Universitat Oberta de Catalunya

PFC: J2EE
Diseño de un Framework de Persistencia

Diseño de un Framework de Persistencia
Alumno: Daniel Peña Gómez
Tutor: Josep Mª Camps Riva
Ingeniería informática
Curso 2011-2012

INDICE

1. Descripción del PFC

En líneas generales, un framework no es más que una colección de objetos que buscan brindar servicios para un subsistema particular.

Nos vamos a centrar en frameworks creados bajo la plataforma Java y orientada a persistencia de datos. Esta es la capacidad de un equipo para mantener la información incluso después de desconectarlo o cerrar una aplicación que utiliza la información.

El objetivo de esta PFC 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 JEE.

Una de las partes más complejas que hay que resolver en plantear cualquier aplicación es el acceso a los datos persistentes que debe manejar (muchas veces bases de datos relacionales). Este PFC deberá estudiar la problemática de la capa de acceso a datos de una aplicación distribuida, y escoger una estrategia para resolverlo.

Lo primero que tenéis que hacer es evaluar distintas alternativas existentes, fijándose os especialmente en los frameworks de mapeo entidad-relación existentes en el mercado (Hibernate, iBatis, etc.). Hay que tratar diferentes tipos de persistencia de datos (bases de datos relacionales, bases de datos OO, XML, ...) Hay que analitzue los diferentes enfoques de cada uno de ellos, las características que tienen en común y las que los hacen diferentes. Una vez hecho esto tenéis que hacer el diseño y la implementación de un framework de este estilo que implemente las funcionalidades básicas y, por último, construir una aplicación que muestre el funcionamiento.

El framework que diseñe estará formado por un conjunto de clases o componentes y cada uno de ellos proporcionará la abstracción de algún concepto. Estos componentes deben ser reutilizables y basados en tecnologías estándar como Servlets, JavaBeans, XML, etc ...

La aplicación debe mostrar claramente cómo se desarrollaría utilizando estos componentes y cómo interaccionan entre ellos.

2. Objetivos generales y específicos

2.1 Objetivos generales

El objetivo principal de esta PFC es la obtención de conocimientos más profundos sobre el uso del patrón arquitectural MVC (Modelo-Vista-Controlador) en el desarrollo de aplicaciones, en nuestro caso investigando/desarrollando la implementación de un framework de persistencia.

En cuanto a la creación de la herramienta de apoyo para el uso del framework, espero que sea intuitiva en su uso y genere código de una manera eficiente para que haga más ameno el uso de los componentes que se han generado en el framework, facilitando la curva de aprendizaje al futuro usuario de ésta.

El proyecto de desarrollo del PFC pretende abarcar en la medida de lo posible los siguientes aspectos:

- Definición básica de framework.
- La persistencia de datos
- Evaluación de alternativas existentes (Búsqueda de información sobre frameworks open source y propietarios relacionados con la persistencia de datos).
- Diseño de framework de persistencia
    a) Análisis de requisitos de un framework de persistencia.
    b) Características a observar en el diseño:
        - Análisis del dominio
        - Diseño del framework
        - Implantación del framework
- Implementación del framework de persistencia usando tecnología java.
- Desarrollo de una aplicación simple de ejemplo para verificar su funcionamiento en real.

2.2 Objetivos específicos

El desarrollo de una aplicación framework de persistencia es muy complejo, por lo que se intentará desarrollar la implementación específica de la manera más simple posible, y reduciendo el campo de aplicación a algún sistema de Gestión de Bases de Datos generalista.

Los elementos que se emplearán en su desarrollo:

- La aplicación se desarrollará en Java para poder ser utilizada en cualquies sistema operativo con compatibilidad java (en nuestro caso Windows 7 y Ubuntu)
- Las versiones estándar que se usarán para el desarrollo y planificación, así como para las pruebas finales, serán:
    1. Modelage UML, Microsoft Visio 2003
    2. Planificación, Microsoft Project 2003
    3. Editor de textos, Microsoft Word 2007
    4. Se prevé realizar el desarrollo pensando en bases de datos Oracle y MySQL

• En cuanto al desarrollo en la etapa del Diseño, se intentará facilitar la mayor cantidad de Diagramas UML posible, con el objeto de valorar y sistematizar el diseño a fin de que pueda ser tomado como base para futuras ampliación del modelo. Concretamente, como mínimo:

a) Modelos estáticos: diagramas de clases.

b) Diagramas de secuencia.

c) Diagramas de uso.

2.3 Objetivo del desarrollo

Este PFC va encaminado al análisis, diseño e implementación de componentes de software, típicamente llamado framework, con la idea de servir de base para desarrollos sencillos directos así como para rediseños escalares posteriores.

El objetivo de este PFC es triple:

• Estudiar y evaluar las diferentes alternativas existentes para implementar la capa de persistencia para aplicaciones JEE de cliente delgado.

• Una vez estudiadas las alternativas que tiene disponibles, hacer el diseño y la implementación de un framework de mapeo entidad-relación.

• Construir una aplicación de ejemplo que muestre el uso del framework que ha implementado.

3. Planificación con hitos y temporalización

3.1 Planificación

3.1.1 Tareas correspondientes a la fase de toma de datos y análisis de frameworks:

- Comprensión de framework:
- Búsqueda de información en Internet.
- Análisis de los modelos de framework.
- Investigación sobre modelos open source y propietarios.
- Análisis de las herramientas generales framework:
- Elementos generales del uso en framework.
- Estructura de framework.
- Diseño general de un framework
- Análisis de la persistencia de datos.
- Características de la persistencia.
- Tipos de sistemas de gestión de la persistencia.
- Tecnología para gestión de la persistencia.
- Búsqueda de información sobre modelos de frameworks orientados a la gestión de la persistencia (Modelos open source, Modelos propietarios y comparación de ambos).

3.1.2 Tareas relacionadas con el diseño de la aplicación framework de persistencia:

El desarrollo de la aplicación seguirá las fases del ciclo de vida en cascada de forma iterativa e incremental:

- Recogida y documentación de los requisitos para el framework:
  a) Elaboración de los diagramas necesarios para la comprensión y sistematización del modelo, esto es diagrama de casos de uso.
  b) Documentación textual de los casos de uso.

- Análisis:
  a) Revisión de los casos de uso de la etapa previa.
  b) Identificación de las clases. Representación mediante diagramas de colaboración para cada uno de los casos de uso.
  c) Identificación de las relaciones entre las clases de entidad y elaboración de un diagrama estático UML.
  d) Especificación formal de los casos de uso.
  e) Especificación de los diversos diagramas UML necesarios.

- Diseño:
  a) Explorar la posibilidad de utilizar patrones adicionales.

b) Establecimiento de los subsistemas de la aplicación, si son necesarios.

c) Especificación de las pruebas que describan con que datos se ha de probar cada programa o grupo de programas pertenecientes a un módulo y cuáles son los resultados esperados en cada caso.

d) Interacción con los Sistemas de Gestión de Bases de Datos.

• Implementación o codificación para cada uno de los módulos:

a) Implementación de las clases del framework utilizando el lenguaje java.

