PFC: J2EE. Diseño e implementación de un Framework de Persistencia

Javier Alvarez-Uria Tejero
Ingeniería en Informática

Consultor: Josep Maria Camps Riba
14 de Enero de 2009

RESUMEN DEL PROYECTO

El objetivo de este proyecto es el análisis y desarrollo de un conjunto de componentes que simplifiquen y agilicen el desarrollo de la capa de persistencia en aplicaciones multicapa, especialmente para las aplicaciones desarrolladas con J2EE.

Una de las partes más complejas que hay que resolver al plantear cualquier aplicación es el acceso a los datos persistentes que tiene que gestionar (muchas veces bases de datos relacionales). En aplicaciones JEE hay diversas opciones para implementar este acceso: uso de JDBC directamente, uso de EJBs, uso de marcos de trabajo que faciliten un mapeo entidad-relación, etc. Lo que se pretende con este PFC es estudiar la problemática de la capa de acceso a datos de una aplicación distribuida y escoger una estrategia para resolverlo.

Para ello, primero se va a hacer un estudio de los conceptos y terminos que se emplean en la capa de persistencia para tener un conocimiento de la misma y a a continuación se evaluan las diferentes alternativas existentes, centrandonos en los frameworks de mapeo entidad-relación existentes al mercado.

Se van a analizar los diferentes enfoques de cada uno de ellos, las características que tienen en común y las que los hacen diferentes. A continuación, se va a realizar el diseño y la implementación de un framework de este estilo que implemente las funcionalidades básicas.

El framework que se ha diseñado esta formado por un conjunto de clases que proporciona la abstracción de algún concepto de la capa de persistencia y que nos va a ayudar a interaccionar con la misma.

Finalmente, se va a implementar una aplicación de ejemplo para mostrar el funcionamiento del framework implementado.

Por tanto, el objetivo de este PFC es triple

- Estudiar y evaluar las diferentes alternativas existentes para implementar la capa de persistencia de aplicaciones JEE de cliente delgado
- Una vez estudiadas las alternativas disponibles, realizar el diseño y la implementación de un framework de mapeo entidad-relación
- Construir una aplicación de ejemplo que muestre el uso del framework implementado.

PALABRAS CLAVE

- J2EE.
- Base de datos.
- Java
- SGBD.
- Persistencia.
- SQL.
- Aplicación Web.

LICENCIA CREATIVE COMMONS

ÍNDICE

1 INTRODUCCIÓN

1.1 DESCRIPCIÓN DEL PFC

Actualmente, todas las empresas almacenan la información que manejan (clientes, empleados, productos, etc.) en sistemas persistentes, los cuales permiten preservar los datos para su posterior uso. El sistema más común y extendido es el uso de una base de datos relacional.

Alrededor de estos sistemas y para facilitar la extracción de información se construyen aplicaciones que se alimentan de los datos almacenados en la base de datos, los procesan, hacen los cálculos necesarios y finalmente se los presenta al usuario. Así mismo, estas aplicaciones, permiten operar con los datos, modificarlos, dar de alta nueva información y borrar información de una forma sencilla y flexible.

En este proyecto nos vamos a centrar en las aplicaciones desarrolladas con JEE, cuyo lenguaje de programación es Java y que sigue un modelo de programación orientado a objetos.

Cuando se desarrollan sistemas orientados a objetos que usan una base de datos relacional, los desarrolladores deben invertir una gran cantidad de tiempo en hacer los objetos persistentes, es decir, convertir los objetos del lenguaje de programación a registros de la base de datos. De la misma manera, hay que implementar la operación inversa, es decir, convertir los registros en objetos.

Esto es debido a la diferencia lógica entre los dos paradigmas, los objetos tienen datos y operaciones así como herencia, mientras que las bases de datos relacionales contienen tablas, atributos y relaciones entre tablas.

Para persistir un objeto en una base de datos relacional, el uso más común es que el desarrollador utilice llamadas JDBC y persista el objeto o construya el objeto a partir de los resultados. Esta opción hace que el desarrollo sea costoso. Además, el desarrollador debe tratar con aspectos complejos tan comunes como abrir la conexión con la base de datos, cerrar la conexión, comprobación de bloqueos de escritura y lectura, conversión de tipos entre los campos de la tabla y los atributos del objeto, etc. Tareas repetitivas en las que resulta fácil equivocarse.

Una solución mejor pasa por la utilización de un framework que se encargue de realizar estas tareas de manera transparente al programador, de manera que éste no tenga por qué utilizar JDBC ni SQL, y que la gestión del acceso a la base de datos este centralizado en un componente único, permitiendo su reutilización. Para ello, el uso de un framework de mapeo de objetos-relacional (ORM) es la opción más óptima y que mayores ventajas aporta.

El objetivo de este proyecto es realizar un estudio general de los frameworks (que son, para que sirven, características,..) para centrar el estudio en los frameworks de persistencia mas relevantes para construir la capa de persistencia en aplicaciones desarrolladas con el lenguaje de programación java, analizando sus características y las ventajas/desventajas que presenta cada uno de ellos.

Para finalizar, se procederá a construir una implementación propia de un framework de persistencia que facilite la construcción de la comunicación entre aplicaciones y la base de datos.

1.2 OBJETIVOS GENERALES Y ESPECÍFICOS

Para el desarrollo de este proyecto se va a emplear la plataforma de programación J2EE, por lo que se profundizara en un conocimiento mas profundo de esta plataforma y de las especificaciones de los APIs que la componen, centrándonos en el API de JDBC en el que nos apoyaremos y extenderemos para llevar a cabo la construcción del framework de persistencia final.

Así mismo, en el desarrollo del PFC se pretende conseguir los siguientes objetivos:

- Definición de framework: Características generales y tipos.

- Persistencia de datos: ¿Qué es una capa de persistencia? Objetivos y mecanismos de persistencia.

- Frameworks de persistencia: Estudio de los frameworks de persistencia existentes en el mercado, estudiando como funcionan y su implementación, así como sus ventajas y desventajas.

- Diseño de un framework de persistencia. A partir del estudio anterior, sacar una serie de conclusiones y diseñar un framework de persistencia que incorpore lo mejor y que incluya alguna mejora que no presente los frameworks estudiados.

- Desarrollo de una aplicación de ejemplo que muestre el funcionamiento del framework implementado.

1.3 PLAN DE TRABAJO CON HITOS Y TEMPORIZACIÓN

La planificación del proyecto se ha hecho teniendo en cuenta las fechas de entrega establecidas para las PECs programadas, estableciendo estas fechas como hitos en los que se deberá de entregar lo previsto en la siguiente planificación:

Tarea	Duración	Fecha Inicio	Fecha Fin
Plan de trabajo	12d	15/09/2008	30/09/2008
Datos Generales frameworks	5d	02/10/2008	08/10/2008
Analisis de la persistencia de datos	6d	09/10/2008	16/10/2008
Evaluación de frameworks de persistencia	12d	17/10/2008	03/11/2008
Entrega PEC2	1d	05/11/2008	05/11/2008
Analisis y Diseño del framework	16d	31/10/2008	21/11/2008
Implementacion del framework	20d	24/11/2008	19/12/2008
Entrega PEC3	1d	17/12/2008	17/12/2008
Aplicación de ejemplo y pruebas	8d	22/12/2008	31/12/2008
Memoria	6d	02/01/2009	09/01/2009
Elaboracion presentación	2d	12/01/2009	13/01/2009
Entrega Final	1d	14/01/2009	14/01/2009

Los hitos principales son los siguientes:

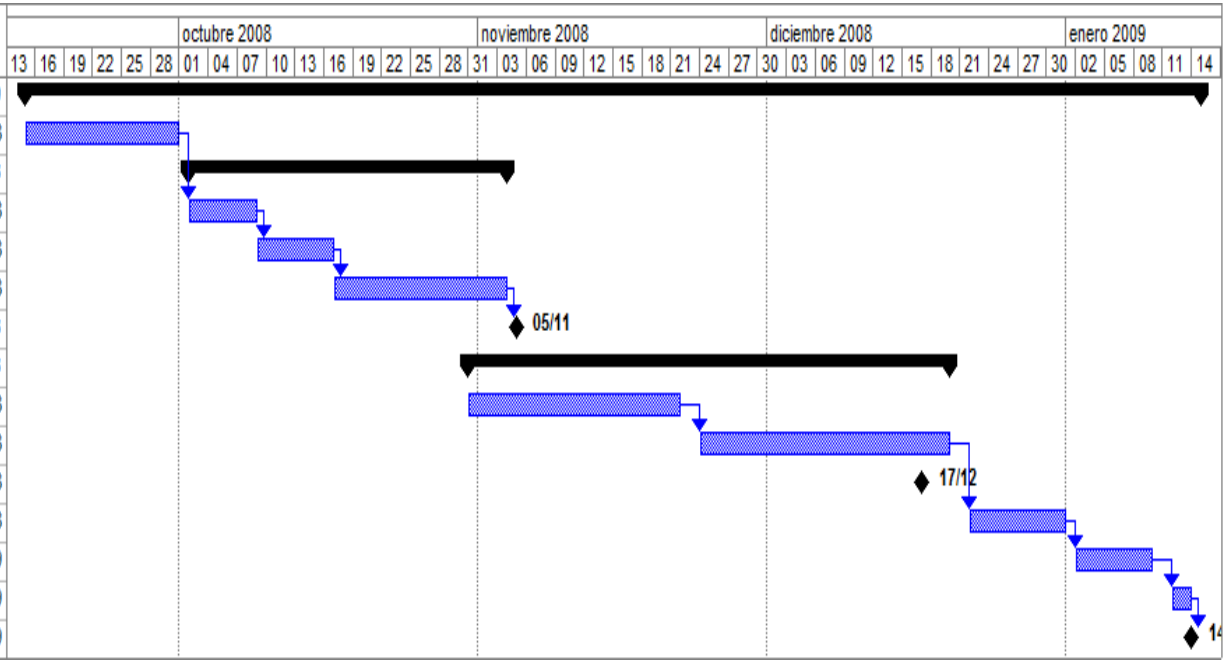
PEC2: 5/11/2008

PEC3: 17/12/2008

Entrega final: 14/01/2008

1.4 DIAGRAMA DE GANTT

Id	Nombre de tarea	Duración	Comienzo	Fin
1	**Proyecto Fin de Carrera**	**88 días**	**lun 15/09/08**	**mié 14/01/09**
2	Plan de trabajo	12 días	lun 15/09/08	mar 30/09/08
3	**Funcionalidades y analisis de frame**	**23 días**	**jue 02/10/08**	**lun 03/11/08**
4	Datos Generales frameworks	5 días	jue 02/10/08	mié 08/10/08
5	Analisis de la persistencia de datos	6 días	jue 09/10/08	jue 16/10/08
6	Evaluación de frameworks de pers	12 días	vie 17/10/08	lun 03/11/08
7	**Entrega PEC2**	**1 día**	**mié 05/11/08**	**mié 05/11/08**
8	**Diseño del framework**	**36 días**	**vie 31/10/08**	**vie 19/12/08**
9	Analisis y Diseño del framework	16 días	vie 31/10/08	vie 21/11/08
10	Implementacion del framework	20 días	lun 24/11/08	vie 19/12/08
11	**Entrega PEC3**	1 día	mié 17/12/08	mié 17/12/08
12	Aplicación de ejemplo y pruebas	8 días	lun 22/12/08	mié 31/12/08
13	Memoria	6 días	vie 02/01/09	vie 09/01/09
14	Elaboracion presentación	2 días	lun 12/01/09	mar 13/01/09
15	**Entrega Final**	1 día	mié 14/01/09	mié 14/01/09



.

2 PERSISTENCIA DE OBJETOS

2.1 INTRODUCCIÓN

Como introducción a este proyecto, a lo largo de este capitulo se presentan una serie de conceptos y términos utilizados a lo largo de este proyecto y que servirán de base para construir el sistema de persistencia. Así mismo, el final de este capitulo nos centraremos en lo relacionado con la persistencia de objetos y las bases de datos relacionales.

2.2 ¿QUE ES UNA CAPA DE PERSISTENCIA?

Las aplicaciones que se desarrollan hoy en día estructuran y guardan la información en sistemas de gestión de datos con el objetivo de que puedan ser reutilizados, bien desde la misma aplicación o desde otro proceso diferente. La capa de persistencia es la pieza que nos permite almacenar, recuperar, actualizar y eliminar el estado de los objetos que pueden ser persistentes en uno o más sistemas gestores de datos.

El sistema gestor de datos puede ser un sistema RDBMS (Relational Database Management System), un sistema OODBMS (Object-oriented database management system) o cualquier otro sistema, de manera que el programador no tenga que hacer ninguna traducción de los objetos para convertirlos en persistentes, sino que sea esta capa la que se encargue de esta transformación.

En el caso de los RDBMS, que son los sistemas gestores de datos mas extendidos, utilizan un modelo teórico llamado relacional y los programas de ordenador suelen utilizar el paradigma de la programación orientada a objetos, que difiere mucho del modelo relacional, se hace necesario un componente de software que permita la comunicación de estos dos instrumentos.

La capa de persistencia traduce entre los dos modelos de datos: de registros a objetos y de objetos a registros. Cuando el programa quiere grabar un objeto llama al motor de persistencia, que traduce el objeto a registros y llama a la base de datos para que guarde estos registros. De la misma manera, cuando el programa quiere recuperar un objeto, la base de datos recupera los registros correspondientes, los cuales son traducidos en formato de objeto por el motor de persistencia. De esta manera el programa sólo ve que puede guardar objetos y recuperar objetos, como si estuviera programado para una base de datos orientada a objetos. La base de datos sólo ve que guarda registros y recupera registros, como si el programa estuviera dirigiéndose a ella de forma relacional.

A esta última técnica de programación se le llama ORM (Object-Relational Mapping). Un ORM es una capa que permite relacionar objetos con un modelo de datos relacional, de modo de ocultar todo el mecanismo de conexión al

motor de base de datos y además no tener que escribir las sentencias SQL necesarias para hacer consultas y/o modificaciones a los registros de la base de datos.

2.3 OBJETIVOS DE UNA CAPA DE PERSISTENCIA

A continuación se detallan los objetivos mas importantes que debe de soportar una capa de persistencia extraídos del trabajo de Scott W. Ambler, "The Design of a Robust Persistence Layer For Relational Databases":

- **Diferentes tipos de mecanismos de persistencia**
  Un mecanismo de persistencia es cualquier tecnología que se puede usar para almacenar permanentemente objetos para su posterior modificación, lectura o borrado en cualquier tipo de base de datos, tales como base de datos relacionales, base de datos orientadas a objetos, bases de datos jerárquicas, xml, etc.

- **Encapsulación total del mecanismo de persistencia**
  Solo necesitaremos enviar al mecanismo de persistencia los mensajes de salvar, borrar, recuperar para salvar, borrar o recuperar un objeto del sistema de persistencia. El se encargara de todo las operaciones necesarias para llevar a cabo estas acciones de forma transparente.

- **Acciones sobre múltiples objetos**
  Es común necesitar recuperar varios objetos a la vez para generar un informe o el resultado de una búsqueda por lo que la capa de persistencia deberá permitir trabajar sobre múltiples objetos. Del mismo modo, se necesitara trabajar sobre múltiples objetos si deseamos borrarlos.

- **Transacciones**
  La capa de persistencia debe de soportar transacciones, es decir, debe de tener la capacidad de realizar distintas operaciones (salvar, recuperar, borrar) sobre un mismo objeto o sobre una lista de objetos de manera que controle que todas las operaciones son correctas o es necesario hacer un roll-back de las mismas dejando al objeto como estaba originalmente antes de comenzar las operaciones.

- **Extensibilidad**
  Al igual que deberíamos de ser capaces de añadir nuevas clases a las aplicaciones, deberíamos de ser capaces de sustituir los mecanismos de persistencia empleados y de poder extenderlo con nuevas características.

- **Identidad de los objetos persistentes**
  La capa de persistencia debe de asociar un identificador a cada objeto persistente, que sirve para localizar, de forma univoca, la imagen del objeto guardado en el sistema de gestión de datos.

