* en la base de datos. Probará las operaciones definida en la interfaz que permite realizar consultas a la base de datos utilizando el lenguaje de consulta de Hibernate (HQL).	
Casos de pruebas:	
Nombre:	testCrearSolicitud()
Descripción:	Con este caso de prueba se verifica que se crea correctamente una solicitud. La solicitud debe ser asignada por un directivo hacia un titular. Se comprueba el buen funcionamiento de la operación *crearSolicitud(String idTitula, String IdDirectivo, Solicitus solicitud)*.
Nombre:	testObtenerSolicitudPorFechaCita()
Descripción:	Con este caso de prueba se verifica que se ejecuta correctamente una consulta HQL escrita en un archivo XML. La consulta HQL escrita en el XML es utilizada por el método *obtenerSolicitudPorFecha(Date fechaCita)*.
Nombre:	testObtenerSolicitudesAceptadas()
Descripción:	Con este caso de prueba se verifica que se ejecuta correctamente una consulta HQL escrita en el mismo código del programa. Se comprueba el buen funcionamiento del método *obtenerSolicitudesAceptadas()* de esta interfaz.

Tabla 4.14: Descripción de la clase de prueba SolicitudPruebas.

A continuación se muestran en la siguiente figura la ejecución de la clase SolicitudPruebas con el resultado de cada caso de prueba.

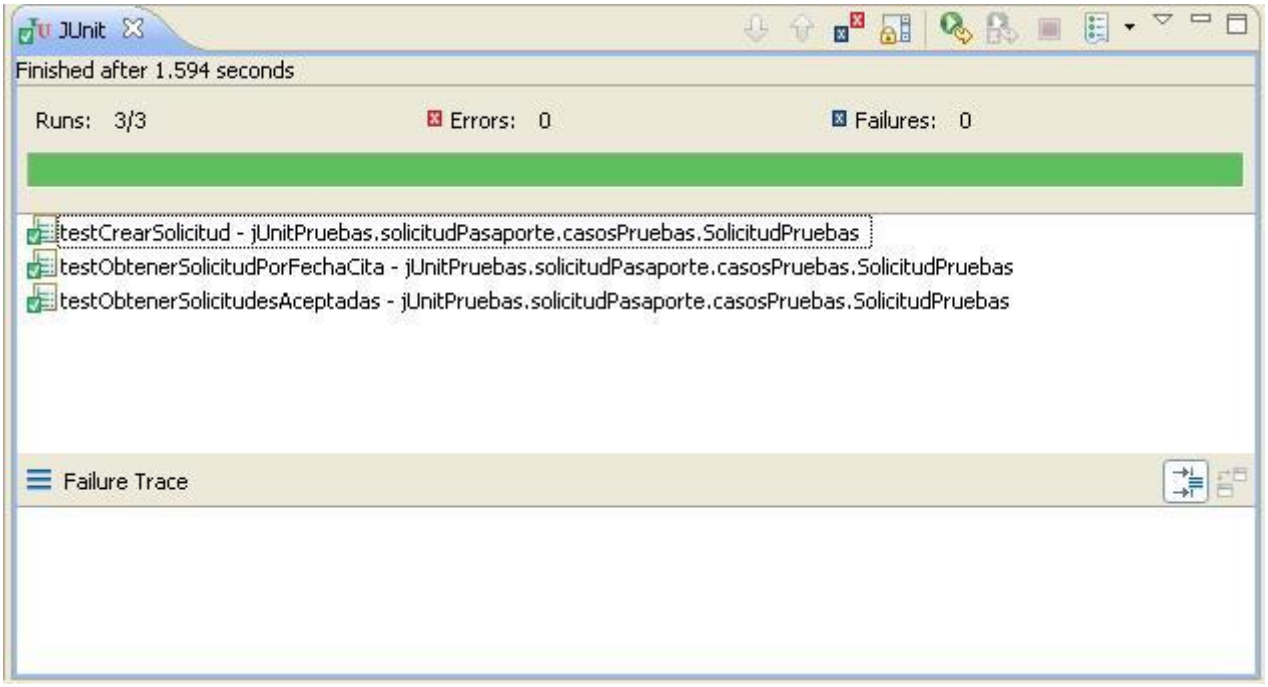


Figura 19: Ejecución de la clase SolicitudPruebas.