• Prueba de la aplicación y test de resultados:

a) Diseño y especificación de un plan de pruebas, tomando las especificaciones de las pruebas elaborados durante la etapa de diseño, así como la validación de los requisitos.

· Diseño de una aplicación de ejemplo.

3.1.3 Tareas relativas a la instalación de software.

• Entrega de la aplicación con sus módulos software de código, clases java, y si es posible contenedor jar.

• Integración del framework en el IDE seleccionado.

3.1.4 Confección de la documentación final del proyecto

• Elaboración de un manual de usuario para el uso del framework de persistencia.

• Revisión de la documentación interna del software (documentación de las clases).

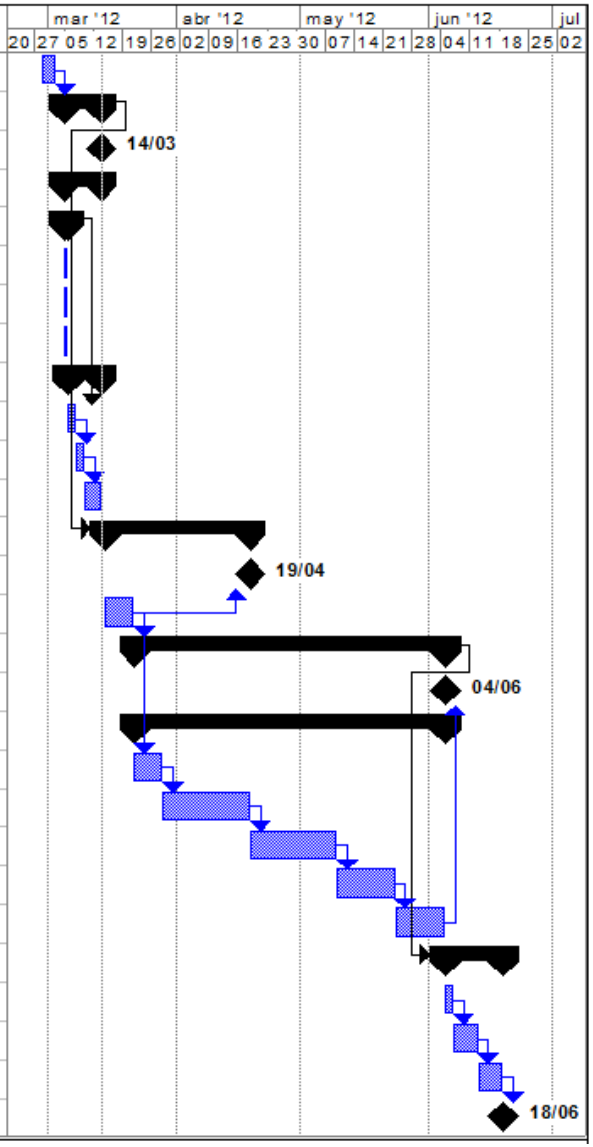
• Elaboración de un javadoc.

• Elaboración de la memoria del PFC

• Elaboración de la presentación del PFC

3.2 Diagrama de Gantt

Id	Nombre de tarea	Duración	Comienzo	Fin
1	Creación del plan del proyecto	3 días	mié 29/02/12	vie 02/03/12
2	**PEC 1**	**8 días?**	**lun 05/03/12**	**mié 14/03/12**
3	Entrega	0 días	mié 14/03/12	mié 14/03/12
4	**Toma de datos**	**8 días?**	**lun 05/03/12**	**mar 13/03/12**
5	**Descripció del PFC**	**1 día?**	**lun 05/03/12**	**lun 05/03/12**
6	Objetivos generales	1 día?	lun 05/03/12	lun 05/03/12
7	Objetivos especificos	1 día?	lun 05/03/12	lun 05/03/12
8	Objetivo del desarrollo	1 día?	lun 05/03/12	lun 05/03/12
9	**Framework**	**7 días**	**mar 06/03/12**	**mar 13/03/12**
10	Información básica	2 días	mar 06/03/12	mié 07/03/12
11	La persistencia de datos	2 días	jue 08/03/12	vie 09/03/12
12	Evaluación de alternativas existentes	3 días	sáb 10/03/12	mar 13/03/12
13	**PEC 2**	**25 días**	**jue 15/03/12**	**jue 19/04/12**
14	Entrega	0 días	jue 19/04/12	jue 19/04/12
15	Análisis de mi framework de persistencia	5 días	jue 15/03/12	mié 21/03/12
16	**PEC 3**	**53 días**	**jue 22/03/12**	**lun 04/06/12**
17	Entrega	0 días	lun 04/06/12	lun 04/06/12
18	**Diseño del Framework**	**53 días**	**jue 22/03/12**	**lun 04/06/12**
19	Recogida y documentación de los requisitos para el framework	5 días	jue 22/03/12	mié 28/03/12
20	Análisis	15 días	jue 29/03/12	mié 18/04/12
21	Diseño	15 días	jue 19/04/12	mié 09/05/12
22	Implementación o codificación para cada uno de los módulos	10 días	jue 10/05/12	mié 23/05/12
23	Prueba de la aplicación y test de resultados	8 días	jue 24/05/12	lun 04/06/12
24	**Memoria y presentación**	**10 días**	**mar 05/06/12**	**lun 18/06/12**
25	Confección de la documentación del proyecto	2 días	mar 05/06/12	mié 06/06/12
26	Elaboración Memoria PFC	4 días	jue 07/06/12	mar 12/06/12
27	Elaboración presentación PFC	4 días	mié 13/06/12	lun 18/06/12
28	Entrega final	0 días	lun 18/06/12	lun 18/06/12



4. Licencia GPL

Puede copiar y distribuir el Programa (o un trabajo basado en él, según se especifica en el apartado 2, como código objeto o en formato ejecutable según los términos de los apartados 1 y 2, supuesto que además cumpla una de las siguientes condiciones:
1. Acompañarlo con el código fuente completo correspondiente, en formato electrónico, que debe ser distribuido según se especifica en los apartados 1 y 2 de esta Licencia en un medio habitualmente utilizado para el intercambio de programas, o
2. Acompañarlo con una oferta por escrito, válida durante al menos tres años, de proporcionar a cualquier tercera parte una copia completa en formato electrónico del código fuente correspondiente, a un coste no mayor que el de realizar físicamente la distribución del fuente, que será distribuido bajo las condiciones descritas en los apartados 1 y 2 anteriores, en un medio habitualmente utilizado para el intercambio de programas, o
3. Acompañarlo con la información que recibiste ofreciendo distribuir el código fuente correspondiente. (Esta opción se permite sólo para distribución no comercial y sólo si usted recibió el programa como código objeto o en formato ejecutable con tal oferta, de acuerdo con el apartado 2 anterior).

5. Framework

5.1 Información básica

Dentro del ámbito de la programación, un framework es un conjunto de funciones o código genérico que para realizar tareas comunes y frecuentes en todo tipo de aplicaciones (creación de objetos, conexión a base de datos, limpieza de strings, etc.). Esto da la oportunidad para mantener una sólida base sobre la cual desarrollar aplicaciones específicas, permitiendo obviar los componentes más triviales y genéricos del desarrollo.