- **Cursores**
  La capa de persistencia debe de recuperar el numero de objetos que

se le pidan, ya que el manejo de grandes cantidades de datos puede ser inviable.

- **Múltiples arquitecturas**
  La capa de persistencia debe de poder adaptarse a cualquier tipo de arquitectura existente, sea cliente/servidor o de n capas.

- **Diferentes bases de datos**
  Debe de soportar diferentes tipos de bases de datos, de manera que si se cambia la base de datos, las aplicaciones no se vean afectadas.

- **Múltiples conexiones**
  La capa de persistencia debe de permitir abrir conexiones a diferentes bases de datos dentro de una misma aplicación.

- **Consultas SQL**
  Normalmente las consultas SQL no estarán escritas dentro del código de una aplicación, pero a veces es necesario hacerlo, por lo que la capa de persistencia deberá de soportar el lanzar código SQL directamente desde el código de la aplicación.

2.4 MECANISMOS DE PERSISTENCIA

Toda aplicación que requiera del procesamiento de datos puede ser dividida a grandes rasgos en dos niveles: datos persistentes y el procesamiento de los mismos. Los mecanismos utilizados para acceder e interactuar con los datos son de vital importancia no solo por su impacto en el desempeño final del sistema sino también desde el punto de vista de aspectos tales como mantenibilidad, diseño, reusabilidad y escalabilidad.

La técnica que permite resolver la persistencia de objetos en el desarrollo de un sistema orientado a objetos se considera “Mecanismo de persistencia”.

Existen numerosas alternativas a la hora de seleccionar un mecanismo para acceder a los datos y posteriormente manipularlos, por lo que a continuación se detallan los más destacados.

2.4.1 ACCESO DIRECTO A BASE DE DATOS

El mecanismo implica el uso directo de la base de datos para implementar la persistencia del sistema. Esto último se refiere al acceso a los datos utilizando por ejemplo una interfaz estandarizada en conjunto con algún lenguaje de consulta soportado directamente por el sistema de gestión de base de datos (SGBD). En este caso, no existe ningún tipo de capa entre la aplicación y el SGBD utilizado.

2.4.2 MAPEADORES

Son aquellos mecanismos que se basan en la traducción bidireccional entre los datos encapsulados en los objetos de la lógica de un sistema orientado a objetos y una fuente de datos que maneja un paradigma distinto. El mapeador es el encargado de convertir los datos entre dos paradigmas distintos y hacer un “mapeo” o conversión entre ellos.

2.4.3 GENERADORES DE CÓDIGO

En el desarrollo de aplicaciones empresariales existen tareas repetitivas que el programador se ve obligado a realizar. En este caso, se pueden usar herramientas que generen código correcto y basado en patrones para resolver la persistencia del sistema, permitiendo al desarrollador centrarse en el código que resuelve la lógica de negocio.

2.4.4 ORIENTACIÓN A ASPECTOS

La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden encapsular los diferentes conceptos

que componen una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables

El objetivo de la POA es la separación de las funcionalidades comunes utilizadas a lo largo de la aplicación y las funcionalidades propias de cada módulo. Dentro de estos módulos se contempla la persistencia de la aplicación para su modularización y reutilización del código.

2.4.5 LENGUAJES ORIENTADOS A OBJETOS

Se trata de resolver la persistencia de datos utilizando funcionalidades provistas por el propio lenguaje de programación evitando la inclusión de frameworks u otras herramientas ajenas al lenguaje y persistiendo los datos usando ficheros de texto plano o ficheros XML usando técnicas de serialización.

2.4.6 ESQUEMAS DE PREVALENCIA

Son marcos de trabajo que mantienen los objetos en memoria volátil y periódicamente utilizan esquemas de serialización de objetos para guardar una imagen (snapshot) de los objetos de la aplicación. Es un acercamiento muy simple y rápido a la persistencia aunque presenta algunos problemas de escalabilidad.

Se trata de una propuesta diferente a utilizar bases de datos para lograr la persistencia.

2.5 EQUIVALENCIA CLASE-TABLAS RELACIONALES

En este último apartado vamos a centrarnos en el modelo relacional, que es el más extendido hoy en día, y de cómo persistir los objetos en este tipo de modelos. Esto se debe a que el framework a desarrollar se basará en bases de datos relacionales (mySQL, Oracle, etc) y los motores de persistencia más conocidos (iBATIS, Hibernate, etc) funcionan con estos modelos.

El problema esta en cómo hacer corresponder objetos en relaciones y viceversa, y de cómo dar solución al problema de la correspondencia clase-tabla.

Clase y tabla son conceptos diferentes, utilizados para modelar las realidades que después manejan las aplicaciones. Una clase define los atributos y el comportamiento de la realidad que modela, define objetos. Una tabla especifica los datos de la entidad que representa, concreta una relación entre valores de los dominios de los datos escogidos. Ambos capturan características de interés para la aplicación, pero la tabla ignora aquellas características ligadas al comportamiento en ejecución, cosa que si modela la clase. Para la clase, la persistencia es una característica más, posiblemente independiente, en cambio, la tabla toma su sentido esencialmente de servir al propósito de persistir compartir los datos que modela. No hay por tanto, una equivalencia exacta y única entre ambos conceptos.

A continuación se muestra una primera aproximación de la correspondencia entre clases y tablas

Modelo de objetos	Modelo Relacional
Identidad	Clave primaria
Clase	Relación (Tabla, Vista)
Instancia (Objeto)	Fila
Atributo de instancia persistente con dominio un tipo básico o primitivo	Columna
Atributo persistente referencia a objetos con tipo conocido	Clave Ajena
Atributo persistente tipo colección	No hay equivalencia directa, debe ser una relación, si la cardinalidad es (m,n), entonces una entidad de asociación. Si es (1,n) entonces atributos en las relaciones

	correspondientes
Atributo persistente de Clase	Una tabla con las características comunes a todas las instancias de una clase
Herencia	Una relación agregada con todos los atributos de instancia de la jerarquía herencia completa o múltiples tablas.

3 FRAMEWORKS DE PERSISTENCIA

3.1 ¿QUE ES UN FRAMEWORK?

Un framework representa una Arquitectura de Software que modela las relaciones que existe entre las entidades del dominio y al mismo tiempo que proporciona una metodología y una estructura de trabajo que se puede extender a las aplicaciones del dominio que en algunos casos puedan utilizarlos.

Un framework de aplicaciones es un término utilizado para referirse a un conjunto de bibliotecas o clases utilizadas para implementar la estructura estándar de una aplicación. En otras palabras, es un esquema o patrón para el desarrollo o implementación de una aplicación completa, o de una parte especifica de una aplicación.

A primera vista puede parecer que de esta definición un framework es una simple librería de clases. Existen varias diferencias entre una librería de clases y un framework que se exponen en la siguiente tabla

Característica	Frameworks	Librerías de Clases
Modelo de la Aplicación	Los frameworks son la aplicación, de forma que manipulan el flujo de control	El usuario ha de crear el control de la aplicación y la lógica para invocar los componentes
Estructura de la Aplicación	Multiples frameworks cooperantes	Una sola aplicación monolítica consistente en un conjunto de librerías de clases
Obtención de Servicios	Los frameworks son el Servicio	Heredando funciones de las librerías de clases
Creación del Sistema	Los frameworks invocan el código del usuario. Este código puede derivar partes del framework	Derivando o creando nuevas clases
Granularidad del Control	Media. El usuario solo puede derivar algunas partes del framework	Alta. El usuario puede derivar cualquier clase de la librería
Abstracción de los Servicios	Alta. Ocultan su complejidad. Automatizan las características estándar y ofrecen un mecanismo de excepciones	El usuario ha de determinar qué métodos ofrece la librería y en qué orden deben ser invocados
Cantidad de nuevo código a implementar	Muy poco	Medio

Coste de mantenimiento	Bajo	Medio
Reducción Complejidad	Mucha. El usuario escribe pequeñas porciones de código	Media.El usuario tiene que crear nuevas clases para cada una de las aplicaciones que desarrolle y desarrollar el programa
Tiempo necesario para desarrollar una aplicación	Bajo	Medio. Depende del grado de reutilización que se pueda hacer de la librería
Reutilización de Código	Muy alto	Alto

Las **ventajas** de emplear un framework son las siguientes:

- **Infraestructura prefabricada**

  Los frameworks reducen la codificación y sobretodo la puesta en marcha, ya que proporcionan subsistemas que sabemos que ya funcionan. En definitiva, proporcionan código que no se tendrá que mantener ni reescribir.

- **Proporcionan una arquitectura**

  Cualquier programador que trabaje con un framework no deberá invertir una gran parte de su tiempo en buscar las clases necesarias, interconectarlas o descubrir los métodos que contienen. Los frameworks ocultan toda esta complejidad dando un nivel alto de abstracción.

- **Reducción del mantenimiento**

- **Buena base para la industria de componentes**

  Los frameworks bien diseñados, permiten que terceras compañías puedan suministrar componentes o partes de componentes que los usuarios finales podrán añadir

Las **desventajas** son:

- **Limitación de la Flexibilidad**

  Los componentes que un usuario construya a partir de un framework deben residir dentro de las restricciones impuestas por la arquitectura del framework.

- **Dificultad de Aprendizaje**

El usuario debe estudiar qué proporciona el framework y cómo debe hacer uso de él. Este aprendizaje (requerido una sola vez) es costoso en el inicio, por partida, el desarrollo será más rápido.

- **Reducción de la Creatividad**

  Puede suceder que el usuario no pueda cambiar el comportamiento de una clase del framework si ésta no permite que sus servicios puedan reescribirse o redefinirse.

3.2 FRAMEWORKS DE PERSISTENCIA

Los frameworks de persistencia introducen todas las ventajas comentadas en el apartado anterior aplicadas a la comunicación de las aplicaciones y el sistema de persistencia. En el caso del lenguaje de programación Java, la persistencia de datos en sistemas relacionales viene facilitada por los mapeadores Objeto/Relacionales (ORM), que permiten enlazar un lenguaje orientado a objeto con una base de datos relacional.

Estos frameworks permiten a los desarrolladores mantener una perspectiva orientada a objetos y centrarse en la programación de la lógica de negocio, encargándose de automatizar las tareas de conectarse a la base de datos y guardar/borrar/recuperar la información de la base de datos y convertirla en objetos, con independencia de la base de datos que estemos utilizando.

En este apartado nos vamos a centrar en las alternativas de frameworks de persistencia para el lenguaje de programación java que más se usan en el desarrollo de aplicaciones informáticas que están basados en JDBC, y las características más notables de los mismos.

3.2.1 IBATIS

iBATIS SQL Maps es un framework open-source sobre JDBC que provee un medio simple y flexible de mover datos entre un objeto java y una base de datos relacional. Esta plataforma ayuda a reducir la cantidad de código Java (JDBC) que es necesario para acceder a una base de datos relacional. El framework permite relacionar un java Bean a una sentencia SQL mediante el uso de archivos xml que permiten crear consultas complejas, ya que es en este archivo xml donde se escriben las sentencias SQL de acuerdo al sistema de gestión de bases de datos. Por tanto, si se cambia de base de datos, será necesario reescribir las consultas, ya que iBATIS no es independiente del proveedor de BD.

Por esta última razón, iBATIS no es un ORM puro, ya que se tienen que escribir las sentencias SQL necesarias para interactuar con la base de datos.

De esta forma, iBATIS permite escribir manualmente los comandos SQL y relacionarlos a un objeto Java. Una vez que uno relaciona la capa de persistencia a un modelo de objeto, entonces ya se puede usar dentro de cualquier aplicación java. Ya no es necesario buscar fuentes de datos, obtener conexiones, crear PreparedStatements, interpretar el ResultSet, o guardar los datos en memoria, pues iBATIS hace todo eso por uno.

3.2.2 HIBERNATE

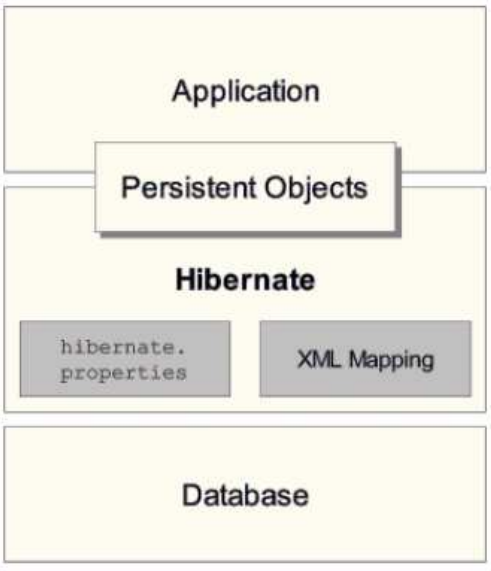
Hibernate es una herramienta ORM para Java. Hibernate permite al desarrollador hacer persistentes objetos, siguiendo las propiedades del lenguaje Java tales como asociación, herencia, polimorfismo, composición y todo lo que son Java Collections.

Hibernate no solo provee relaciones entre clases Java y tablas de base de datos (y de tipos de datos Java a tipos de datos SQL), sino también provee recuperación de datos a través de un mecanismo propietario de consulta llamado HQL (Hibernate Query Language). Todo esto permite reducir los tiempos de desarrollo de manera drástica evitando el manejo manual de SQL y JDBC.

El objetivo final de Hibernate es poder permitir al desarrollador reducir hasta en un 95% las tareas comunes de persistencia que perfectamente se pueden automatizar.

Hibernate existe bajo licencia LGPL la cual es suficientemente flexible para permitir usar esta herramienta en proyectos comerciales como en proyectos de código abierto.

La arquitectura de alto nivel de Hibernate se describe en la siguiente figura.



Hibernate hace uso de clases POJO (Plain Old Java Object) java en conjunto con ficheros xml que permiten relacionar estos objetos con la capa de base de datos. En vez de utilizar procesamiento de byte code o generación de código, Hibernate usa en tiempo de ejecución una característica de Java denominada reflexión, la cual permite hacer instrospección de un objeto en tiempo de ejecución.

Hibernate no está restringido en el uso de ningún tipo de datos Java ni objetos ni primitivos.Todos estos tipos pueden relacionarse, incluso clases que pertenecen a la plataforma de Java Collections. Estas se pueden asociar como valores o asociaciones a otras entidades en base de datos. En Hibernate existe una propiedad especial que es el campo id el cual representa el identificador o llave primaria de la clase.

Los objetos que se desean hacer persistentes se definen en un documento de mapping (el cual es un archivo xml), y que permite describir los campos persistentes con sus respectivas asociaciones, así como también las subclases que posee el objeto en cuestión. Estos documentos son compilados en el momento de inicialización de la aplicación y proveen a la herramienta de la información necesaria para cada clase.

Adicionalmente, Hibernate es capaz de generar los esquemas de base de datos a partir del modelo de clases e incluso crear clases Java a partir del modelo de datos.

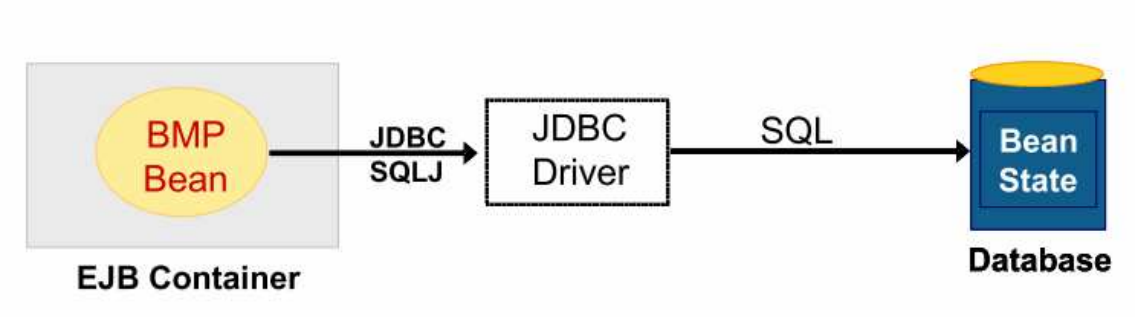
3.2.3 EJB DE ENTIDAD

Los EJB de entidad son componentes que representan los datos persistentes del modelo de negocio de la aplicación. Se trata de una representación Java, en memoria y en forma de objeto de la información almacenada en algún medio persistente.