4.5. Conclusiones

Una de las últimas fases del ciclo de vida antes de entregar un programa para su explotación, es la fase de pruebas. Esta fase del desarrollo de un software es la que mayor cantidad de tiempo y de esfuerzo requiere, se estima que la mitad del esfuerzo de desarrollo de un programa tanto en tiempo como en gastos se invierte en esta. Esta fase añade valor al producto, todos los programas poseen errores y la fase de pruebas los descubre, siendo este el valor que le añade. Mientras más errores se encuentren al software en esta fase mejor. Una prueba del software es un conjunto de actividades que se lleva a cabo sistemáticamente, que puede planificarse por adelantado y ejecutar una vez construido el código para la revisión final del diseño y de la codificación del software.

En este capítulo con la realización de las pruebas de unidad a las clases del componente por separado y con la realización de un ejemplo práctico de acceso a datos, se pudo comprobar que el componente realiza correctamente el manejo de conexión, así como el manejo de transacciones a la base de datos. También se pudo probar mediante el ejemplo práctico que las operaciones de soporte brindadas por el componente funcionan correctamente.

CONCLUSIONES

Como conclusiones generales de la investigación se muestran alcanzados los objetivos propuestos satisfactoriamente:

- *Seleccionar una herramienta de persistencia que permita eliminar la diferencia de impedancia entre el mundo relacional de las base datos y el mundo orientado a objetos*. Para darle cumplimiento a este objetivo se analizaron dos herramientas ORM muy utilizadas, este análisis posibilitó la elección del framework Hibernate para desarrollar el componente de acceso a datos propuesto. Al seleccionar esta Herramienta de mapeo objeto-relacional se logró que la implementación del acceso a datos, utilizando el componente, se realice totalmente orientada a objeto, aprovechando además todas las capacidades para gestionar la base de datos con Hibernate.

- *Manejar transacciones de acceso a datos utilizando programación orientada a aspectos*. Para llevar a cabo este objetivo se utilizó el lenguaje AspectJ como lenguaje de programación orientado a aspectos. Mediante este lenguaje se logró separar de las operaciones de acceso a datos todo código dedicado a las transacciones de base de datos, eliminando de esta forma la dispersión de código por toda la aplicación al escribir estas transacciones.

- *Realizar diseño e implementación del componente*. Para la realización de este objetivo se partió de la creación de un modelo general que permitió identificar una lista de rasgos a partir de la cual se logró diseñar e implementar el componente.

- *Realizar pruebas unitarias al componente de acceso a datos*. Para la realización de las pruebas de unidad del componente se utilizó el framework JUnit. Este permitió realizar las

pruebas tanto a nivel de clases, como a nivel del componente en general, comprobando de esta forma el buen funcionamiento del componente de acceso a datos.

Por lo anteriormente descrito, se concluye el cumplimiento del objetivo general de la investigación de desarrollar un componente de acceso a datos, que facilite el trabajo de los programadores de la capa de persistencia en soluciones de emisión de documentos de identificación.

RECOMENDACIONES

Teniendo en cuenta el objetivo que se persigue con el desarrollo del componente de acceso a datos, se recomienda que para próximas versiones del componente, este incorpore las siguientes funcionalidades:

- Que mediante el componente de acceso a datos se genere automáticamente las interfaces de acceso a datos y sus implementaciones a partir de un esquema de base de datos existentes. Donde cada interfaz y su implementación generada ya extiendan de la interfaz IDaoBasica y la clase SoporteDaoHibernate respectivamente, para eliminar que este proceso se realice manualmente.

- También que el componente incorpore algún mecanismo propio que permita a los programadores, probar unitariamente las operaciones de acceso a datos implementadas mediante el componente de acceso a datos.