Básicamente, los frameworks son construidos en base a lenguajes orientados a objetos. Esto permite la modularización de los componentes y una óptima reutilización de código. Además, en la mayoría de los casos, cada framework específico implementará uno o más patrones de diseño de software que aseguren la escalabilidad del producto.

Dentro del proceso de desarrollo de software, un Framework consiste en una estructura de soporte definida en la cual un proyecto de software puede ser organizado y desarrollado. Un Framework puede incluir soporte de programas, bibliotecas y un lenguaje de scripting para ayudar a desarrollar y unir los diferentes componentes de un proyecto. Provee de una estructura y una metodología de trabajo la cual extiende o utiliza las aplicaciones del dominio.

En general, con el término Framework, nos estamos refiriendo a una estructura de software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación. En otras palabras, un Framework se puede considerar como una aplicación genérica incompleta y configurable a la que podemos añadirle las últimas piezas para construir una aplicación concreta.

Los objetivos principales que persigue un Framework son:

- acelerar el proceso de desarrollo
- reutilizar código ya existente
- promover buenas prácticas de desarrollo como el uso de patrones

5.2 La persistencia de datos

Persistencia es el hecho de guardar permanentemente la información generada por una aplicación en un sistema de almacenado, con el objetivo de qué se pueda usar más adelante.

Se pueden encontrar diferentes definiciones del término persistencia, según distintos puntos de vista y autores. Veamos dos que con más claridad y sencillez, concretan el concepto de persistencia de objetos. La primera, más antigua, dice así: “Es la capacidad del programador para conseguir que sus datos sobrevivan a la ejecución del proceso que los creo, de forma que puedan ser reutilizados en otro proceso. Cada objeto, independiente de su tipo, debería poder llegar a ser persistente sin traducción explicita. También, debería ser implícito que el usuario no tuviera que mover o copiar los datos expresamente para ser persistentes”.

Esta definición nos recuerda qué es tarea del programador, determinar cuándo y cómo una instancia pasa a ser persistente o deja de serlo, o cuando, debe ser nuevamente reconstruida; asimismo, que la transformación de un objeto en su imagen persistente y viceversa, debe ser transparente para el programador, sin su intervención; y que todos los tipos, clases, deberían tener la posibilidad de que sus instancias perduren.

Otra definición dice así: “Persistencia es «la capacidad de un lenguaje de programación o entorno de desarrollo de programación para, almacenar y recuperar el estado de los objetos de forma que sobrevivan a los procesos que los manipulan”. Esta definición indica que el programador no debería preocuparse por el mecanismo interno que hace un objeto ser persistente, sea este mecanismo

soportado por el propio lenguaje de programación usado, o por utilidades de programación para la persistencia, como librerías, framework o compiladores.

En definitiva, el programador debería disponer de algún medio para poder convertir el estado de un objeto, a una representación adecuada sobre un soporte de información, que permitirá con posterioridad revivir o reconstruir el objeto, logrando que como programadores, no haya que preocuparse de cómo esta operación es llevada a cabo.

Así, en vista de lo anterior, la persistencia de los objetos es, como su nombre indica, el almacenamiento de los objetos.

Este hecho permite que siempre se mantengan disponibles para la aplicación, para que esta los pueda consultar, modificar, combinar, etc. Es decir, gestionarlos. La persistencia de los objetos debe conservar dicho objeto en el estado que el usuario considere interesante para sus objetivos.

Un objeto en sí mismo se compone de atributos, métodos y relaciones con otros objetos. Los atributos podríamos decir que son el equivalente a los registros o campos de una base de datos. Cada uno define los aspectos que componen un objeto, en definitiva todos aquellos aspectos que creemos que definen el objeto dentro de la realidad y que nos resulta imprescindible definir en nuestro programa.

Los atributos pueden ser de tipo primitivo, referenciado u otros objetos. Los métodos se pueden definir como funciones u operaciones que trabajan sobre los mismos atributos del objeto, o sobre relaciones con otros objetos. Las relaciones entre objetos pueden ser la herencia, de uso o asociación, parametrizada y la composición, entre otras.

Las bases de datos trabajan con registros. Los registros son las unidades básicas de trabajo de la información de las bases de datos relacionales. Estas unidades se clasifican en diferentes tipos, según la información: numéricos, caracteres, etc. Son los llamados tipos primitivos, y forman lo que denominamos modelo de datos.

Todos los lenguajes no orientados a objetos tienen un modelo de datos muy parecido a las bases de datos relacionales. Las variables son de tipo primitivo al igual que los registros de las bases de datos relacionales.

Una aplicación que trabaja con variables de tipos primitivos puede trabajar directamente y requiere de pocas adaptaciones con una base de datos relacional, pero como se puede entender fácilmente, una base de datos relacional cualquiera no puede almacenar un objeto a partir de registros. Puede llegar a tener un equivalente, a partir de los registros, parecido al objeto, pero nunca podrá almacenar todas sus propiedades, métodos y relaciones con otros objetos que lo definen, puesto que una base de datos relacional no está preparada. La base de datos relacional no tiene forma de almacenar de forma directa los métodos y las relaciones. Es preciso adaptar de alguna manera los modelos de datos.

El modelo de datos de objetos lo podemos describir cómo: Una colección de conceptos bien definidos matemáticamente que ayudan a expresar las propiedades estáticas y dinámicas de una aplicación con un uso de datos intensivo.

Conceptualmente, una aplicación quizás caracterizada por:

- Propiedades estáticas: entidades (u objetos), propiedades, (o atributos) de estas entidad, y relaciones entre estas entidades.
- Propiedades dinámicas: operaciones sobre entidades, sobre propiedades o relaciones entre operaciones.
- Reglas de integridad sobre entidades y las operaciones.

La diferencia de los modelos de datos hizo que los programadores, necesitados de encontrar la forma en que los objetos fuesen persistentes, adaptaran los modelos de datos de alguna manera. Se trataba que encontraran un modelo correspondiente o equivalente entre los dos modelos de datos: los lenguajes OO y las bases de datos.

Se desarrollaron varios sistemas que intentaban hacer una correspondencia entre los modelos de datos de las bases de datos relacionales y los objetos. Pero también aparecieron ideas que solucionaban el problema de la persistencia de forma más directo y automatizado, un sistema que consiguiera la correspondencia entre los modelos de datos: las capas de persistencia.