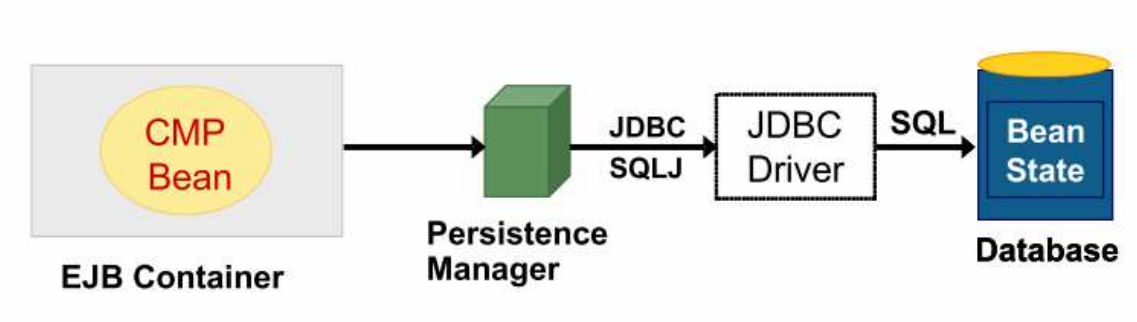
Dentro de los EJB de entidad, hay dos maneras diferentes de gestionar la persistencia: si dejamos que sea el desarrollador el que especifique el comportamiento persistente, estaremos hablando de **EJB de entidad con persistencia gestionada por el componente (BMP, Bean Managed Persistence)**; si es el contenedor el que gestiona la persistencia, estaremos hablando de **EJB de entidad con persistencia gestionada por el contenedor (CMP, Container Managed Persistence)**.

En los EJB de entidad con **persistencia gestionada por el componente (BMP)**, el desarrollador del componente es el responsable de programar la persistencia del componente. Esto implica que el desarrollador del componente tiene que codificar el código que permita mantener sincronizados los datos del EJB y los datos de la fuente de datos, por ejemplo codificando con JDBC la SELECT que permita cargar los datos de la base de datos en los atributos del EJB de entidad. Desde el punto de vista del desarrollador, la persistencia gestionada por el componente requiere más esfuerzo que la persistencia

gestionada por el contenedor, porque se tiene que escribir explícitamente la lógica de la persistencia en la clase EJB



En los EJB de entidad con **persistencia gestionada por el contenedor (CMP)**, el desarrollador no tiene que programar la persistencia del componente. El contenedor implementa la gestión de la persistencia según las definiciones que proporciona el desarrollador al descriptor de despliegue del componente.



El desarrollador del componente tiene que definir los campos y las relaciones persistentes del componente que se quiere desarrollar, y el contenedor se encarga de generar el código de gestión de la persistencia según esta definición cuidadosa en tiempo de despliegue del componente.

Los EJB de entidad tienen métodos finder y select que sirven para hacer búsquedas de éstos. Si el EJB utiliza persistencia gestionada por el componente, estos métodos de busca se programan con código JDBC estándar, pero si utiliza persistencia gestionada por el contenedor se definen de manera declarativa en el descriptor de despliegue.

Para definir de manera declarativa el comportamiento de los EJB de entidad con persistencia gestionada por el contenedor, la especificación de EJB define un lenguaje muy similar a SQL, denominado EJB QL.

El EJB QL es muy similar al SQL, pero se escribe como XML en los descriptores de despliegue y permite a los programadores describir el comportamiento de métodos de consulta de una manera abstracta, haciendo estas consultas transportables a bases de datos de fabricantes distintos

3.2.4 JAVA DATA OBJECTS (JDO)

Desde mayo de 2002 ha sido propuesto como nuevo estándar de persistencia para objetos Java la especificación Java Data Objects (JDO).

JDO es un estándar de persistencia de objetos diseñado por Sun; el mismo proporciona un conjunto de librerías que permite al programador liberarse de escribir procedimientos para persistir los objetos de la aplicación que esta desarrollando.

El API Java Data Objects (JDO) proporciona una forma estándar y sencilla de conseguir la persistencia de objetos en la tecnología Java. JDO utiliza una combinación práctica de metadatos XML y bytecodes mejorados para hacer más sencilla la complejidad y la sobrecarga, comparado con otras tecnologías de unión de objetos.

Una razón de la simplicidad de JDO es que permite trabajar con objetos normales de Java (POJO´s) en lugar de con APIs propietarios. JDO corrige el aumento de la persistencia en un paso de mejora de bytecodes posterior a la compilación, así proporciona una capa de abstracción entre el código de la aplicación y el motor de persistencia.

JDO ha recibido ataques por muchas razones, entre las que se incluyen: el modelo de mejora de código debería reemplazarse por un modelo de persistencia transparente; los vendedores podrían perder sus bloqueos propietarios; JDO se solapa demasiado con EJB y CMP. A pesar de estos puntos de resistencia, muchos expertos están tutorizando JDO como una enorme mejora sobre las tendencias actuales de tecnologías objeto a datos y datos a objeto.

El API JDO, consta sólo de unos pocos interfaces, es uno de los APIs más fáciles de aprender de toda la tecnología existente actualmente para la persistencia de objetos estandarizada y existen muchas implementaciones de JDO entre las que elegir.

4 DISEÑO DE UN FRAMEWORK DE PERSISTENCIA

4.1 INTRODUCCIÓN

Del estudio de los frameworks de persistencia del capitulo anterior se han sacado una serie de conclusiones a partir de las cuales modelar una nueva implementación que nos permita modelar el acceso a base de datos en aplicaciones.

4.2 CARACTERÍSTICAS DEL FRAMEWORK

Una característica común a todos los frameworks u opciones de persistencia vistas en el apartado anterior es que requieren de un gran esfuerzo para comprender su funcionamiento y configuración, apoyándose en complicados ficheros xml que el desarrollador tiene que comprender para implementar el acceso a los datos. Esto, para proyectos de media/gran envergadura es asumible, ya que estos frameworks, además de abstraer la capa de acceso a datos, incorporan muchas mas características (caches, pool de conexiones, control de transacciones, etc.) que son de utilidad en este tipo de desarrollos.

Por tanto, se va a implementar un framework de pequeño tamaño, de configuración muy básica y que cubra las funcionalidades mínimas necesarias para interactuar con bases de datos relacionales. Los requerimientos a cumplir son los siguientes:

- Se podrá utilizar en cualquier tipo de aplicación, tanto aplicaciones stand-alone como aplicaciones servidor.
- Tamaño reducido. Las clases necesarias para resolver la persistencia.
- Independiente de la base de datos. Se podrá usar con cualquier base de datos para la que exista driver JDBC. Solo se necesitará pasarle una conexión con la base de datos para realizar las operaciones con el motor.
- Fácil configuración y puesta en marcha. Reducir el número de ficheros de configuración al mínimo posible.
- Generación de consultas SQL. El framework será el encargado de generar automáticamente las consultas básicas de selección, actualización, inserción y borrado que se hagan contra una tabla del

modelo.

- Proporcionara una forma fácil de realizar operaciones CRUD (Create, Read, Update y Delete) y búsquedas definidas por el usuario utilizando clases java POJO (Plain Old Java Object))

Para realizar las conversiones necesarias entre el modelo de objetos y el modelo relacional, el framework a implementar va a utilizar las **anotaciones Java** para que el framework sea capaz de hacer estas conversiones.

Las anotaciones Java se han introducido en la versión de Java 1.5 y son un mecanismo con el cual se puede dotar a las clases de meta-información o auto-información que se puede usar tanto en tiempo de compilación como de ejecución. La forma de hacer uso de las anotaciones en tiempo de ejecución es mediante el API de **reflection,** que permite acceder y manejar la información contenida dentro de una clase, sus atributos, metodos, etc.

4.3 ANÁLISIS Y DISEÑO DEL FRAMEWORK

A continuación vamos a hacer un estudio de las diferentes clases que componente el framework atendiendo al tipo de funcionalidad que resuelven.

4.3.1 DIAGRAMA DE PAQUETES

Todas las clases agrupadas en un paquete tienen alguna característica en común (estructura, funcionalidad, comportamiento, objetivo etc.). La estructura de paquetes de las clases del framework es la siguiente:

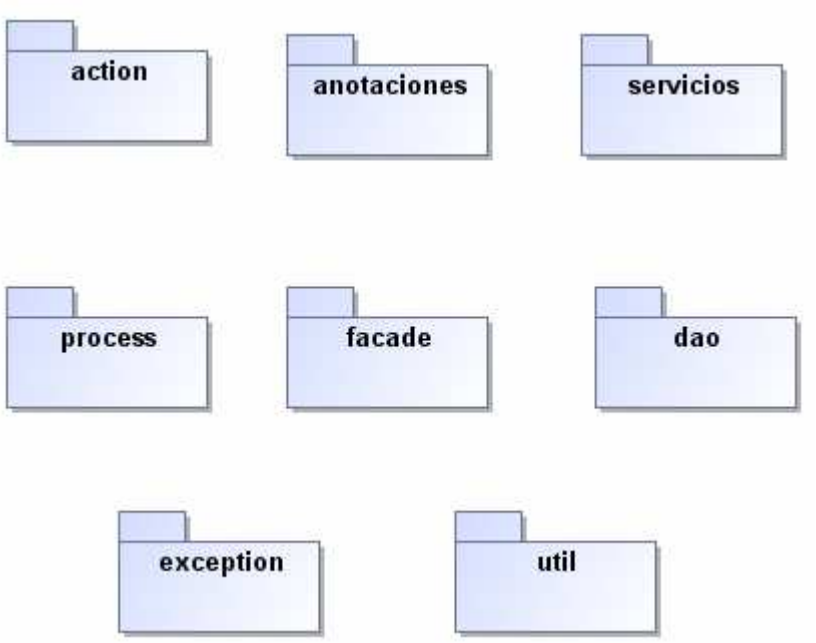


Diagrama de paquetes del framework (edu.uoc.pfc)

- *edu.uoc.pfc.anotaciones*: contiene las anotaciones que nos van a servir para añadir metainformacion a los POJOS y poder identificarlos con tablas del modelo de datos

- *edu.uoc.pfc.servicios:* clases que gestionan la lectura de la información de las anotaciones definidas sobre los objetos, asi como las clases que generan automáticamente las consultas a partir de los mismos.

- *edu.uoc.pfc.dao*: clases que realizan la conexión con la base de datos y ejecutan las consultas haciendo uso de las clases del paquete anterior.

- *edu.uoc.pfc.action*: clases utilizadas por la fachada para procesar sus funciones.

- *edu.uoc.pfc.action.process*: gestionan la ejecución de los action

- *edu.uoc.pfc.facade*: clases de fachada de servicios a través de los cuales se accede a las operaciones del framework

- *edu.uoc.pfc.exception:* jerarquia de excepciones manejadas por el framework para la gestión de errores

- *edu.uoc.pfc.util:* clases de utilidad

4.3.2 MODELADO DE LA BASE DE DATOS

Como hemos visto en el apartado 2.5 del capitulo 2, podemos establecer una correspondencia entre el modelo relacional (tablas, relaciones, etc.) y el modelo de objetos. Para ello, y con ayuda de las anotaciones Java vamos a construir una serie de anotaciones que van a añadir información adicional a las clases y que nos permitira identificar los mapeos desde y hacia la fuente de datos relacional. Es la misma técnica que usa la última versión de Hibernate y que viene definido en la especificación de JPA (Java Persistence API).

Las anotaciones a definir en este framework son las básicas que permiten realizar las operaciones más comunes contra bases de datos.

Las clases a las que añadiremos esta Meta información o metadata serán del tipo POJO, es decir, clases que implementan la interfaz Serializable y con una serie de atributos con sus correspondientes getters y setters, que nos identificarán una tabla de la base de datos y sus relaciones. Por tanto tendremos las siguientes anotaciones que aplicadas sobre los POJOS nos ayudarán a identificar la correspondencia entre las clases con las tablas del modelo de base de datos:

Clase Table

Con esta anotación aplicada sobre una clase indicaremos que la clase corresponde a una tabla de la base de datos. Tiene dos atributos:

- **tableName**: Cadena en la que indicaremos el nombre de la tabla a la que corresponde la clase. Si no se indica, se coge el nombre de la clase como nombre de la tabla.
- **generatedValue**: Indicador de si la tabla tiene clave primaria auto incremental o es parte de una secuencia. Nos sirve para saber que hacer cuando se hace una operación de inserción en la base de datos, ya que sabremos si para insertar en la base de datos tenemos que indicar la clave primaria o es el motor de la base de datos quien la genera.

Clase NoTable

Con esta anotación aplicada sobre una clase indicaremos que la clase no corresponde a una tabla de la base de datos, sino que el objeto se usa para recuperar datos de la base de datos mediante una consulta del tipo SELECT. Esto nos sirve para si queremos hacer nuestra propia consulta, los campos que se recuperen se mapeen a los campos del objeto anotado. Se utiliza junto con la anotación Column.

Clase Column

Esta anotación se aplica sobre los atributos de una clase y nos indica que el atributo corresponde a una columna de una tabla de la base de datos y si el atributo forma parte de la clave primaria. Tiene dos atributos:

- **nombre:** Cadena en la que indicaremos el nombre de la columna a la que corresponde el atributo. Si no se indica, se coge el nombre del atributo como nombre de la columna de la tabla.

- **primaryKey:** Indicador de si el atributo es parte de la clave primaria de la tabla. Por defecto no pertenecen a la clave primaria.

- **insertable:** Indica que la columna se tiene que meter en la generación de una sentencia SQL INSERT. Por defecto se ponen todas las columnas. Si queremos que no se incluya en estas sentencias, poner esta propiedad a false (Indicaría que el campo se quede null en el insert)

- **updatable:** Indica que la columna se tiene que meter en la generación de una sentencia SQL UPDATE. Por defecto se ponen todas las columnas. Si queremos que no se incluya en estas sentencias, poner esta propiedad a false

Esta anotación se aplica sobre atributos cuyo tipo de dato se puede convertir al tipo de dato de la base de datos y admite los siguientes tipos de

datos, que son los que maneja la clase PreparedStatement del paquete java.sql:

String, Integer, Long, Double, Float, Calendar, Date, Bolean, Byte, Short, Carácter y BigDecimal

Con estas dos clases, ya tendríamos hecha parte de la correspondencia tabla-clase que habíamos visto en el capitulo 2, pero todavía nos falta por modelar las relaciones que tienen las tablas con otras tablas y las acciones a realizar en la entidad relacionada cuando se realice una operación sobre la entidad principal, es decir, si por ejemplo hacemos un operación delete sobre una entidad, borrar también las entidades relacionadas.

Clase OneToOne

Esta anotación se aplica sobre los atributos de una clase y nos indica que el atributo es clave ajena o foreign key de otra tabla del modelo y la relación es 1 a 1. Esta anotación tiene usarse junto con la anotación Column, ya que es en el atributo nombre donde indicaremos el nombre de la columna que es clave ajena en la tabla relacionada.