REFERENCIAS BIBLIOGRÁFICAS

CALABRIA, L. *Metodología FDD.* Uruguay: 2003, Disponible en: http://athenea.ort.edu.uy/publicaciones/ingsoft/investigacion/ayudantias/metodologia_FDD.pdf.

CEYUSA. *El problema de la diferencia de impedancia.* Disponible en: http://www.glib.org.mx/article.php?story=20060611151541892

CHRISTIAN BAUER, G. K. Hibernate in Actions. 2005.

DISEÑO. Material de Apoyo Conferencia Diseño. Sitio de la Asignatura Ingeniería de Software, 2005. Disponible en: http://teleformacion.uci.cu/

HYDE, J. *Base de datos*. 2002.
Disponible en: http://www.monografias.com/trabajos11/basda/basda.shtml

JUAN BERNARDO QUINTERO, D. M. H., ANDREA YANZA. *Directrices para la construcción de artefactos de persistencia en el proceso de desarrollo de software*. Colombia: 2008.
Disponible en: http://revista.eia.edu.co/articulos9/articulo%206.pdf

JULIAN SKINNER, B. J., DONNY MACK. *Professional ADO.Net Programming*, 2001.

KEVIN MUKHAN, T. L., JOHN CARNNELL. *Fundamentos Base Datos con Java*, 2002.

LAGO, R. *Patrones de diseños*. Disponible en: http://www.proactiva-calidad.com/java/patrones/index.html

MOREA, L. Introducción a Java.
Disponible en: http://www.monografias.com/trabajos/java/java.shtml

MORENO, J. M. N. *AspectJ en la Programacion Orientada a Aspectos*. España: Universidad de sevilla, 2006. Disponible en: http://es.geocities.com/nulain/computing/AspectJ

MONTERO, J. *Proyecto Eclipse* Disponible en: http://150.244.56.228/descargas_web/cursos_verano/20040801/Jesus_Montero/Entornos_de_programacion.pdf.

MSDN. *Transacciones*. Página específica de Microsoft Visual Studio 2008/.NET Framework 2003. 2005. Disponible en: http://msdn.microsoft.com/es-es/library/tyes10w5.aspx

REFERENCIAS BIBLIOGRÁFICAS

PÉREZ, F. C. *IBatis.* Disponible en: http://www.juntadeandalucia.es/xwiki/bin/view/MADEJA/iBatis

STEPHEN R PALMER, J. M. F. *A Practical Guide to Feature-Driven Development*. 2002. Disponible en: http://www.amazon.com/exec/obidos/ASIN/0130676152#reader

USAOLA, M. P. *Pruebas de programas Java mediante JUnit*. España: Tutorial de JUnit de la Escuela Superior de Informática de Ciudad Real (Universidad de Castilla-La Mancha). Disponible en: http://www.inf-cr.uclm.es/www/mpolo/tutorial/

VALDÉS., D. P. *¿Qué son las bases de datos?*
Disponible en: http://www.maestrosdelweb.com/principiantes/%C2%BFque-son-las-bases-de-datos/