Una capa de persistencia robusta dirigida a bases de datos relacionales debe cumplir los siguientes requerimientos:

- Diversificación de los mecanismos de persistencia: Debe poder trabajar con varios sistemas almacenado de datos, no con uno sólo. Algunos podrían ser bases de datos, sistemas de ficheros, etc..
- Encapsulación cumplida del mecanismo de persistencia: El sistema sólo debe poder almacenar, eliminar y recibir los objetos. Del resto de operaciones se debe ocupar la capa de persistencia.
- Acciones sobre múltiples objetos: La capa debe consultar o eliminar múltiples objetos a la vez.
- Transacciones: Debe poder trabajar con acciones combinadas: consultar - borrar. También debe permitir parametrizar las transacciones. O sea realizar todas las operaciones propias del lenguaje SQL, en una base de datos.
- Identificadores de los objetos: Atributo numérico para cada objeto, que permita identificarlo respecto de otros totalmente iguales.
- Cursores: De sistema para poder recibir los resultados de las operaciones de la base de datos.
- Proxies: Aproximación completaría a los cursores. Es un objeto representativo de otro pero sin entrar en el propio objeto, sino realizando un tratamiento superficial del objeto que representa. Es como un apuntador sobre los datos de un objeto.
- Registros: Para generar informes de resultados.
- Múltiples arquitecturas: Debe poder adaptarse tanto a arquitecturas cliente/servidor de dos capas como de múltiples capas.
- Trabajar con diferentes bases de datos de diferentes fabricantes: La capa de persistencia debe poder adaptarse fácilmente sin tener que modificar la aplicación.
- Múltiples conexiones: Debe poder trabajar con sistemas que no tienen la base de datos de forma centralizada: sistemas SGBD descentralizados.
- Controladores nativos: Debe trabajar con los sistemas de acceso a las bases de datos más comunes, o en su defecto, con los proporcionados por los fabricantes.
- Consultas SQL.: Debe permitir hacer consultas SQL, no de forma obligada, sino como una excepción, y nos casos muy especiales.

Estas características, bien fijadas e implementadas, deben poder conformar una capa de persistencia consistente y fiable, de forma que un programador experimentado pueda confiar y escogerla ante varias opciones.

Todas estas características conforman la denominada transparencia. Esta característica tiene como objetivo entre otros no restarle ninguna característica ni a las aplicaciones generadas por el lenguaje OO ni a las bases de datos, aprovechando al máximo todas las propiedades y características, en beneficio del conjunto.

La transparencia de la capa permite que tanto la aplicación como la base de datos trabajen como si se comunicaran entre ellas de forma directa, configurando un modelo de datos común. Incluso si cambiáramos la base de datos, la aplicación no tiene porque verse afectada en modo alguno, debiendo continuar trabajando del mismo modo. La transparencia se encarga de volver a conexionar los diferentes modelos de datos de forma automatizada, sin restar rendimiento y propiedades. Cómo podemos ver la transparencia respeta la independencia a las dos partes a la

vez que permite que trabajen de forma conjunta como un sistema único. Este es el motivo por el que la persistencia por capas también se denomina persistencia transparente.

5.3 Evaluación de alternativas existentes

5.3.1 Java Persistence API (JPA)

El Java Persistence API (JPA) es una especificación de Sun Microsystems para la persistencia de objetos Java a cualquier base de datos relacional. Esta API fue desarrollada para la plataforma JEE e incluida en el estándar de EJB 3.0, formando parte de la Java Specification Request JSR 220.

Para su utilización, JPA requiere de J2SE 1.5 (también conocida como Java 5) o superior, ya que hace uso intensivo de las nuevas características de lenguaje Java, como las anotaciones y los genéricos. Este documento ofrece un panorama general de JPA. A menos que se indique otra cosa, la información presentada se aplica a todas las implementaciones de JPA.

5.3.2 HIBERNATE

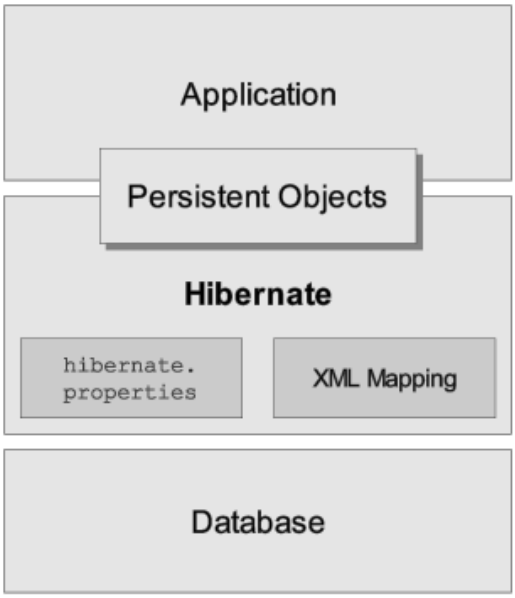
Hibernate no sólamente se ocupa del mapeo desde las clases Java a las tablas de las bases de datos (y desde los tipos de datos de Java a los tipos de datos de SQL), sino que también facilita la consulta y recuperación de datos. Esto puede reducir de manera importante el tiempo de desarrollo que se tomaría con el manejo de datos de forma manual en SQL y JDBC.

Hibernate busca solucionar el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación: el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional). Para lograr esto permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información Hibernate le permite a la aplicación manipular los datos de la base operando sobre objetos, con todas las características de la POO. Hibernate convertirá los datos entre los tipos utilizados por Java y los definidos por SQL. Hibernate genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para poder adaptarse a su uso sobre una base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible.

Hibernate ofrece también un lenguaje de consulta de datos llamado HQL (Hibernate Query Language), al mismo tiempo que una API para construir las consultas programáticamente (conocida como "criteria").

Hibernate para Java puede ser utilizado en aplicaciones Java independientes o en aplicaciones Java EE, mediante el componente Hibernate Annotations que implementa el estándar JPA, que es parte de esta plataforma.



5.3.3 Oracle

• Motor de persistencia para POJO (Plain Old Java Objects) ejb 2.1, cmp y bmp.

• Implementa la api JPA enfocada a la estandarización del la persistencia objetorelacional.

• Permite persistencia de objetos Java en bases de datos relacionales accesibles utilizando los drivers JDBC

• También permite la persistencia de objetos java en bases de datos objeto relacional como las bases de datos de Oracle.

• Soporta Enterprise information system (EIS); permitiendo la persistencia de objetos java a fuentes de datos no relacionales que utilizan la arquitectura J2C (J2EE Connector architecture adapter).

• Conversión entre objetos Java y documentos del esquema XML (XSD), usando la arquitectura java para XML (JAXB)

• Soporte para servidores de aplicaciones IBM WebSphere y BEA WebLogic.

• Soporta el cache de objetos para mejorar el rendimiento

5.3.4 EJB (Enterprise Java Beans)

Los EJB proporcionan un modelo de componentes distribuido estándar del lado del servidor. El objetivo de los EJB es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad, etc.) para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes permite que éstos sean flexibles y sobre todo reutilizables.