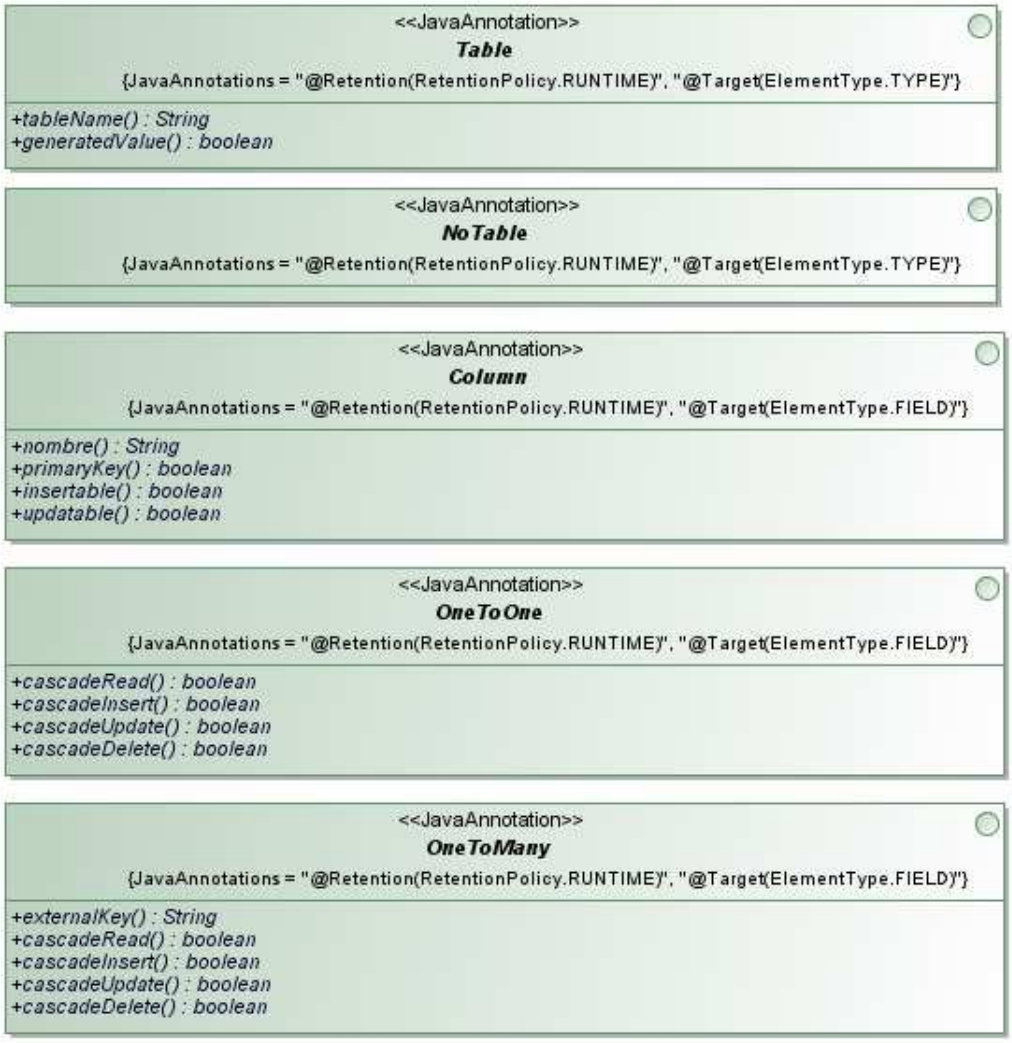
Tiene cuatro atributos booléanos para indicar que hacer en la entidad asociada cuando se hacen operaciones sobre la principal. Los atributos son: cascadeRead cascadeInsert cascadeUpdate y cascadeDelete

Clase OneToMany

Esta anotación se aplica sobre los atributos de una clase que son del tipo Collection y nos indica que el atributo es clave ajena o foreign key de otra tabla del modelo y la relación existente es 1 a N. Mediante el atributo externalKey indicamos los nombres de las columnas que forman parte de la clave ajena en la tabla relacionada.

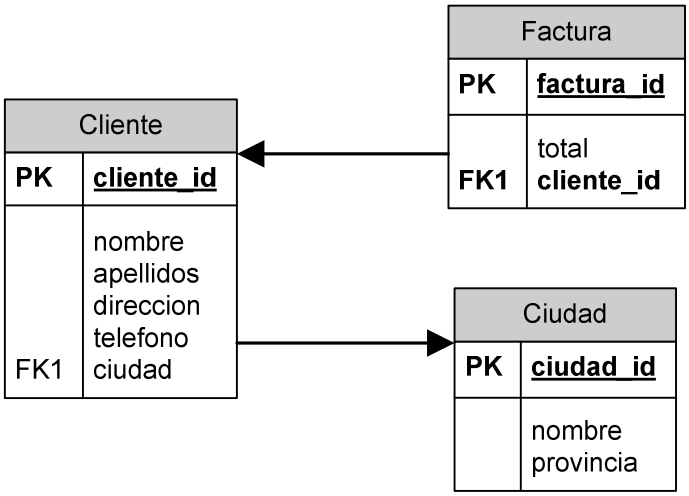
Tiene cuatro atributos boléanos para indicar que hacer en la entidad asociada cuando se hacen operaciones sobre la principal. Los atributos son: cascadeRead cascadeInsert cascadeUpdate y cascadeDelete.

El diagrama de clases correspondiente a esta parte es el siguiente:



Un ejemplo sencillo de cómo usar las anotaciones definidas anteriormente es el siguiente:

Modelo E/R



Lo podemos definir mediante las siguientes clases POJO anotadas:

Tabla Cliente

```
@Table(tableName="CLIENTE", generatedValue=true)
public class Customer implements Serializable {

     @Column(primaryKey=true, nombre="CLIENTE_ID")
    private Integer cliente_Id;

     @Column(nombre = "NOMBRE")
    private String nombre;

     @Column(nombre = "APELLIDOS")
    private String apellidos;

     @Column(nombre = "DIRECCION")
    private String direccion;

     @Column(nombre = "TELEFONO")
    private String telefono;

     @Column (nombre = "CIUDAD_ID")
     @OneToOne
    private Ciudad ciudad;

    @OneToMany(externalKey="CUSTOMER_ID")
    private Collection<Factura> facturas;
```

…

// Métodos Get y Set de los atributos

…

Tabla Factura

```
@Table(tableName="FACTURA", generatedValue=true)
public class Customer implements Serializable {
     @Column(primaryKey=true, nombre="FACTURA_ID")
    private Integer factura_Id;

     @Column(nombre = "TOTAL")
    private BigDecimal total;

     @Column(nombre="CLIENTE_ID")
    private Integer cliente_Id
```

…

// Métodos Get y Set de los atributos

…

Tabla Ciudad

```
@Table(tableName="CIUDAD")
public class Customer implements Serializable {
     @Column(primaryKey=true, nombre="CIUDAD_ID")
    private String ciudad_Id;

     @Column(nombre = "NOMBRE")
    private String nombre;

     @Column(nombre="PROVINCIA")
    private String provincia;
```

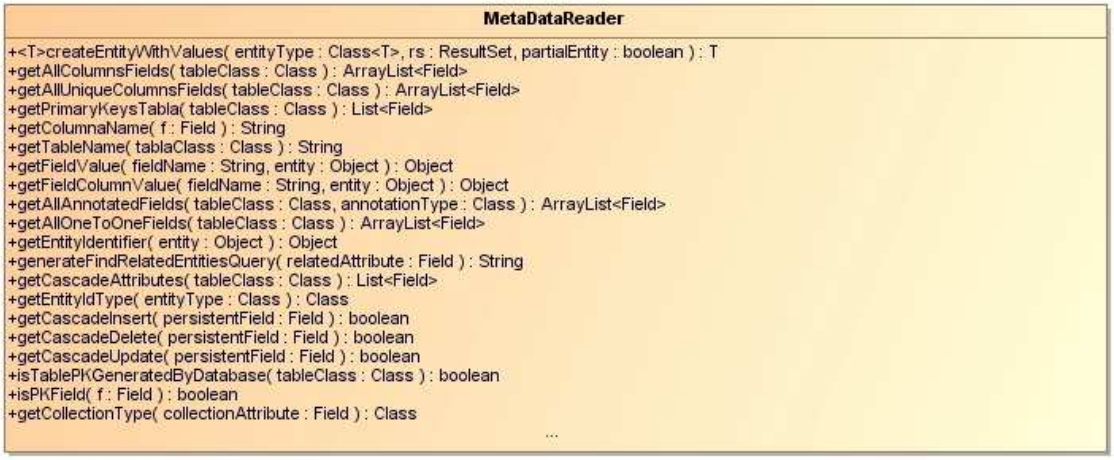
…

// Métodos Get y Set de los atributos

4.3.3 LECTURA DE METADATOS

Una vez definidas las clases que nos permitirán añadir meta información, hemos de crear una clase con las operaciones que permitan al framework leer y procesar esta información para realizar las acciones necesarias. Esta clase será la encargada de decirnos si una clase tiene definida la anotación Tabla, como se llama la tabla, identificar los campos que son clave primaria, las entidades que están relacionadas, etc. Esta clase hace un uso intensivo del API de reflection para obtener toda la información.

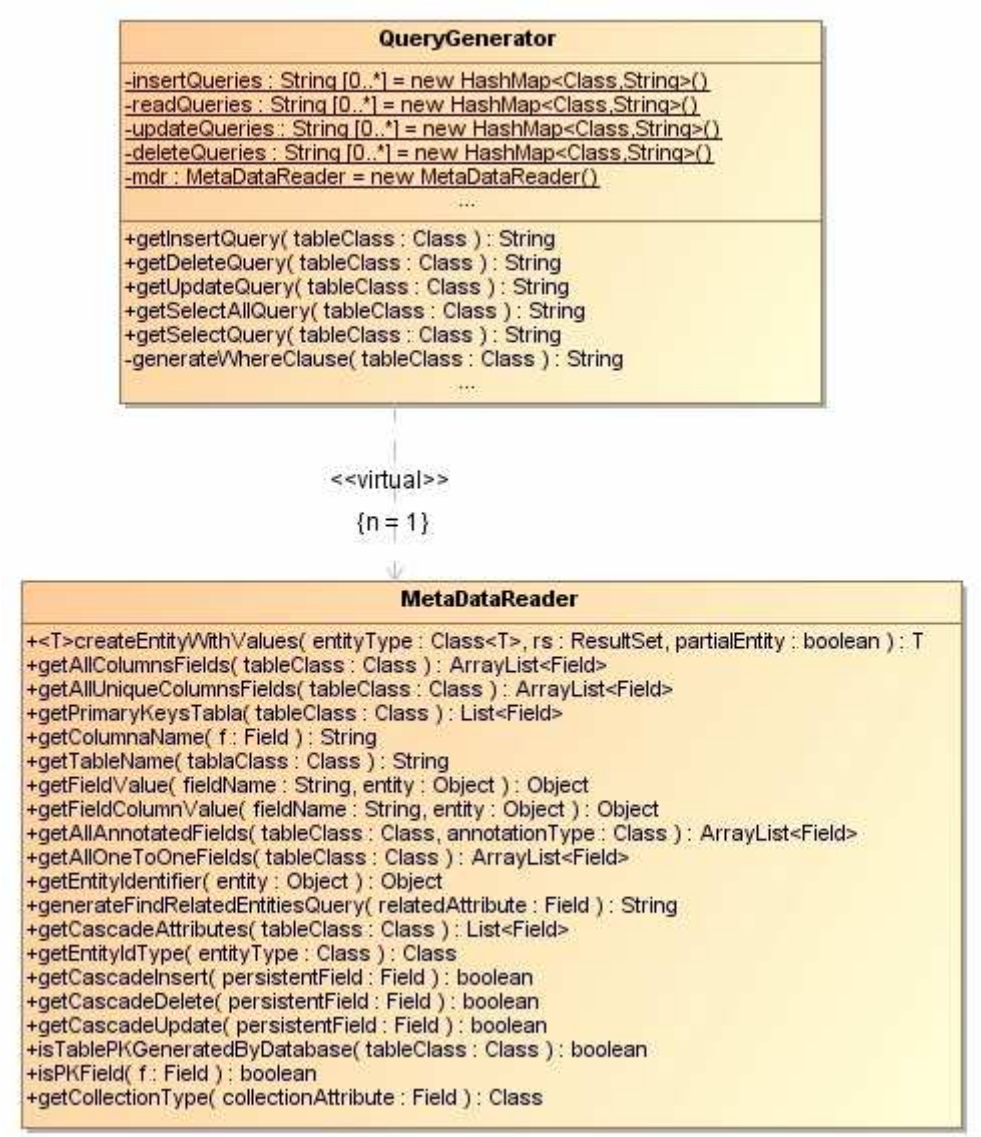
La clase involucrada en este proceso es **MetaDataReader** y su diagrama de clases es el siguiente:



Dispone de todos los métodos necesarios para buscar el las clases anotadas la información existente en ellas y que nos va a guiar para conseguir la persistencia de las mismas.

4.3.4 GENERADOR DE CONSULTAS

Para la generación de consultas vamos a definir una clase que con la ayuda de la anterior sea capaz de generar las consultas contra las tablas de la base de datos. Esta clase es **QueryGenerator** y además de generar las consultas de selección, inserción, actualización y borrado tendrá definidas unas caches donde se irán almacenando las consultas que se van ejecutando sobre los objetos persistentes. Esto nos permite acelerar el proceso de generación de consultas, ya que solo se generarán la primera vez que utilicemos la consulta

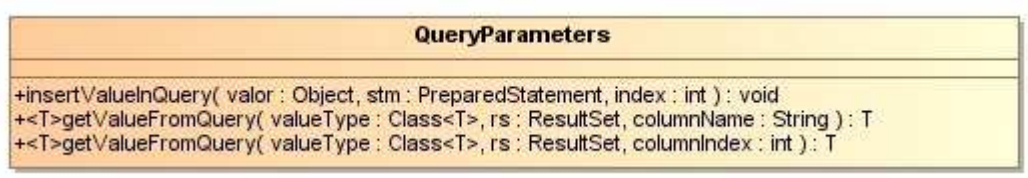


4.3.5 RECUPERACIÓN E INSERCIÓN DE DATOS EN LAS CONSULTAS

El framework se basa en la utilización de clases que representan el modelo de la base de datos. Por lo tanto no conocemos a priori las clases con las que trabajara el framework, por lo que tendremos que hacer un uso intensivo de los tipos genéricos de datos introducidos en la última versión de java (java 1.5) para poder operar con las clases del modelo de una forma genérica, es decir, no nos tiene que importar el contenido de la clase. Lo único que tiene que cumplir es que sea un POJO con sus atributos y anotaciones, y sus correspondientes métodos getters y setters.

La clase que tiene la responsabilidad de introducir los parámetros en las querys y de recuperarlos a partir de un POJO es la clase QueryParameters,

que leyendo un objeto, conocerá el tipo de dato (Integer, String, etc) que tiene que insertar en un statement, y lo mismo cuando tenga que copiar el resultado de una consulta a los atributos de un objeto, donde se tiene que convertir el tipo de dato que retorna la base de datos al tipo de dato Java correspondiente.



Los tipos de datos java que se van a utilizar son: String, Integer, Long, Double, Float, Calendar, Boolean, Byte, Short, Character y BigDecimal, aunque la clase esta preparada para ampliar los tipos de datos si fuera necesario.

4.3.6 ARQUITECTURA DEL FRAMEWORK

Una vez que tenemos definidas las clases con las que vamos a interactuar con el modelo de base de datos, vamos a definir la arquitectura del framework, a través de la cual podremos utilizar las funcionalidades que nos ofrece.

Para implementar la arquitectura del framework nos vamos a apoyar en el uso de los siguientes patrones de diseño:

- **Facade**

Utilizamos este patrón a través del cual publicaremos los métodos que nos va a ofrecer el framework.