BIBLIOGRAFÍA CONSULTADA

1. AMARO CALDERON, S. D., VALVERDE REBAZA. *Metodoloías Ágiles.* Perú: 2007, Disponible en: http://www.seccperu.org/files/Metodologias%20Agiles.pdf
2. ANTONIUCCI, J. *Persistencia Básica en Java.* 2007, Disponible en: http://www.adictosaltrabajo.com/tutoriales/pdfs/PersistenciaJava.pdf.
3. CARLOS REYNOSO, N. K. *Estilos y Patrones en la Estrategia de Arquitectura de Microsoft.* Buenos Aires: 2004, Disponible en: http://www.willydev.net/InsiteCreation/v1.0/descargas/prev/estiloypatron.pdf
4. EUROPA, E. Disponible en: http://www.eclipse.org/europa/
5. JUAN B. QUINTERO, D. M. H., RAQUEL ANAYA DE P. *Experiencia Práctica de la Aplicación de Aproximaciones Orientadas por Aspectos en el Desarrollo de un Portal Temático.* Medellín. Colombia: 2007, Disponible en: http://pisis.unalmed.edu.co/avances/archivos/ediciones/Edicion%20Avances%202007%201/14.pdf
6. LADDAD, R. *Aspect in Action. Practical Aspect-Oriented Programming.* 2003.
7. LUCA, J. D. *Sitio oficial de discusión de FDD* Disponible en: http://www.featuredrivendevelopment.com/
8. LTD., N. P. Página Web de consultaría de Jeff De Lucas. Disponible en: http://www.nebulon.com/index.html
9. MILES, R. *AspectJ Cookbook.* 2004.
10. MOLPECERES, A. *Procesos de desarrollo: RUP, XP y FDD.* 2002, Disponible en: http://www.willydev.net/InsiteCreation/v1.0/descargas/articulos/general/cualxpfddrup.pdf.
11. MOTA, K. *IBatis SQL Maps.* 2006, Disponible en: http://svn.apache.org/repos/asf/ibatis/trunk/java/ibatis-2/ibatis-2-docs/es/iBATIS-SqlMaps-2-Tutorial_es.pdf
12. NUÑEZ, I. *Tutorial de Hibernate.* 2007, Disponible en: http://www.scribd.com/doc/454457/tutorialHibernate.

BIBLIOGRAFÍA CONSULTADA

13. ORTEGA, Á. L. C. *Hibernate.* Murcia: 2005, Disponible en: http://ditec.um.es/ssdd/trabajos/0506/Hibernate-Angel_Luis_Calvo_Ortega.pdf
14. SARAH GUTIÉRREZ, H. Z., JUAN PABLO ARIAS, CRISTIAN ZAMBTANO. *Desarrollo Basado en Rasgos.* Disponible en: http://pisis.unalmed.edu.co/cursos/material/3004582/1/PresentacionFDD.ppt
15. SCOTT OAKS, H. W. *Java Threads.* O'Reilly, 2004.
16. *Sitio oficial del Framework Hibernate*. Disponible en: https://www.hibernate.org/
17. *Sitio oficial del Framework IBatis*. Disponible en: http://www.ibatis.apache.org
18. *Sitio oficial de Visual Paradigm*. Disponible en: http://www.visual-paradigm.com/
19. VINCENT MASSOL, T. H. *JUnit in Action.* 2004.

GLOSARIO

API: Application Programming Interface (Interfaz de Programación de Aplicaciones), es un conjunto de convenciones internacionales que definen cómo debe invocarse una determinada función de un programa desde una aplicación. Cuando se intenta estandarizar una plataforma, se estipulan unos APIs comunes a los que deben ajustarse todos los desarrolladores de aplicaciones.

Clase POJO: acrónimo de Plain Old Java Object, es una sigla utilizada por programadores de Java para enfatizar el uso de clases simples que no dependen de un framework en especial. Estas clases se caracterizan por tener solo los métodos comúnmente llamados “get” y “set”, además de mantener en ocasiones asociaciones con clases del mismo tipo.

Componente: es una parte no trivial, casi independiente, y reemplazable de un sistema que llena claramente una funcionalidad dentro de un contexto en una arquitectura bien definida.

Framework: es un marco de trabajo que expone un conjunto de librerías que no forman parte de una aplicación específica y que han sido creadas para ser utilizados por cualquier aplicación. Sus componentes pueden ser usados para propósitos generales y están ampliamente probados para funcionar en gran variedad de escenarios.