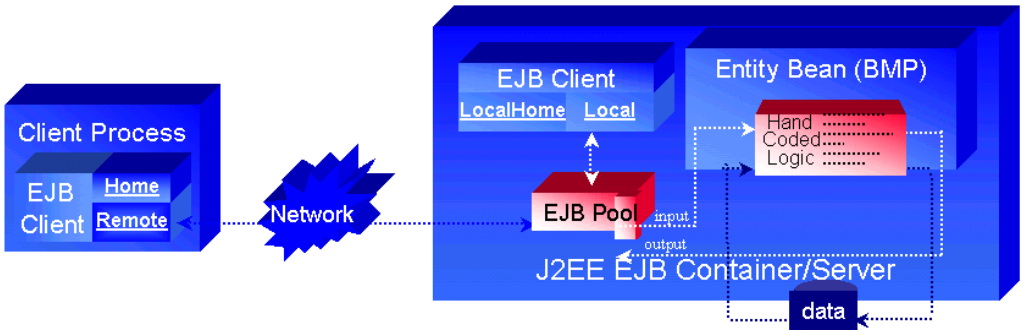
Dentro de los EJB, hay dos maneras diferentes de gestionar la persistencia de objetos:

• El desarrollador especifica el comportamiento de la persistencia, estaremos hablando de EJB de entidad con persistencia gestionada por el componente (BMP, Bean Managed Persistence).

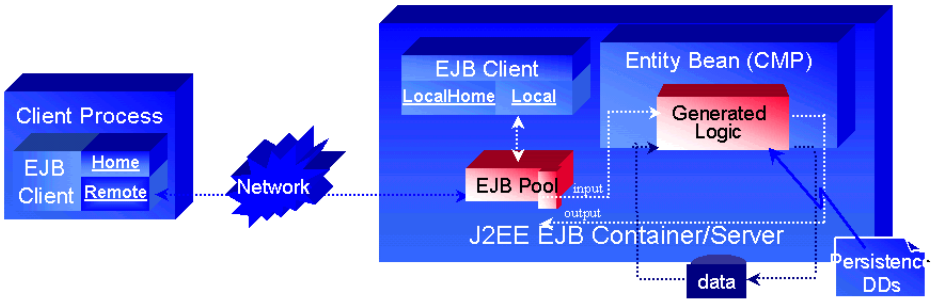
• El contenedor especifica el comportamiento de la persistencia, estaremos hablando de EJB de entidad con persistencia gestionada por el contenedor (CMP, Container Managed Persistence).

En los BMP (EJB de entidad con persistencia gestionada por el componente), el desarrollador del componente es la persona que debe implementar la persistencia de dicho componente. Esto quiere

decir que el desarrollador debe especificar cada uno de los diferentes comportamientos que tendrá el componente en la comunicación con la base de datos relacional. Este tipo de EJB es mucho más costoso para el desarrollador y puede llevar a un número mayor de errores de persistencia.



La función de un bean CMP es muy similar a la de JDO. Ambas tecnologías permiten persistir datos con una perspectiva orientada a objetos ya que siempre se persisten objetos y evitan tener que conocer los detalles de cómo los objetos están almacenados. JDO y CMP también son capaces de manejar relaciones entre objetos. Por tanto sí se puede hablar de que JDO puede sustituir a CMP.



Se debe tener en cuenta el crecimiento de las tecnologías. Los beans CMP todavía necesitan aumentar su aceptación en la comunidad de programadores. La mayor parte de las mejoras y crecimiento en general de las EJBs ha sido en el área de sesión. CMP y JDO padecen los mismos problemas a la hora de ser acogidos ya que ambas tecnologías abstraen demasiado al programador de lo que realmente sucede a nivel SQL. A la hora de realizar consultas complicadas el programador debe dedicar mucho tiempo intentando descubrir cómo generar dicha consulta equivalente a la sentencia SQL. En estos casos los programadores preferirían haber programado en SQL desde un primer momento.

5.3.5 JPOX Java Persistent Objects

JPOX es una puesta en práctica de los objetos de los datos de OpenSource Java (JDO) que proporciona la persistencia transparente de los objetos de Java. JPOX pone las especificaciones en ejecución JDO1 y JDO2 y pasa los boths TCKs. JPOX apoyan datastores de RDBMS. JPOX está poniendo JPA en ejecución 1 (EJB 3).

Con una puesta en funcionamiento versátil y de gran rendimiento, JPOX está sobre la vanguardia de las puestas en funcionamiento de objetos (JDO) de datos de Java disponibles, ofreciendo uno JDO dócil gratis, que se puso en funcionamiento liberada bajo una licencia Open Source. JPOX 1.0 es completamente dependiente de JDO 1.01, y JPOX 1.1 tiene muchas características previas de JDO 2.0.

5.3.6 Smyle

Smyle suministra un enfoque de innovación y pragmático para almacenamiento de datos fácil de comprender, fácil de uso, pero lo suficientemente fuerte para aplicaciones del mundo real. Smyle es Open Source y bajo licencia LGPL.

5.3.7 Speedo

Speedo es una implementación opensource de JDO desarrollada por el consorcio ObjectWeb.

Algunas de sus principales características son:

• Posibilidad de elegir entre transacciones pesimistas u optimistas.

• Caché de instancias, con posibilidad de elegir algoritmos de gestión (LRU | MRU | FIFO | ...)

• Obtención previa de datos al evaluar las consultas (cacheado)

• Acceso a bases de datos relacionales heredadas mediante JORM y MEDOR, así como otros almacenes de datos.

5.3.8 IBATIS

Ibatis es un framework de persistencia con tecnología open-source basado JDBC.

La principal diferencia entre iBatis e Hibernate es que iBatis no es un ORM (Mapeo Objeto Relacional). Por lo tanto Hibernate genera el SQL para mapear objetos a tablas de la base de datos, mientras que en iBatis el SQL lo tendremos que escribir nosotros. Si tenemos un modelo de datos (nuestras tablas de BBDD) bastante diferente a nuestro modelo de negocio (nuestras clases Java) Hibernate nos puede dar bastantes dolores de cabeza.

Sus caracteristicas:

- Ejecuta el SQL escrito por nosotros mediante JDBC (Java DataBase Connectivity), por lo que nos olvidamos de los múltiples try/catch.
- Mapea propiedades de objetos a parámetros para las PreparedStatement (sentencias SQL parametrizables).
- Mapea los resultados de una query a un objeto o una lista de objetos.

Este framework permite relacionar un bean de java con una sentencia SQL mediante la configuración de archivos xml. En estos ficheros xml, se crean consultas complejas que son adaptadas al sistema de gestión de base de datos que estemos utilizando en el proyecto. Esto supone un gran inconveniente, ya que IBATIS no es independiente al proveedor de Base de Datos, de forma que si se cambiase este, habría que editar todas las sentencias definidas en los ficheros xml.

5.3.9 Torque

Torque es una capa de persistencia, las herramientas que incorpora permiten crear un modelo físico de datos adaptado a diferentes SGBD gracias al empleo de XML.

Además permite evitar la codificación directa de sqls, así como abstraer al programador de la problemática vinculada con la apertura y cierre de conexiones a BBDD.

Es una herramienta muy potente que se debe usar con sumo cuidado, puesto que si se usa de manera inadecuada puede evitar los inconvenientes del acceso a BBDD a costa de introducir nuevos problemas.

El Torque incluye un generador para generar todos los recursos de base de datos requeridos por su aplicación e incluye un ambiente de tiempo de ejecución para dirigir las clases generadas. El IDE en tiempo de ejecución del Torque incluye todo que usted necesita para usar las clases de OM / Peer generadas. Incluye un pool de conexión con JDBC.