- **BusinessDelegate**

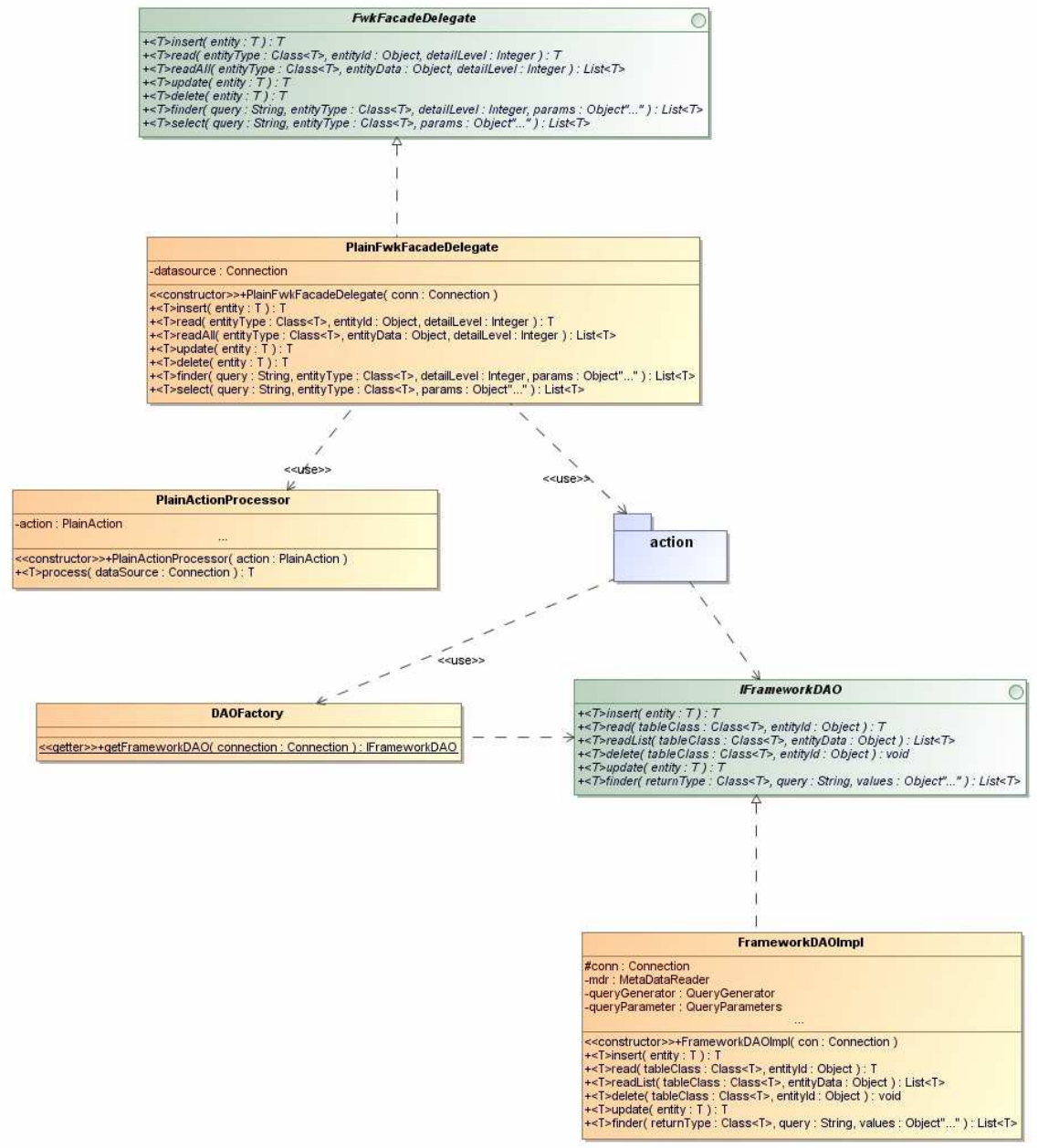
Utilizamos este patrón para delegar la ejecución de las funcionalidades ofrecidas por el framework en los action correspondiente, que a su vez y con llamadas a los DAO serán resuelvan la lógica de la operación correspondiente.

Las clases XXXAction serán las encargadas de establecer la conexión con la base de datos y pueden ser transaccionales o no transaccionales, según el tipo de operación que tenga que realizar.

- **DAO (Data Access Object)**

Existe un único DAO que tendrá los métodos necesarios para resolver la persistencia y el que están encapsulados la forma de acceder a los datos. Hace uso de las clases comentadas anteriormente para resolver la persistencia.

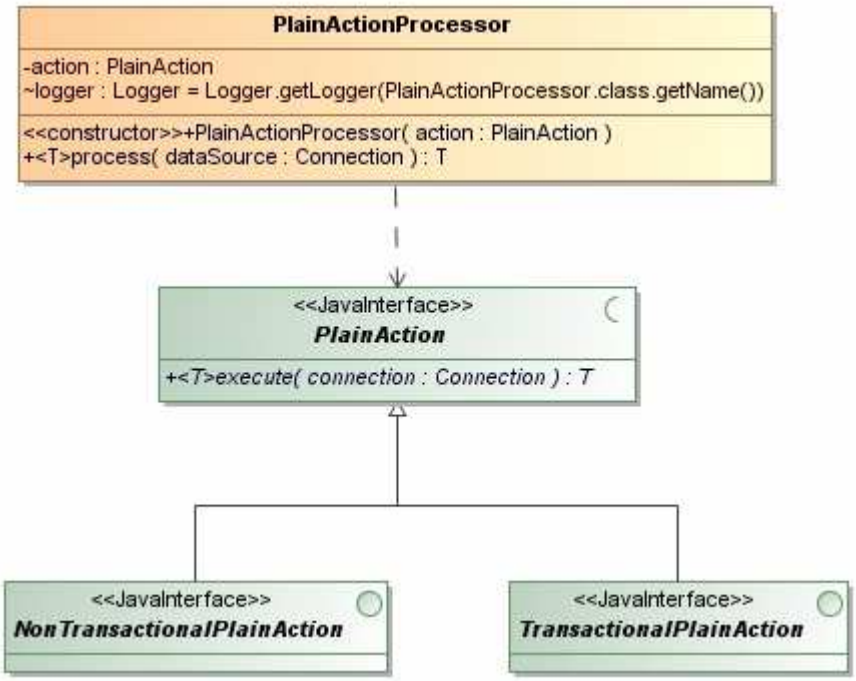
La arquitectura sobre la que se apoya el framework la podemos ver en el siguiente diagrama de clases:



PlainFwkFacadeDelegate implementa el patrón BussinessDelegate que por si mismo no realiza otra labor que delegar en el action correspondiente, que será la que implemente el método de la fachada.

Por lo tanto, para cada método de **PlainFwkFacadeDelegate** se creara la action adecuada, pasándole en su constructor los parámetros necesarios para su ejecución, en este caso la conexión con la base de datos. Una vez instanciado el action, se delega en el **PlainActionProcessor** la responsabilidad de su ejecución. Este tendrá en cuenta si la action es o no transaccional para adecuarla al entorno transaccional del SGDB.

Para la clasificación de las actions en transaccionales o no transaccionales utilizaremos dos interfaces marcadoras: **TransacionalPlainAction** y **NonTransacionalPlainAction**. Esta clasificación la podemos ver en el siguiente diagrama de clases.



El PlainActionProcessor invoca el metodo execute() del action para comenzar su ejecución. Este obtendrá el DAO del framework **FrameworkDAOImpl** a partir de su correspondiente factoría **DAOFactory**.

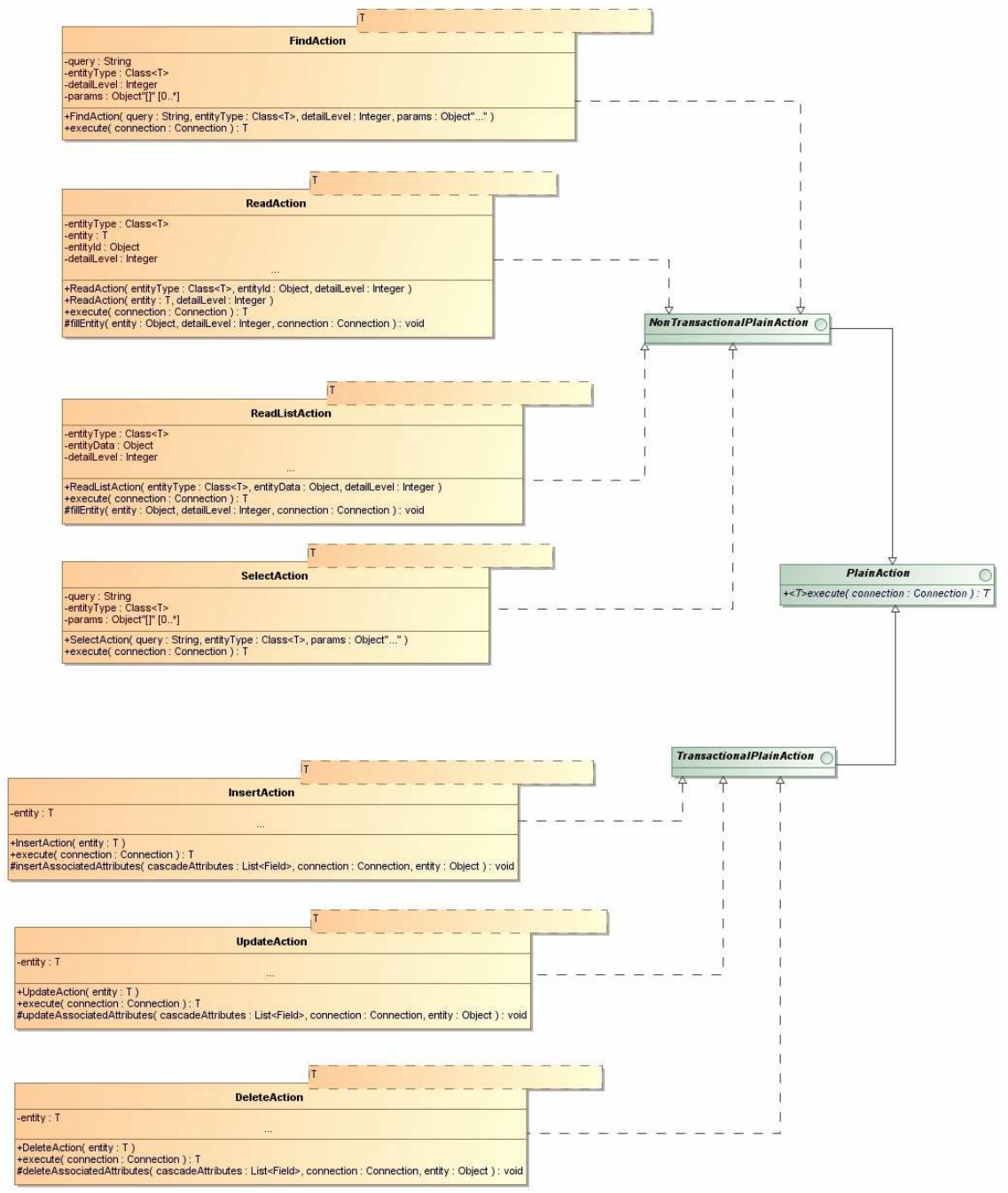
Una vez obtenido el DAO podremos insertar, actualizar, recuperar, etc. datos de la base de datos convenientemente.

Los **Action** implementados para resolver las funcionalidades que ofrece el framework son los siguientes:

- **ReadAction:** Action que se ejecuta para recuperar datos de una tabla a partir de la clave primaria
- **FindAction:** Recupera una lista de datos de una tabla usando un POJO anotado con @Table y utilizando una query hecha por el usuario
- **ReadListAction:** Action que se ejecuta para recuperar datos de una tabla a partir de los datos proporcionados en el POJO. Recupera una lista de datos de la tabla
- **SelectAction**: Action que se ejecuta para recuperar datos de una consulta y mapear los resultados en un POJO del tipo @NoTable
- **InsertAction**: Accion que inserta un nuevo elemento en una tabla

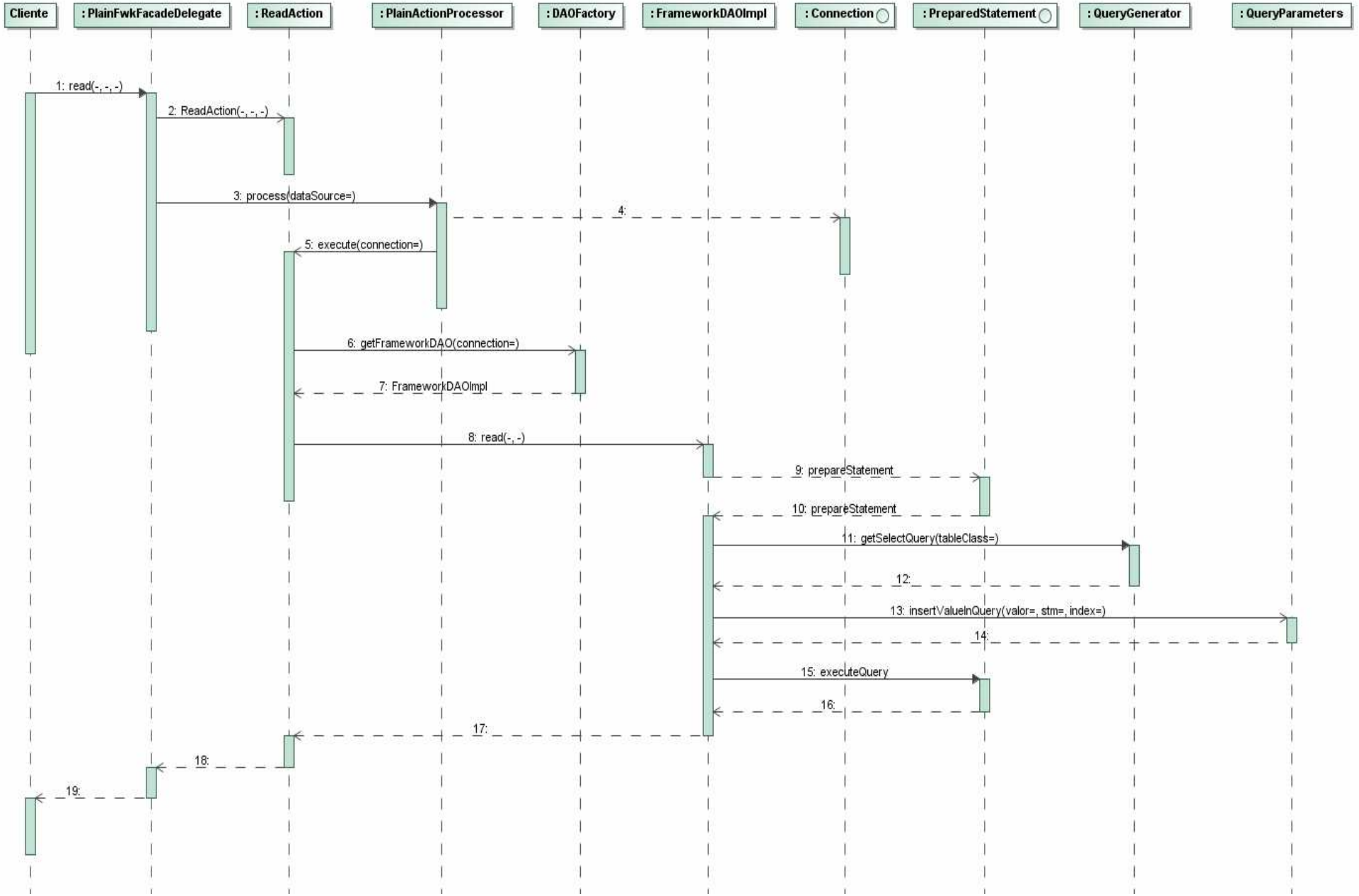
- **UpdateAction**: Actualiza un registro de una tabla de la BBDD
- **DeleteAction**: Ejecuta el borrado de una entidad de la base de datos

A continuación se muestra el diagrama de clases de los Action en donde se puede ver la clasificación a la que pertenecen (TransacionalPlainAction y NonTransacionalPlainAction)