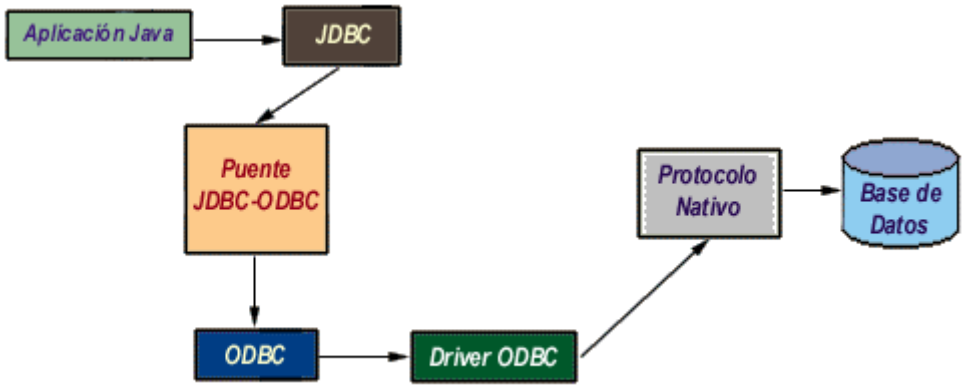
Manejador de base de datos: representa un conjunto de clases que permite comunicarse con un sistema de base de datos. En el caso del la API JDBC solo se exponen interfaces para acceder a la base de datos y los manejadores de base de datos tiene las clases concreta que implementan estas interfaces para acceder al sistema de base de datos específico.

Software Libre: es el software que una vez obtenido, puede ser usado, copiado, estudiado, modificado y redistribuido libremente. El software libre suele estar disponible gratuitamente.

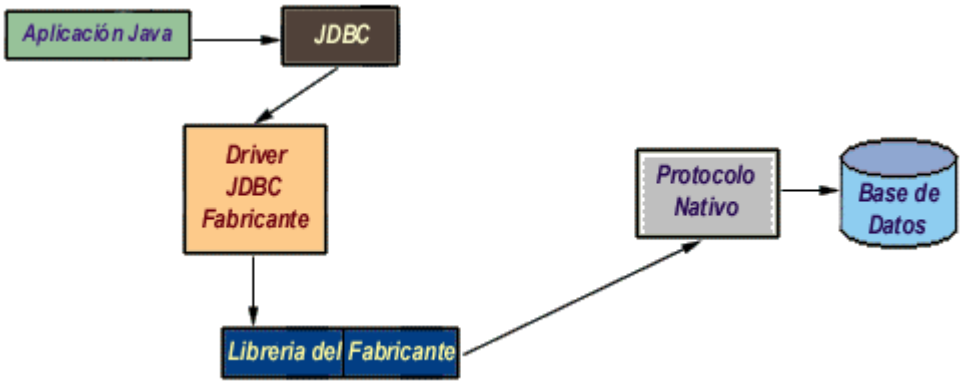
XML: Extensible Markup Language (Lenguaje de Marcas Extensible), es un metalenguaje que define la sintaxis utilizada para definir otros lenguajes de etiquetas estructurados.

ANEXOS

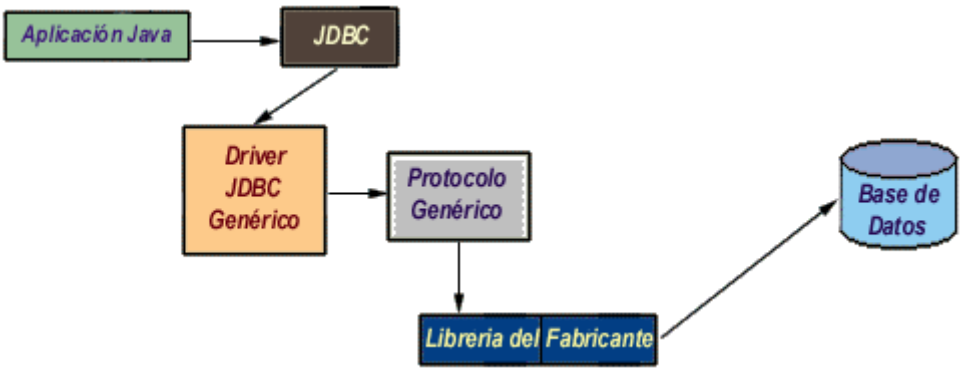
ANEXO A. Tipos driver de JDBC



Driver JDBC de tipo 1



Driver JDBC de tipo 2.