5.3.10 JDBCPersistence

JDBCPersistence es Frameworks de mapeo de persistencia O/R. Es diferente de sus semejantes en lo que respecta a genera el bytecode requerido para mapear una clase en una tabla. Ha sido creado con los siguientes requisitos en mente:

• Ser rápido para cargar

• Soporte para CLOBs y BLOBs

• Cargar objetos persistentes desde java.sql.ResultSet

• Tenga API compacto

• Tenga dependencias mínimas sobre otros proyectos

• Configuración de soporte vía API

5.3.11 Java Ultra-Lite Persistence (JULP)

Un Frameworks de muy pequeño tamaño.

• Puede funcionar tanto en modo containes y stand-alone

• Ocupa poco espacio (menos de 50 KB)

• No hay dependencias en las bibliotecas, sólo JDK (necesita driver JDBC)

• Independientes de base de datos (probado hasta ahora con Oracle 10g y MySQL)

• Herencia

• Muchas clases por tabla

• Muchas tablas por clase

• Asignaciones simples: la asignación lo único que necesita es catálogo>> <schema> table.column = fieldName.

5.3.12 JDO (Java Data Object)

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. Hay muchas implementaciones de JDO entre las que elegir. La implementación de referencia de JDO la ha puesto Sun a nuestra disposición.

Una razón de la simplicidad de JDO es que permite trabajar con objetos normales de Java (POJOs plain old Java objects) 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 abstacción 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.

No exige la implementación de interfaces de desarrollo. La configuración de este framework se realiza mediante un XML. JDO permite trabajar con objetos normales de Java a la vez que los hace persistentes de forma que requiere de una menor estructura.

5.4 Evaluación comparativa

5.4.1 JDO vs. EJB

Desde la aparición de JDO se ha especulado que esta tecnología podría sustituir a las EJB de entidad.

Para entender esta afirmación se debe examinar que es exactamente un EJB de entidad. Como ya se ha explicado, los Beans de entidad se dividen en dos categorías: persistencia manejada por contenedor (CMP) y persistencia manejada por Bean (BMP). Los Beans BMP contienen un código que puede almacenar el contenido del Bean en un almacén de datos permanente.

Los BMP tienden a ser independientes y no forman relaciones directas con otros Beans BMP. No sería correcto decir que los Beans BMP pueden ser sustituidos por JDO, puesto que un Bean BMP hace uso directo de código JDBC.

Esto viola uno de los principios de diseño de JDO ya que esta tecnología pretende abstraer al usuario de codificar con JDBC.

Los Beans CMP permiten manejar la persistencia al contenedor. El contenedor es cualquier servidor que esté ejecutando el Bean, y se encarga de manejar todo el almacenamiento actual. Los Beans CMP también pueden formar las típicas relaciones [1 – n] ó [n a m] con otros Beans CMP.

La función de un Bean CMP es muy similar a la de JDO. Ambas tecnologías permiten persistir datos con una perspectiva orientada a objetos ya que siempre se persisten objetos y evitan tener que conocer los detalles de cómo los objetos están almacenados. JDO y CMP también son capaces de manejar relaciones entre objetos. Por tanto sí se puede hablar de que JDO puede sustituir a CMP.

Se debe tener en cuenta el crecimiento de las tecnologías. Los Beans CMP todavía necesitan aumentar su aceptación en la comunidad de programadores. La mayor parte de las mejoras y crecimiento en general de las EJBs ha sido en el área de sesión. CMP y JDO padecen los mismos problemas a la hora de ser acogidos ya que ambas tecnologías abstraen demasiado al programador de lo que realmente sucede a nivel SQL. A la hora de realizar consultas complicadas el programador debe dedicar mucho tiempo intentando descubrir cómo generar dicha consulta equivalente a la sentencia SQL. En estos casos los programadores preferirían haber programado en SQL desde un primer momento.

5.4.2 JDO vs. JDBC/SQL

Sustituir a JDBC no es exactamente lo que busca JDO de la misma forma que podría hacerlo con CMP. JDO puede ser realmente una buena capa en el nivel superior a JDBC. En la mayoría de las instancias de JDO, se debe especificar una fuente de datos JDBC que establezca una referencia a la base de datos que JDO va a estar manejando. Por tanto, comparar a JDO con JDBC es algo que se debe hacer si se duda entre usar directamente JDBC o permitir que JDO lo use en lugar del programador.

Por una parte JDO libera al programador de la tarea de construir consultas SQL y de distribuir entre los atributos de un objeto Java los resultados obtenidos en un result set. Si se considera que la mayor parte de consultas JDBC se realizan con el fin de dar valores a los atributos de objetos Java, JDO debería ser una alternativa a tener en cuenta, puesto que en lugar de ejecutar una consulta y

copiar los campos desde el result set de JDBC a objetos Java, JDO se puede hacer cargo de todo ello.

Las críticas a JDO vienen precisamente por las grandes cantidades de accesos innecesarios que realiza para llevar a cabo su tarea. JDO debe coger su consulta JDOQL y convertirla en su consulta SQL correspondiente. Entonces, esta consulta SQL es enviada a la base de datos, los resultados son recibidos y almacenados en sus respectivos objetos. Si hubiera un gran número de relaciones entre los objetos, es muy fácil que, como consecuencia, JDO haya accedido a muchos más datos de los necesarios. Es evidente que JDBC siempre va a ser más rápido que JDO ya que es más directo. Será elección del programador si la comodidad en la forma de trabajar que le ofrece JDO compensa su peor rendimiento.

Otro de los aspectos discutidos de JDO es la posibilidad de sustituir SQL por JDOQL. Cuando se usa JDBC se debe acudir a SQL para componer las consultas mientras que con JDO se usa JDOQL. JDOQL es un lenguaje de consultas basado en Java. Por una parte, JDOQL es mucho más sencillo de componer que SQL, especialmente cuando nos referimos a consultas sencillas. Sin embargo, no existen muchos programadores que dominen JDOQL.

De momento, este problema va a seguir existiendo ya que, como se ha comentado anteriormente, JDO no ha sido muy acogido en la industria del desarrollo de software. La mayoría de los programadores dominan SQL, y además son capaces de construir consultas SQL muy optimizadas a pesar de tener una gran complejidad.

Para muchos programadores, una herramienta que crea consultas automáticamente no es de gran utilidad, sobre todo si nos referimos a JDOQL, que solamente actúa en aplicaciones Java. Antes o después SQL será sustituido, pero para ello tendrá que llevarlo a cabo una tecnología más universal.

5.4.3 Hibernate vs. JDO

Hibernate se caracteriza por su completa transparencia para el programador. Al contrario que JDO, Hibernate se encarga de todo el proceso de persistencia. No hay que pasar por la tarea de ejecutar nada parecido al JDOEnhancer. Hibernate se encarga de hacer todo el proceso transparente, ya que basta con añadir sus librerías a la aplicación y rellenar su archivo de configuración para asignar la base de datos con la que se va a trabajar. Una vez dispuestos los ficheros de mapeo, se puede trabajar con la misma facilidad con la que se codifica con cualquier librería Java.