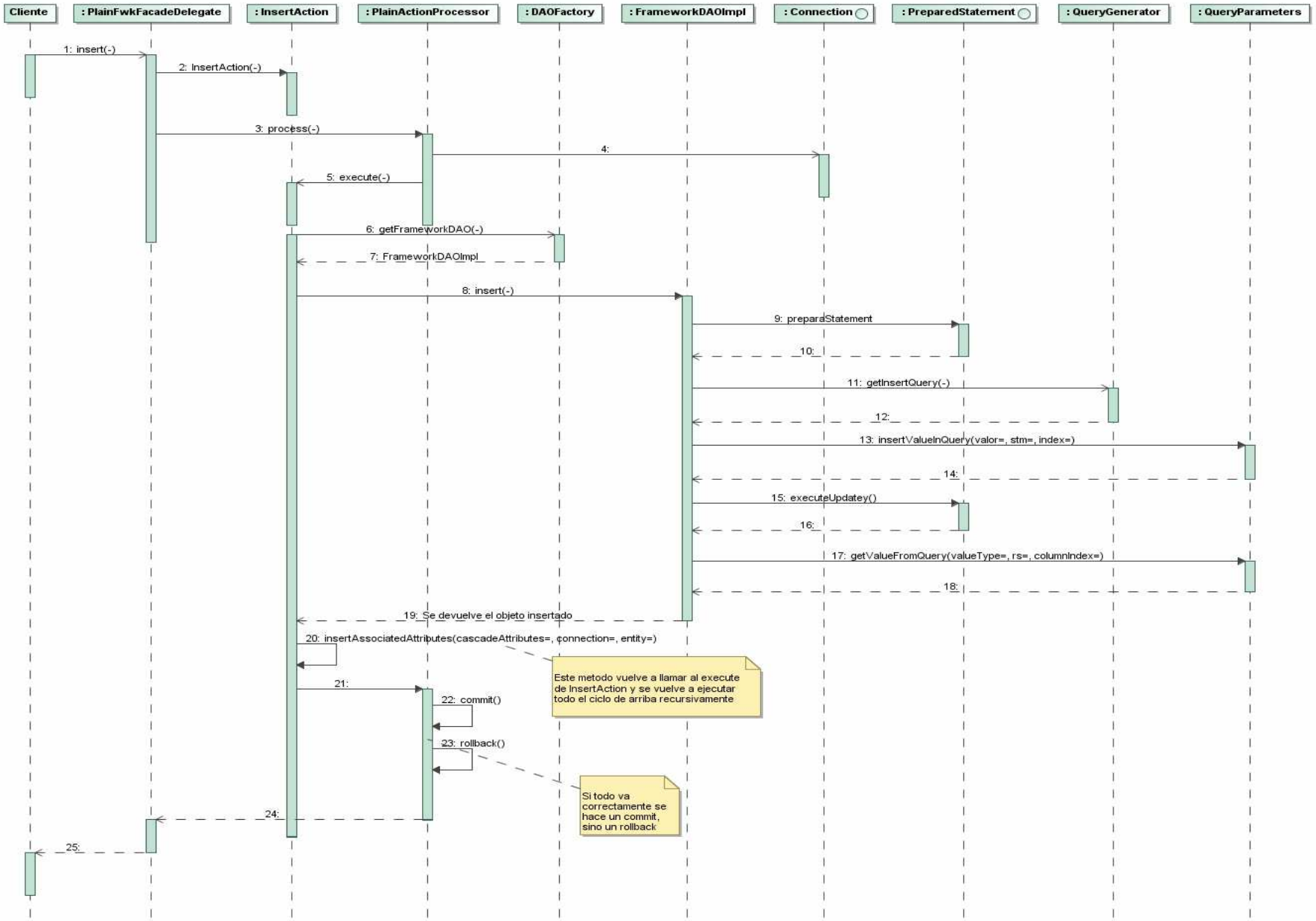


Para comprender el funcionamiento de esta arquitectura, a continuación vamos a mostrar el diagrama de secuencia de ejecución de la funcionalidad de leer un registro de la base de datos. Para ello se hace uso del Action ReadAction y de los metodos correspondientes en la fachada y en el DAO.

El comportamiento para el resto de action Transaccionales (FindAction, ReadListAction, SelectAction) del framework es similar


Cliente
: PlainFwkFacadeDelegate
: ReadAction
: PlainActionProcessor
: DAOFactory
: FrameworkDAOImpl
: Connection
: PreparedStatement
: QueryGenerator
: QueryParameters
1: read(-, -, -)
2: ReadAction(-, -, -)
3: process(dataSource=)
4:
5: execute(connection=)
6: getFrameworkDAO(connection=)
7: FrameworkDAOImpl
8: read(-, -)
9: prepareStatement
10: prepareStatement
11: getSelectQuery(tableClass=)
12:
13: insertValueInQuery(valor=, stm=, index=)
14:
15: executeQuery
16:
17:
18:
19:

En el caso de que el Action se transaccional (InsertAction, UpdateAction, DeleteAction) estan pensados para que pueda realizar la inserción, la actualización y el borrado de las entidades asociadas que tiene la entidad, por lo que se tienen que ejecutar en una única accion. Si el proceso es correcto, se hace commit. En cambio, si alguna de las operaciones falla, se hace rollback de la operación

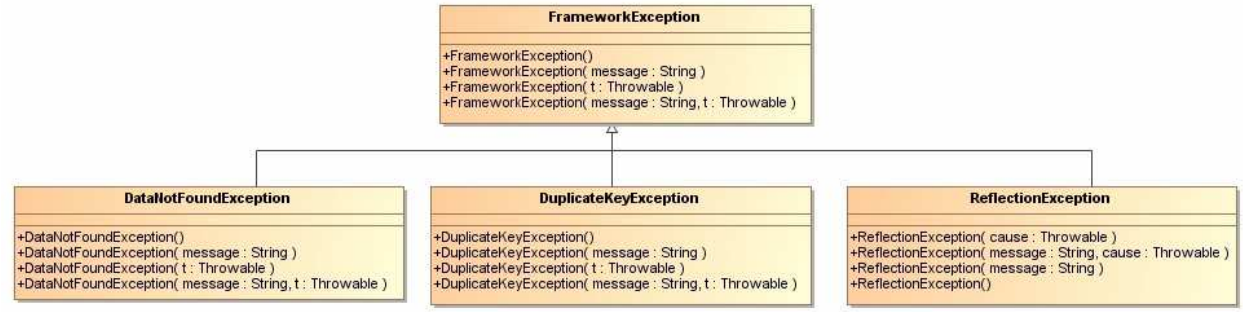

Cliente
: PlainFwkFacadeDelegate
: InsertAction
: PlainActionProcessor
: DAOFactory
: FrameworkDAOImpl
: Connection
: PreparedStatement
: QueryGenerator
: QueryParameters
1: insert(-)
2: InsertAction(-)
3: process(-)
4:
5: execute(-)
6: getFrameworkDAO(-)
7: FrameworkDAOImpl
8: insert(-)
9: preparaStatement
10:
11: getInsertQuery(-)
12:
13: insertValueInQuery(valor=, stm=, index=)
14:
15: executeUpdatey()
16:
17: getValueFromQuery(valueType=, rs=, columnIndex=)
18:
19: Se devuelve el objeto insertado
20: insertAssociatedAttributes(cascadeAttributes=, connection=, entity=)
Este metodo vuelve a llamar al execute de InsertAction y se vuelve a ejecutar todo el ciclo de arriba recursivamente
21:
22: commit()
23: rollback()
Si todo va correctamente se hace un commit, sino un rollback
24:
25:

4.3.7 CONTROL DE ERRORES

Pueden producirse excepciones al hacer uso de las operaciones que ofrece el framework de persistencia que deben de ser manejadas.

Las excepciones recogerán las situaciones de errores anormales e inesperados que no pueden ser tratadas en el flujo normal de una operación del framework, por ejemplo, la interrupción de la comunicación

Para ello se ha definido una jerarquía de excepciones en la que FrameworkException es la clase base para todas las excepciones del framework.



4.3.8 USO DEL FRAMEWORK

Como ya se ha comentado en los apartados anteriores, el uso de las operaciones que ofrece el framework se hace a traves de una fachada, que es la que nos da acceso a ejecutar las operaciones. Los pasos a seguir para utilizar el framework en nuestras clases java, bien desde una aplicación web o desde una aplicación stand-alone son los siguientes:

1. Crear los POJOS anotados que representan el modelo de datos a utilizar

2. Obtener una conexión con la base de datos a traves de la clase "Connection"

3. Crear una instancia de PlainFwkFacadeDelegate y pasarle la conexión obtenida anteriormente.

4. Llamar al metodo correspondiente de la fachada PlainFwkFacadeDelegate

A continuación se muestra un extracto del codigo utilizado en la aplicación de ejemplo para listar los clientes de la tabla Cliente:

```
public class ClienteAction extends DispatchAction {
    private Log logger = LogFactory.getLog(this.getClass());
    private Connection conn = null;
    private PlainFwkFacadeDelegate instance=null;

    public ActionForward listarClientes(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws Exception {
        logger.debug("listarClientes");
        ActionMessages errors = new ActionMessages();
        List<Cliente> clientes=null;
        StringBuilder query = new StringBuilder("SELECT CLIENTE_ID," +
       "NOMBRE,DIRECCION,CODIGOPOSTAL,PROVINCIA_ID,TELEFONO,EMAIL,DESCUENTO_ID FROM
CLIENTE ORDER BY CLIENTE_ID");

       try {
              conn = Conexion.getConnection();
              instance = new PlainFwkFacadeDelegate(conn);
              clientes = instance.finder(query.toString(),Cliente.class,2,null);

       } catch (DataNotFoundException e) {
       errors.add("NoData", new ActionMessage("No existen clientes en el sistema"));

       } finally{
        if (conn != null)
              try {
                     conn.close();
              } catch (SQLException e) {
                     throw new Exception ("ClienteAction.listarClientes--> Error
cerrando la conexion");
              }
       }
        if (!errors.isEmpty()) {
            saveErrors(request, errors);
        }
        request.setAttribute(Constants.CLIENTES, clientes);
        return mapping.findForward(Constants.SUCCESS);
    }
```

5 APLICACIÓN DE EJEMPLO

5.1 INTRODUCCIÓN

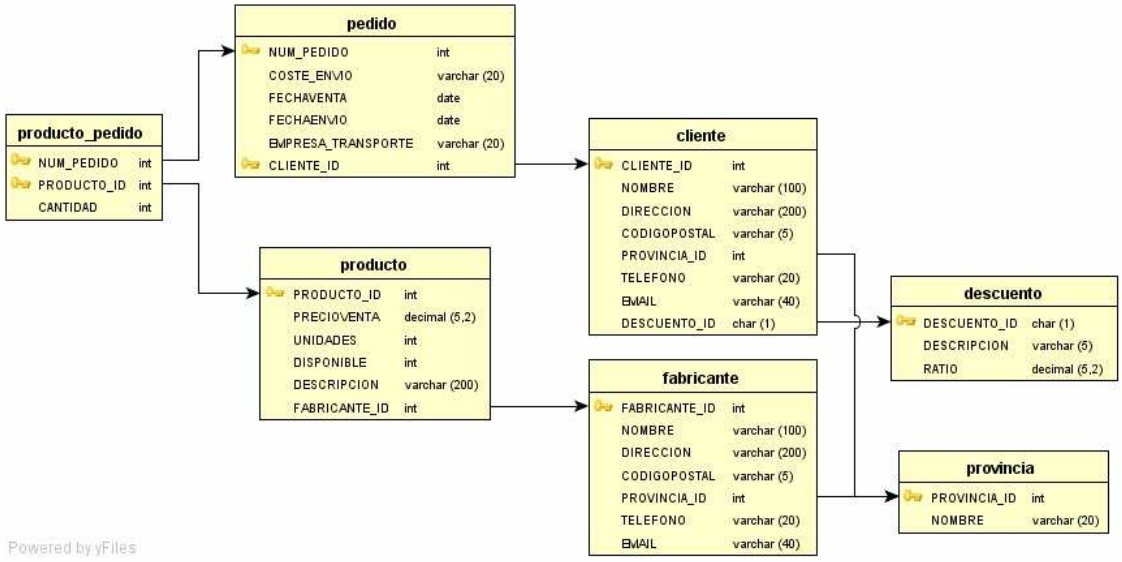
Para demostrar el funcionamiento del framework y de las operaciones que ofrece, se va a implementar una pequeña aplicación web que apoyandose en las funcionalidades del framework realiza operaciones sobre un modelo de datos definido.

El framework esta preparado para poder utilizarse dentro de cualquier tipo de aplicación java, pero se ha escogido construir una aplicación web porque resulta mas sencillo ver los resultados y es mas comodo de ejecutar.

La aplicación web esta construida sobre Struts, que es un framework de desarrollo de aplicaciones web basado en el patron MVC (Modelo-Vista-Controlador) y usa como base de datos MySQL.

Se ha empaquetado las clases del framework en un archivo jar llamado “persistenciaPFCUOC.jar” para su uso dentro de la aplicación web.

El modelo de datos sobre el que se va a operar es el siguiente:



Se trata de un modelo de datos sencillo donde existe una tabla de clientes, que tienen pedidos, que a su vez estan formados por uno o varios productos.

También tenemos una tabla de fabricantes que pueden tener uno o más productos.

Tabla Cliente:
Relación 1 a1 con Descuento y Provincia
Relacion 1 a N con tabla Pedido

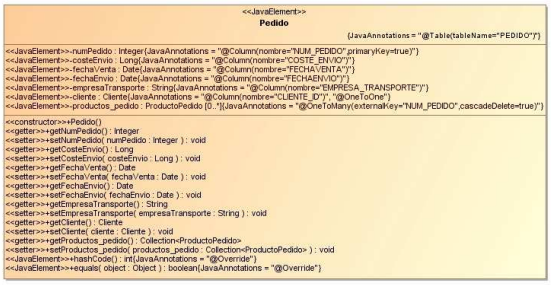
Tabla Fabricante

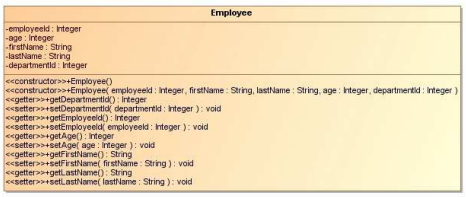
Relacion 1 a1 con Descuento y Provincia
Relacion 1 a N con Fabricante

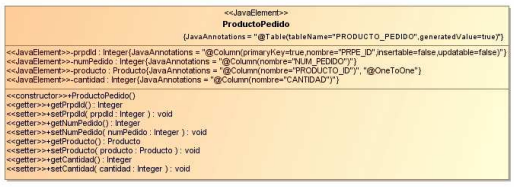
Tabla Pedido

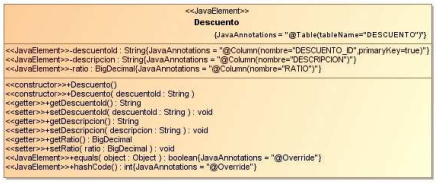
Relación 1 a N con Producto_pedido (1 pedido puede tener 1 o mas productos).