Driver JDBC de tipo 3

Driver JDBC de tipo 4.

ANEXO B. Diagramas generados para las pruebas unitarias al componente.

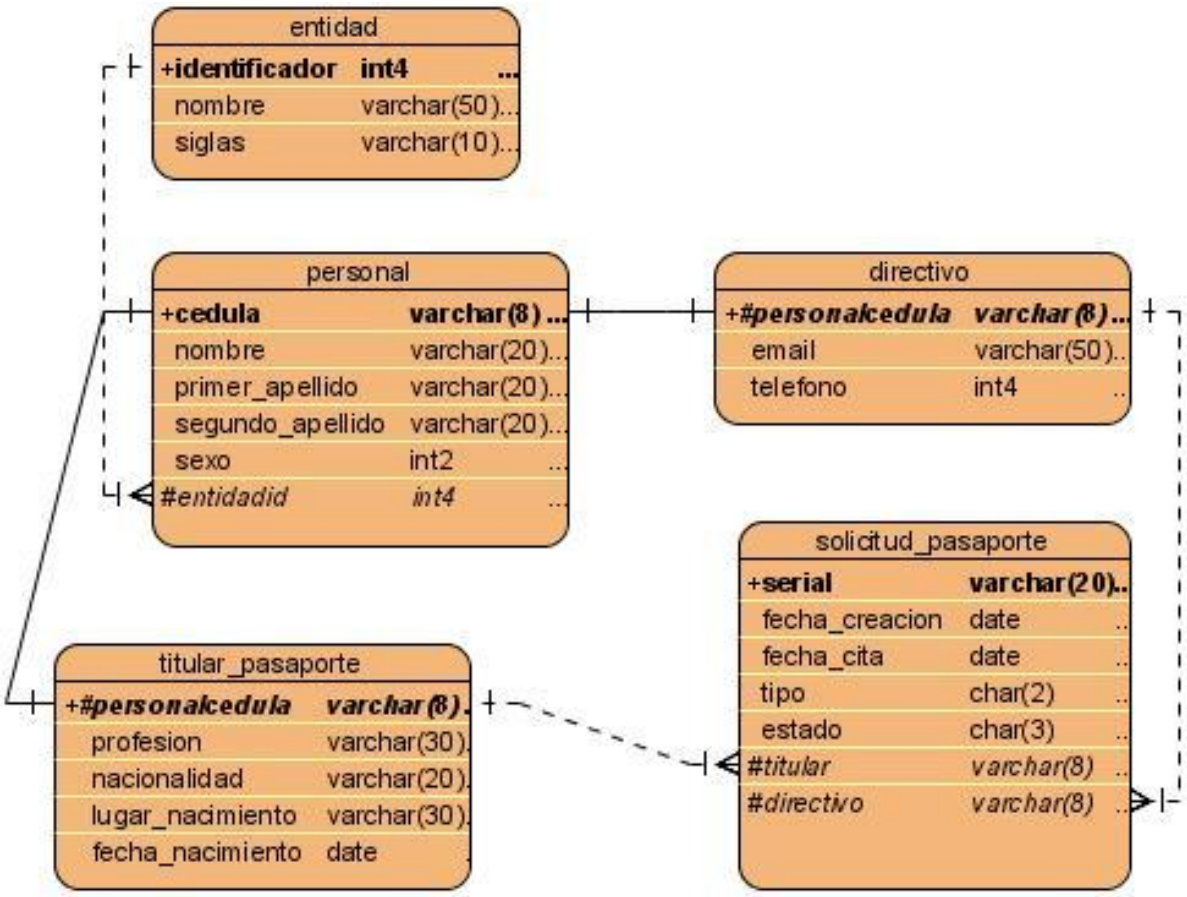


Diagrama Entidad-Relación de la base de datos experimental para las pruebas unitarias.