Otro punto muy a su favor es que Hibernate mantiene la posibilidad de realizar sus consultas en SQL.

El HQL es el lenguaje para consulta específico de Hibernate, al igual que el JDOQL es el lenguaje a través del cual se realizan las consultas cuando se trabaja con JDO. Para usar JDO, el JDOQL es indispensable, ofreciendo una gran comodidad al programador a la hora de componer consultas sencillas. Sin embargo, cuando se trata de realizar sentencias más complejas, un programador que domine SQL con un nivel alto seguramente eche de menos la alternativa estándar. Hibernate ofrece ambas posibilidades.

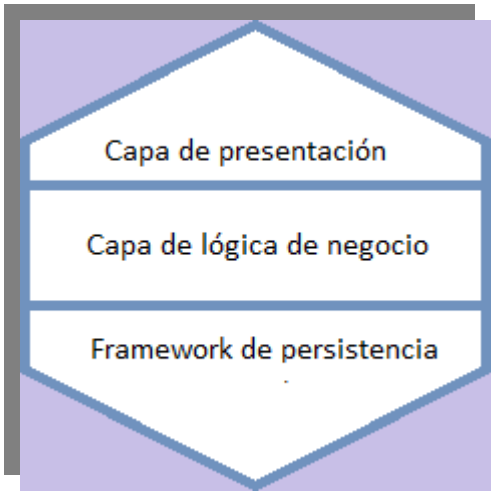
6. Diseño del framework de persistencia

6.1 Introducción

Actualmente, las bases de datos relacionales son la solución más utilizada por los sistemas empresariales con el fin de almacenar gran volumen de datos. Como contrapartida, éste tipo de sistemas poseen una lógica de negocios que suele recurrir al paradigma orientado a objetos como marco para modelar y resolver su complejidad inherente. Esto genera un problema. El mundo de las bases de datos y los objetos no siempre se llevan tan bien como uno deseara. Esto requiere un esfuerzo importante para compatibilizarlos. Un framework de persistencia transporta objetos en memoria desde y hacia un almacenamiento permanente, siendo una base de datos relacional el medio más utilizado. El framework maneja el mapeo de los objetos contra la base de datos, abstrae al desarrollador del SQL y resuelve diversos temas como el manejo de concurrencia. Existen numerosos frameworks de persistencia tanto open source como comerciales.
Básicamente el objetivo de este framework de persistencia es ofrecer al usuario un interfaz de interacción con una base de datos relacional, sin que este tenga que hacer modificaciones, únicamente indicando el tipo de base de datos a la que se accederá.

6.2 Características

- Desarrollado en base a un modelo de tres capas:



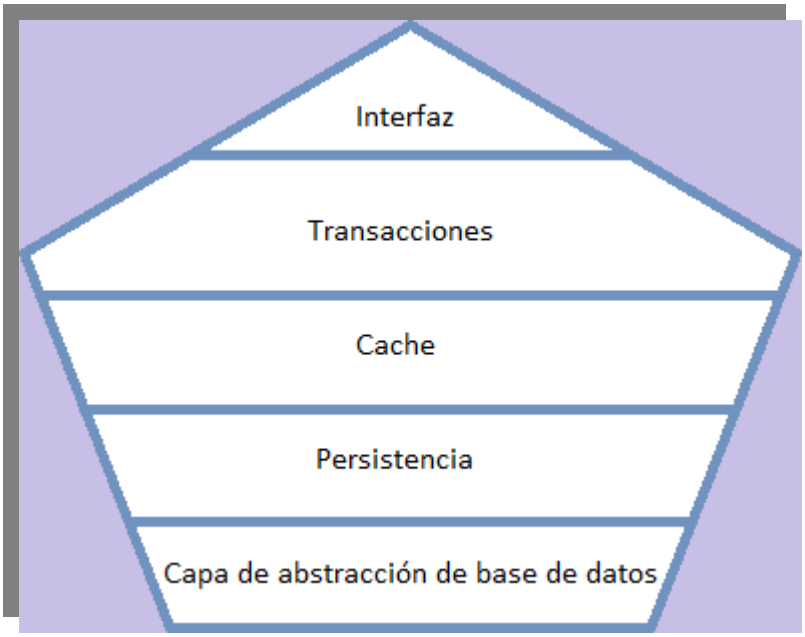
  Hemos optado por realizar un modelo de negocios complejo, aprovechando todas las ventajas que nos provee la programación orientada a objetos y los patrones de diseño. A los objetos de esta capa, responsable de resolver la problemática para la cual el sistema fue construido, los llamaremos objetos de negocio. Esta es una organización conceptual donde la lógica de negocio está aislada de la capa de persistencia desconociendo la existencia de la misma.
  En muestro desarrollo tendremos un “mapeador” (ORM – “Object Relational Mapping”) que mediará entre las dos capas de modo que ambas se mantengan independientes entre sí.
- Se deberá de poder ejecutar tanto en modo contenedor como de forma autónoma.
- El código deberá ser contenido, es decir, no deberá ser de gran tamaño, tipo Hibernate.
- Debe ser independiente de otros códigos, librerías o plataformas, excepto que será dependiente de JDK, es decir, que necesita del driver JDBC para poder funcionar.
- También será independiente de la plataforma SGBD, es decir, que no se desarrolla para ser ejecutado en una plataforma dependiente (en nuestro caso MySQL y ORACLE).
- El requerimiento mínimo será de uso de JDBC 1.X o superior.

- Soportará la herencia directamente.
- Deberá poder soportar la implementación de varias clases por cada tabla de la BBDD.
- También deberá poder utilizar varias tablas del SGBD por cada clase implementada.
- Debe destacar por su sencillez a la hora de realizar el mapeo entre la base de datos y nuestro framework.

- Realizaremos el mapeo mediante fichero de propiedades.

6.3 Arquitectura

La arquitectura del framework de persistencia propuesto se organiza en distintos capas o “layers”, cada uno con una responsabilidad bien definida.



Como trabajamos con un modelo de dominio, es muy común implementar un capa superior, es decir, un objeto base de quien hereden todos nuestros objetos de negocio. Una de las primeras decisiones importantes es si el framework de persistencia nos proveerá de una para todos nuestros objetos de negocio persistentes: una clase base que podremos llamar “Objeto De Negocio Persistente”. En esta clase encontraremos ciertos servicios propios del framework que pueden abarcar identificación, recuperación y métodos para saber el estado de persistencia del objeto. Desde un punto de vista teórico esta clase base nos genera cierto grado acople entre nuestro modelo de negocio y el framework de persistencia.
Es de vital importancia que el framework y los objetos de negocio estén lo más separado posible. De esta forma la capa de negocios quedará englobada en un paquete donde sólo existirá una interfaz o clase abstracta que defina los servicios requeridos. Otro paquete distinto será el que contendrá la implementación.