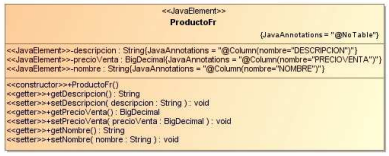
El modelo anterior quedaria representado por los siguientes POJOS anotados con las anotaciones definidas en el framework de la siguiente manera:

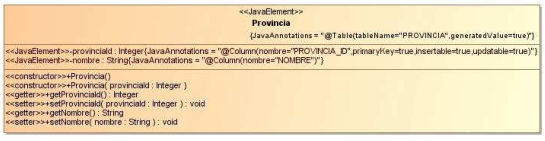


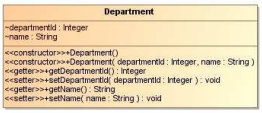












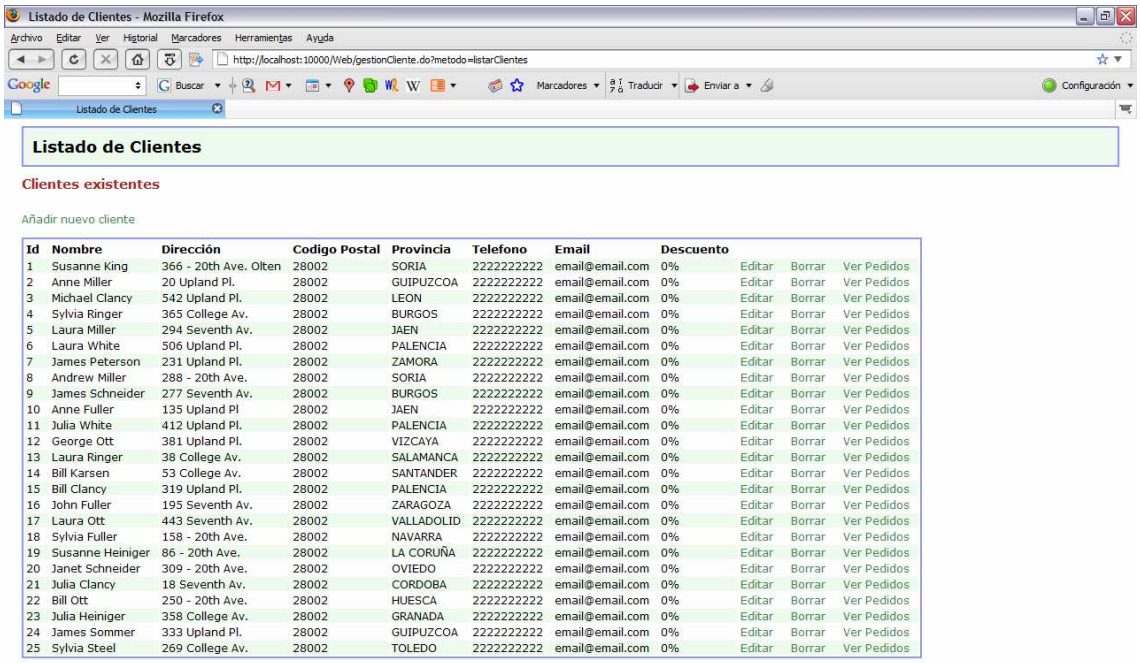
Con lo que conseguiríamos el mapeo entre las tablas del modelo y los objetos que utilizaremos para conseguir la persistencia dentro de la aplicación

5.2 USO DE LA APLICACIÓN

La página de inicio de la aplicación es: http://{RUTA_SERVIDOR}:8080/Web/index.jsp , a partir de la cual, se pueden ir ejecutando las operaciones sobre el modelo de datos.

5.2.1 LISTAR CLIENTES

Esta operación recupera el contenido de la tabla “Cliente” y muestra sus datos en un listado como el siguiente:



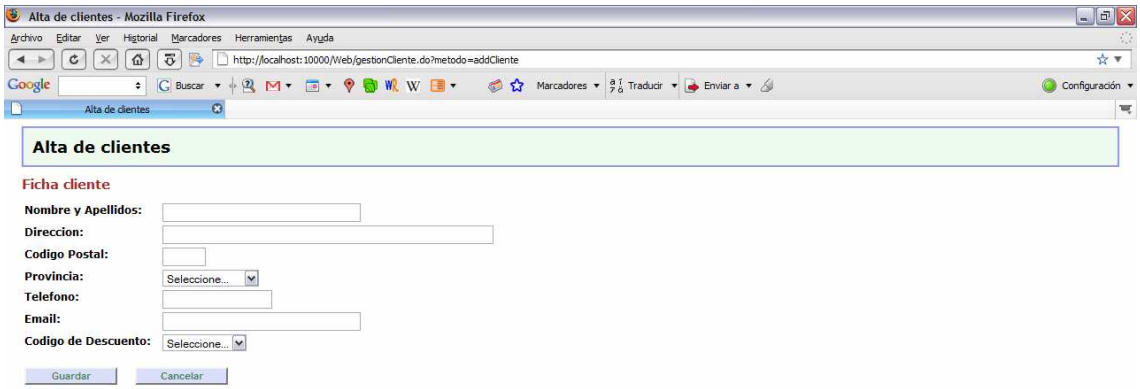
Id	Nombre	Dirección	Codigo Postal	Provincia	Telefono	Email	Descuento			
1	Susanne King	366 - 20th Ave. Olten	28002	SORIA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
2	Anne Miller	20 Upland Pl.	28002	GUIPUZCOA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
3	Michael Clancy	542 Upland Pl.	28002	LEON	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
4	Sylvia Ringer	365 College Av.	28002	BURGOS	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
5	Laura Miller	294 Seventh Av.	28002	JAEN	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
6	Laura White	506 Upland Pl.	28002	PALENCIA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
7	James Peterson	231 Upland Pl.	28002	ZAMORA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
8	Andrew Miller	288 - 20th Ave.	28002	SORIA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
9	James Schneider	277 Seventh Av.	28002	BURGOS	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
10	Anne Fuller	135 Upland Pl	28002	JAEN	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
11	Julia White	412 Upland Pl.	28002	PALENCIA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
12	George Ott	381 Upland Pl.	28002	VIZCAYA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
13	Laura Ringer	38 College Av.	28002	SALAMANCA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
14	Bill Karsen	53 College Av.	28002	SANTANDER	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
15	Bill Clancy	319 Upland Pl.	28002	PALENCIA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
16	John Fuller	195 Seventh Av.	28002	ZARAGOZA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
17	Laura Ott	443 Seventh Av.	28002	VALLADOLID	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
18	Sylvia Fuller	158 - 20th Ave.	28002	NAVARRA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
19	Susanne Heiniger	86 - 20th Ave.	28002	LA CORUÑA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
20	Janet Schneider	309 - 20th Ave.	28002	OVIEDO	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
21	Julia Clancy	18 Seventh Av.	28002	CORDOBA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
22	Bill Ott	250 - 20th Ave.	28002	HUESCA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
23	Julia Heiniger	358 College Av.	28002	GRANADA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
24	James Sommer	333 Upland Pl.	28002	GUIPUZCOA	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos
25	Sylvia Steel	269 College Av.	28002	TOLEDO	2222222222	email@email.com	0%	Editar	Borrar	Ver Pedidos

Este listado se recupera a partir de la operación “*finder*” del framework accesible desde la fachada “PlainFwkFacadeDelegate”

A partir de esta pantalla se pueden realizar diferentes operaciones con los clientes:

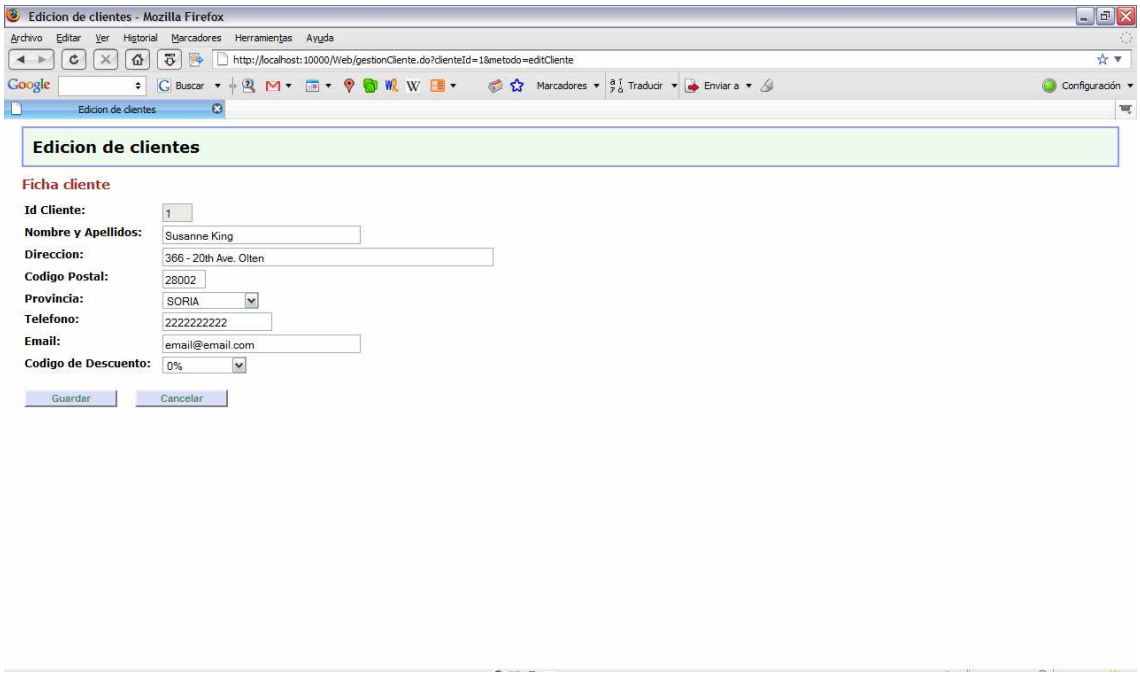
5.2.2 AÑADIR NUEVO CLIENTE

Operación que permite dar de alta un nuevo cliente en la tabla cliente utilizando la operación ***insert***



5.2.3 EDITAR CLIENTE

Esta operación hace uso del metodo ***read*** para recuperar los datos del cliente a editar, y del metodo ***update*** del framework para actualizar los datos del cliente

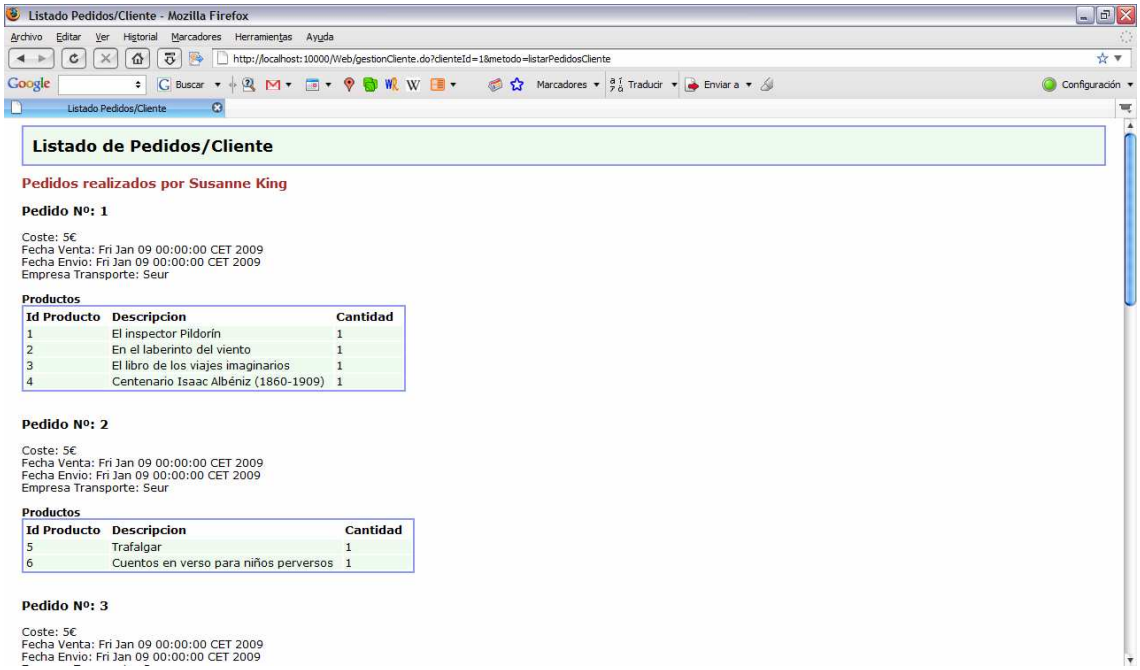


5.2.4 BORRAR

Borra el cliente y todos los pedidos asociados mediante la operación ***delete***. Se hace un borrado en cascada de las entidades asociadas a la tabla cliente, ya que asi se ha definido el POJO Cliente.

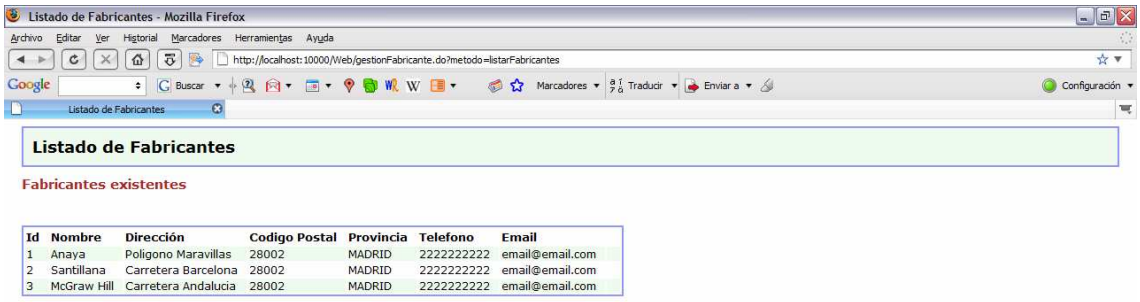
5.2.5 VER PEDIDOS

Haciendo uso de la operación ***readAll*** del framework, se recuperan los pedidos y la información asociada al mismo del cliente seleccionado.



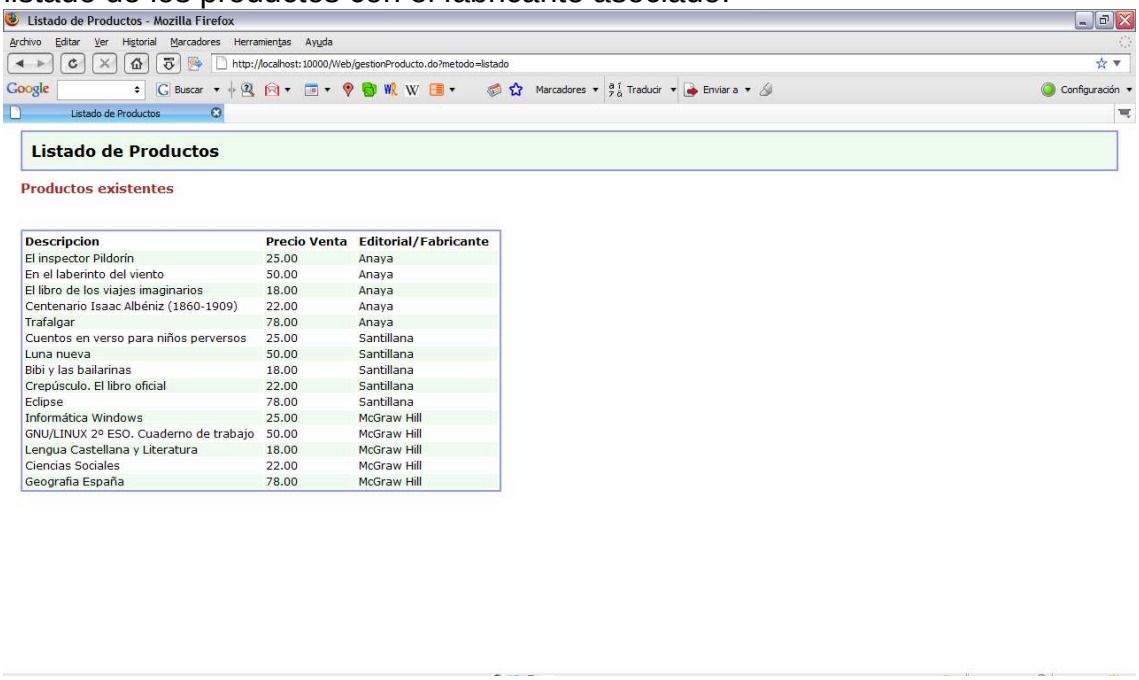
5.2.6 LISTAR FABRICANTES

Operación que nos muestra el listado de fabricantes existente en la tabla Fabricante haciendo uso de la operación ***finder*** del framework



5.2.7 LISTADO PRODUCTOS

Se ha definido esta operación para mostrar el funcionamiento del metodo *select* a partir del cual se pude definir una consulta y devolver el resultado en un POJO anotado con @NoTable (ProductoFr). Se trata de una busqueda sobre la tabla producto y fabricante, realizando el cruce entre las dos para mostrar un listado de los productos con el fabricante asociado.



5.3 INSTALACIÓN DE LA APLICACION

Para proceder al uso de la aplicación de ejemplo desarrollada, debemos instalar previamente el software necesario para poder utilizarla.