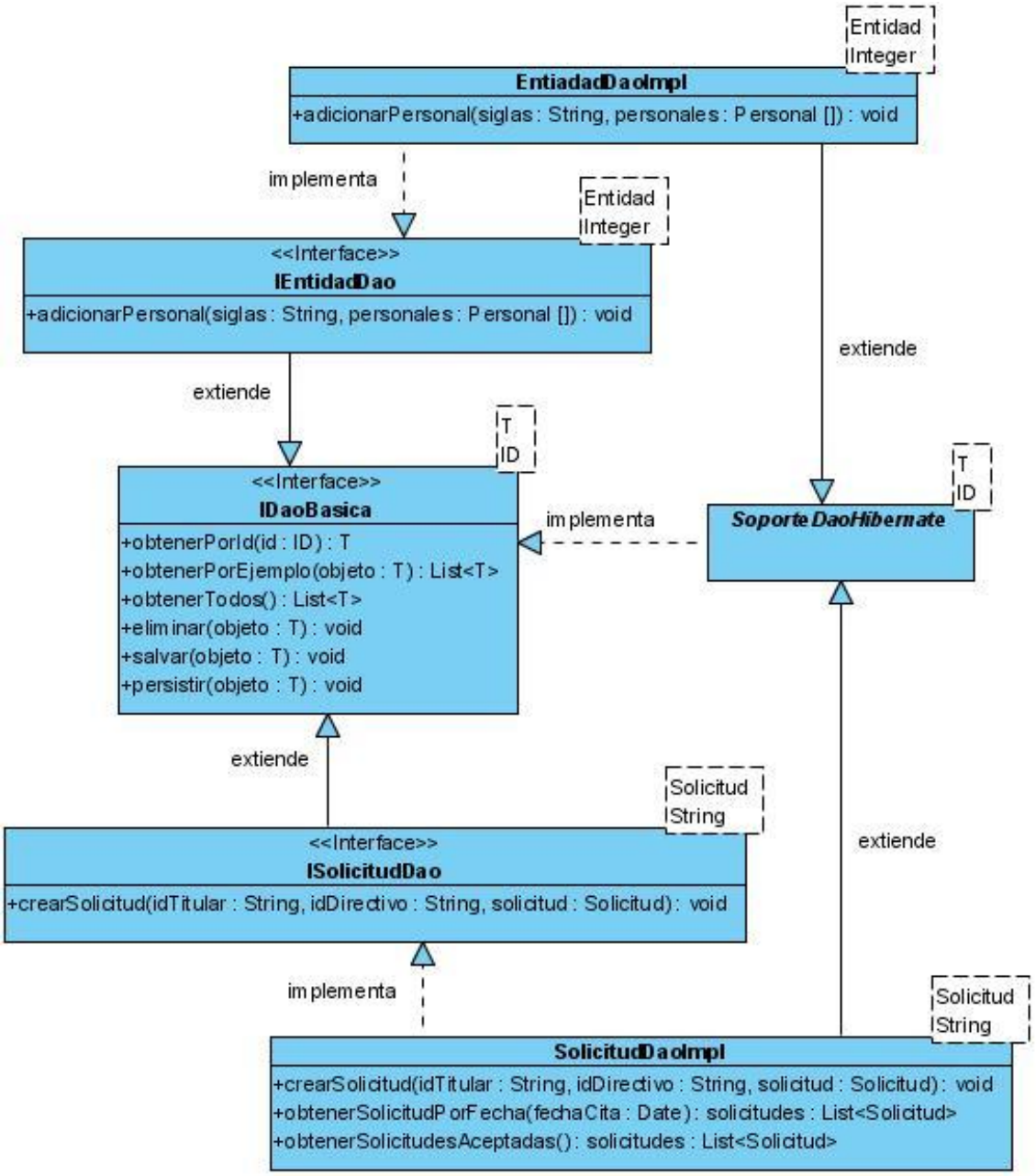


Diagrama de clases de diseño para las pruebas unitarias del componente.