Java Runtime Environment (JRE) version 5.0 or posterior

Para ejecutar el servidor de aplicaciones y poder usar el framework deberemos tener instalado en la maquina el JRE version 5.0 o posterior.

Para ello, deberemos de descargarlo de http://java.sun.com/javase/downloads/index.jsp e instalarlo de acuerdo a las instrucciones que acompañan a la distribución.

Una vez instalado el JRE, deberemos de definir una variable de entorno JRE_HOME con el pathname del directorio donde se ha instalado el JRE, por ej c:\jre5.0

Tambien, existe la posibilidad de instalarse el JDK (Java Development Kit) en vez de instalarse solo el JRE. En este caso hay que definir una variable de entorno JAVA_HOME con el path del directorio de la instalación del mismo, por ej. C:\ jdk1.6.0_05

Para añadir una variable de entorno en Windows, tenemos que ir al siguiente cuadro de diálogo que se abre con Inicio→Configuración→Panel de Control→Sistema→Avanzado→Variables de Entorno

Tomcat

Es necesario descargar el fichero apache-tomcat-5.5.27.zip del sitio de Apache en la dirección http://apache.rediris.es/tomcat/tomcat-5/v5.5.27/bin/apache-tomcat-5.5.27.zip.

Posteriormente se descomprime el fichero en el disco duro de nuestro ordenador, preferiblemente en el directorio raíz.

Tras esta operación se habrán creado una serie de directorios entre los que cabe destacar los siguientes:
bin: Directorio con los ficheros ejecutables para arrancar y para el servidor
doc: Documentación de la distribución en formato html
conf: Ficheros de configuración.
webapps: Contiene diversas aplicaciones web de ejemplo. En este directorio será donde depositaremos la aplicación de ejemplo.

Para arrancar el motor será necesario ejecutar el fichero startup.bat que está en la carpeta bin de Tomcat.

Para pararlo hay que ejecutar shutdown.bat

MySQL

MySQL es un sistema de administración de bases de datos muy potente. La principal virtud es que es totalmente gratuito, por lo que es una fuerte alternativa ante sistemas como SQL u Oracle. Ademas, cumple los requerimientos minimos para mostrar el funciomiento del framework. Cabe decir que el framework se puede utilizar con cualquier motor de base de datos para el que exista driver JDBC.

Aquí veremos como instalar y configurar la versión de MySQL para Windows.

Para proceder a la instalación es necesario descargar en el sitio oficial de MySQL (http://www.mysql.com/) la versión correspondiente al sistema operativo Windows. La versión utilizada en este proyecto es mysql-4.1.22-win32 y la versión se puede descargar desde la siguiente url http://dev.mysql.com/get/Downloads/MySQL-4.1/mysql-4.1.22-win32.zip/from/http://mysql.easynet.be/

Para instalar esta versión, descomprimimos el archivo .ZIP en cualquier carpeta, ya que posee un sistema de instalación gráfico como la mayoría de los programas de la actualidad, y ejecutamos el archivo setup.exe. Este paso es realmente sencillo, ya que solamente debemos seleccionar el directorio donde se instalará (por defecto c:\mysql) y el tipo de instalación a realizar: Típica, Completa o Personalizada.

Para ejecutar MySQL Nos dirigimos hacia el directorio c:\mysql\bin y ejecutamos el archivo winmysqladmin.exe. y a continuación nos aparecerá un icono de un semáforo en la barra de tareas.

También puede instalarse como servicio de Windows, para ello nos dirigiremos a Inicio→Configuración→Panel de Control→Herramientas Administrativas→Servicios y buscaremos el servicio llamado MySQL. Con el botón derecho del ratón elegimos propiedades y modificamos la opción Tipo de inicio, seleccionando Automático. De esta manera MySQL se ejecutara siempre que el ordenador este encendido.

Instalación de la aplicación

Una vez tenemos todos los componentes de la arquitectura instalados, solo resta la instalación y configuración de la aplicación de ejemplo. Este proceso es de lo más simple.
Comenzaremos con copiar la aplicación web al directorio /webapps de Tomcat copiando el archivo Web.war situado en la carpeta dist.

En este momento, reiniciando Tomcat, ya tendremos la aplicación web disponible a través del servidor web, pero la base de datos estará completamente vacía.

A continuación habrá que crear las tablas en servidor de bases de datos

mediante la ejecución del script SQL que viene en el directorio dist y que se llama “productoEjemplo.sql”. Para ello primero tenemos que crear la base de datos que se llamara “pfc”. Entramos en una sesión MS-DOS, nos colocamos en el directorio/bin de mysql y tecleamos “mysql” e importamos la base de de datos desde el scrit productoEjemplo.sql. Para ello tecleareMos *“source productoEjemplo.sql”*

Por ultimo, se ha que verificar los parámetros de conexión de la aplicación web con la base de datos. Tendremos que editar los valores del fichero “conexion.properties” situado en la ruta %DIRECTORIO INSTALACION TOMCAT%\webapps\Web\WEB-INF\

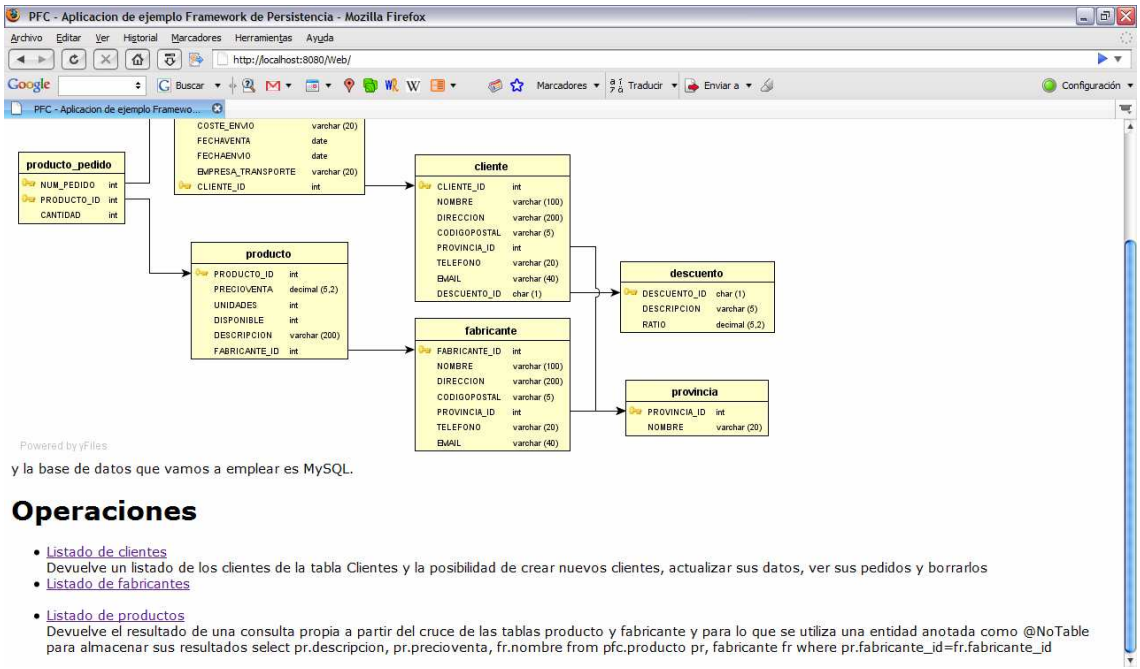
Los valores por defecto que traen son:

driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/pfc
user=root
password=

y tendremos que cambiarlos por los de la instalación de mysql que hayamos realizado.

Si todo finaliza con éxito ya podremos acceder a la aplicación web y trabajar con ella accediendo a la siguiente dirección web desde nuestro navegador: http://127.0.0.1:8080/Web.

La página de inicio es la siguiente:



6 CONCLUSIONES

6.1 OBJETIVOS CUMPLIDOS

En la introducción se plantearon una serie de objetivos del proyecto y personales a completar durante la realización del proyecto que se han cumplido.

La realización de este proyecto me ha permitido estudiar a fondo el problema de la persistencia dentro de aplicaciones java y la correspondencia entre le modelo relacional y el modelo de objetos.

Asi mismo, me ha permitido conocer a fondo una de las APIs menos conocidas de java, que el API de reflection, de la cual se hace un uso intensivo en el framework construido. También de una de las últimas caracteristicas añadidas en la versión 1.5 de Java como son las *anotaciones* de las que he hecho uso para construir la base del framework.

6.2 AMPLIACIONES

El framework desarrollado solo cubre las operaciones basicas a realizar sobre una base de datos. Ademas, solo se ha probado sobre modelos de datos muy basicos y que presentan poca complejidad.

No se ha probado sobre un modelo de grandes dimensiones con multiples relaciones, tipos de datos y con caracteristicas propias del motor de base de datos que pueden hacer que el framework no se comporte de la manera deseada.

A continuación, paso a enumerar una serie de ampliaciones que se pueden hacer al framework, asi como mejoras que se pueden implementar para facilitar su utilización:

- **Ampliación de las anotaciones**: Crear nuevas anotaciones para dotar a los POJOS de mayor información acerca de las tablas de la base de datos, asi como la amplicación de relaciones entre entidades y el manejo de las claves.

- **Capa de abstracción de base de datos:** Crear una capa intermedia que abstraiga realmente el funcionamiento del framework del motor de base de datos a utilizar (mysql, oracle, db2, etc) y permita realizar consultas SQL mas avanzadas o complicadas. Ahora mismo, unicamente se han implementado consultas SQL estandar que funcionan en cualquier base de datos para la que existe driver JDBC.

- **Herramienta de importación/exportacion:** herramienta que fuese capaz de crear los POJOS anotados automáticamente a partir de un modelo de datos y viceversa, a partir de nuestros POJOS anotados, se cree el modelo de datos.

- **Capa de conexión con la base de datos:** Implementar las clases necesarias para que la conexión con la base de datos se haga de una forma sencilla y sea soportada por el framework, con la posibilidad de recuperar la conexión de un pool de conexiones o a traves de JNDI. Tambien, abstraer al programador de la tarea de crear la conexión, utilizarla y liberarla, ya que son tareas que pueden llevar a errores inesperados en la aplicación si se olvida alguna de ellas.

Cualquier ampliación sobre el framework y sus operaciones es fácilmente aplicable, ya que la arquitecutura que sigue el mismo, hace que sea muy sencillo implementar nuevas anotaciones, nuevas operaciones, tan solo introduciendo las piezas necesarias en los paquetes pertinentes.

7 GLOSARIO

- **API:** Application Programming Interface) es el conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción

- **Atributo de una entidad:** Propiedad que interesa de una entidad.

- **Base de Datos:** Es un conjunto estructurado de datos que representa, entre otros, entidades y sus interrelaciones, con integración y compartimentación de datos.

- **BD:** Ver Base de Datos.

- **Cache:** Memoria que permite mejorar el rendimiento del sistema al almacenar los datos mas comunes y repetidos.

- **Diseño Conceptual:** Etapa del diseño de una base de datos que obtiene una estructura de la información de la futura base de datos independiente de las tecnología que se quiere utilizar.

- **Diseño Lógico:** Etapa del diseño de una base de datos que parte del resultado del diseño conceptual y lo transforma de modo que se adapte al modelo del SGBD con el que se desea implementar la base de datos.

- **Driver JDBC:** Implementación particular del JDBC (Java Database Connectivity) para un SGBD concreto.

- **Entidad Débil:** Entidad cuyos atributos no la identifican completamente, sino que sólo la identifican de forma parcial.

- **Entidad:** Es un objeto que es distinguible de otros objetos por medio de un conjunto específico de atributos.

- **Interrelación:** Asociación entre entidades.

- **Java:** Java es un lenguaje de programación expresamente diseñado para utilizarse en el entorno distribuido de Internet. Fue diseñado para que fuera muy similar a C++, pero es más fácil de utilizar que este y sigue un modelo de programación orientada a objetos.

- **JDBC:** Java Database Connectivity. Es un API estándar de Java para interactuar con las abstracciones y conceptos de las BD relacionales y, en concreto, para trabajar con SQL.

- **Modelo de datos**: Termino que hace referencia al sistema formal y abstracto que nos permite describir los datos a partir de una serie de reglas.

- **Persistencia:** Propiedad que hace referencia al hecho de que los datos sobre los que trata un programa no se pierdan al acabar su ejecución.

- **SGBD:** Ver Sistema de Gestión de Bases de Datos.

- **Sistema de Gestión de Bases de Datos:** Software que gestiona y controla bases de datos. Sus principales funciones son las de facilitar la utilización simultánea a muchos usuarios de tipos diferentes, independizar al usuario del mundo físico y mantener la integridad de los datos.

- **SQL:** Lenguaje pensado para describir, crear, actualizar y consultar bases de datos. Fue concebido por IBM a finales de los años setenta y estandarizado por ANSI e ISO en el año 1986 (el último estándar del SQL es de 1999). Actualmente lo utilizan casi todos los SGBD del mercado (incluso algunos SGBD no relacionales y algunos sistemas de ficheros).

- **Transacción:** Es una acción.

- **XML:** Tambien conocido como Extensible Markup Lenguaje o lenguaje extensible de marcas. Es un lenguajes dirigido al etiquetado de las partes que forman un documento y es ademas un metalenguaje que permite definir otros lenguajes y que facilita la comunicación entre lenguajes de diferente naturaleza.

BIBLIOGRAFIA

The Design of a Robust Persistence Layer for Relational Databases
Scott W. Ambler
Ronin International, Noviembre, 2000.

Mapping objects to relational database, White Paper,
Scott W. Ambler
Ronin Internacional, Octubre, 2000.

Java Data Objects
Robin Roos
Addison Wesley Publisher LTd, 2002

Object Data Management
R.G.G. CATTELL
Addison Wesley, 1994

Fundamentos de bases de datos, Tercera Edición
Autores: Henry F. Korth y Abraham Silberschatz
Editorial: McGraw-Hill

Thinking Java
Autores: Bruce Eckel
Prentice Hall

Aprenda Java como si estuviera en primero.
Autores: Javier García de Jalón, José Ignacio Rodríguez, Iñigo Mingo, Aitor Imaz, Alfonso Brazález, Alberto Lazarbal, Jesús Calleja, Jon García.
Escuela Superior de Ingenieros Industriales de la Universidad de Navarra

Java Sun
http://java.sun.com

Java en Castellano
http://java.programacion.net

Design Patterns: Elements of Reusable Object- Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,
Addison-Wesley, 1994

El Proceso Unificado de Desarrollo de Software
Autores: Ivar Jacobson, Grandy Booch y James Rumbaugh
Madrid: PEARSON EDUCACIÓN, S.A., 2000
ISBN: 84-7829-036-2

Building Web Applications with UML
Autor: Jim Conallen
Canada: PEARSON EDUCACIÓN CORPORATE SALES DIVISION, 2000
ISBN: 0-201-61577-0

Rational Software
http://www.rational.com/

UML Gota a Gota
Autor: Martin Fowler y Kendall Scott
Mexico: ADDISON WESLEY LONGMAN, INC, 1999
ISBN: 968-444-364-1

Documentos en la Red

Adopting a Java Persistence Framework: Which, When, and What?
http://today.java.net/pub/a/today/2007/12/18/adopting-java-persistence-framework.html

JPA-
http://www.devx.com/Java/Article/33650/

http://www.devx.com/Java/Article/33906

iBatis
http://ibatis.apache.org/

Hibernate
http:// www.hibernate.org/

Caso de estudio: diseño e implementación de la capa modelo de MiniBank con JDBC. Patrones usados
www.tic.udc.es/~fbellas/teaching/is-2001-2002/Tema2Apartado2.2.pdf

http://www.codeproject.com/KB/architecture/WhatIsAFramework.aspx

ANEXO A – IMPLEMENTACIÓN

A1. CÓDIGO FUENTE DEL FRAMEWORK

El codigo fuente del framework y la documentación javadoc puede consultarse en las fuentes entregadas con el proyecto.