6.3.1 Interfaz

Esta es la capa visible del framework con la que interactúan los usuarios del mismo y por lo tanto provee todos los servicios necesarios para hacerlo, tratando de ocultar, en la medida de lo posible, los detalles de implementación. Sin quitar la posibilidad de proveer un conjunto de métodos que permitan parametrizar de alguna forma el funcionamiento. Esta capa internamente implementa la traducción de excepciones de las capas inferiores, adaptación de los mensajes de error y soporte de logging.

6.3.2 Transacciones

El objetivo de esta capa es coordinar la escritura de los datos resolviendo problemas de concurrencia. Cada transacción mantiene internamente un conjunto de colecciones:

- Los objetos nuevos son objetos nuevos a ser insertados en la base de datos.
- Los objetos a eliminar van a ser eliminados de la base de datos. La eliminación puede tratarse tanto de una baja física (mediante un delete) o una baja lógica mediante la actualización de algún campo de estado.
- La colección de los objetos sucios contiene objetos preexistentes. Al momento de hacer commit de la transacción se deberá actualizar su representación en la base de datos.

Cada transacción representa una unidad de trabajo y soportan como mínimo las operaciones inicio de transacción, “commit” y “rollback”. Se usan para iniciar una transacción, confirmar los cambios o deshacerlos respectivamente. Dentro del framework todo objeto a ser persistido debe estar dentro de una transacción y los cambios sobre los objetos de una transacción se realizan todos o no se realiza ninguno tratando de respetar las propiedades ACID (atomicidad, consistencia, aislamiento y durabilidad) de las transacciones de la bases de datos.
Existe una relación entre una transacción del framework de persistencia y una transacción de la base de datos, aunque puede no haber relación directa entre sus operaciones. Las transacciones de objetos pueden mantenerse abiertas sin ser confirmadas durante un período de tiempo mayor (en la base de datos esto no es recomendable).
El commit de una transacción del framework puede disparar un inicio de transacción, ejecutar un conjunto de sentencias SQL y realizar commit contra la base de datos.
Por otra parte el rollback de la transacción del framework simplemente opera internamente a nivel framework descartando los cambios en memoria sin necesidad de realizar ninguna notificación a la base de datos.
Podremos definir dos tipos de transacciones de commit activado y solo lectura. Este último, aplica cuando se recuperan un conjunto de objetos pero no se permite realizar (o se ignoran) operaciones que modifiquen el estado de los mismos.
Existe otro tipo de clasificación posible sobre las transacciones: simultáneas y anidadas. Las simultáneas son transacciones que se ejecutan en forma paralela e independiente. Cada una actúa sobre un conjunto de objetos distintos. Por un motivo de seguridad se prohíbe que dos transacciones simultáneas modifiquen el mismo objeto. Las transacciones anidadas prevén los casos donde una operación se puede realizar en una secuencia finita de pasos donde cada uno se puede confirmar en forma independiente. Una transacción puede lanzar subtransacciones anidadas y de esta forma implementar un mecanismo de puntos de retorno en memoria. Los cambios en la base se reflejarán en el momento que se ejecute commit en la transacción de nivel superior. Implementar transacciones anidadas puede llegar a ser muy útil si se está dispuesto a pagar el coste en complejidad y tiempo que implica su implementación. Cada transacción anidada debe proveer un mecanismo para serializar o guardarse de alguna manera el estado de los objetos de la transacción al momento de iniciar la misma, de modo que al hacer rollback de una transacción anidada el estado de los objetos sea el mismo que tenían antes de comenzarla. En el caso de no disponer de transacciones anidadas pero necesitar de dicha funcionalidad siempre tenemos la opción de implementarlas a nivel objeto de negocio.
Coordinar la escritura de los datos es otra de las responsabilidades de las transacciones. Esto implica determinar el orden en el que se realizarán los INSERTS para que no haya problemas con

las limitaciones/constraints definidas en la base de datos. Hay motores que permiten verificar las constraints de la base al final de cada transacción.

6.3.3 Cache

La principal tarea de la cache es asegurarse que no exista duplicidad en la representación de un mismo objeto en memoria. Si se recupera un objeto con un ID determinado y se vuelve a recuperar luego el mismo objeto, debemos tener cuidado de no instanciar dos objetos con la misma información. Conceptualmente son el mismo objeto pero si se realizara una modificación sobre alguno no se verían los cambios realizados sobre el otro. Al momento de actualizarlos en la base de datos esto puede resultar en actualizaciones perdidas.
Otra funcionalidad que puede ofrecernos la cache es la posibilidad de bloquear objetos por parte de las capas superiores, de este forma podemos evitar que dos transacciones simultáneas modifiquen el mismo objeto. Por último la cache puede servirnos para mejorar el rendimiento de la aplicación manteniendo los objetos en memoria. Para ello a cada objeto en la cache se le coloca un "Timestamp" (tipo de dato, que detalla fecha, hora y milésimas de segundo) que indica cuando fue accedido por última vez. Periódicamente un hilo se encarga de recorrer la cache y buscar aquellos objetos que no han sido accedidos en un período de tiempo, de esta manera podemos liberar la memoria o le asignamos un valor nulo a la referencia.
Si todo el sistema interactúa con la base de datos mediante el framework de persistencia y además no existen otros sistemas externos que acceden a la base de datos, esta opción resulta interesante. No obstante si la base de datos se accede concurrentemente por más de una aplicación se recomienda eliminar la cache como mecanismo de mejora de rendimiento e incluir algún sistema de concurrencia como explicaremos más adelante.

6.3.4 Persistencia

Esta capa mapéa los objetos contra la base de datos y viceversa. Para ellos conoce una serie de mapeos que describen como se realiza. Los mapeos pueden ser descritos de tres formas distintas:

- Mediante archivos externos, siendo el formato XML el más adoptado para tal fin.
- Mediante anotaciones o atributos colocados dentro de la fuente de los objetos como en el caso de los atributos en C# o las anotaciones en Java.
- Codificados manualmente en el mismo lenguaje de programación.

Describir los mapeos en archivos auxiliares es la estrategia más flexible, aunque se dificulta hacer chequeos de sintaxis y encontrar errores de escritura como por ejemplo el nombre de un campo de la base de datos mal escrito. Estos errores suelen aparecer frecuentemente en tiempo de ejecución y suelen controlarse mediante el uso de las pruebas unitarias. Describir los mapeos mediante anotaciones o codificarlos a mano conducen también al mismo problema salvo que se use un diccionario de dato (una generación de una jerarquía de clases que replique el esquema de la base de datos mediante alguna estrategia de generación de código para poder realizar verificaciones de nombres en tiempo de compilación dentro el entorno de desarrollo).
La información de los recibe el nombre de metadatos. Cada clase tendrá su metadato asociado. Los mapeos más comunes con los que nos podemos encontrar son a nivel clase son:

- El mapeo de tabla, define las tablas sobre la que persistirán los datos de una clase debiendo especificársela como parte del mapeo cual es el campo ID.
- El mapeo de subclases, las alternativas para persistir una jerarquía son Single Table Inheritance, ClassTable Inheritance, Concrete Table Inheritance.
- El SelectMapping, define una consulta sql que se utilizara para obtener los datos. Dada la naturaleza de este mapeo generalmente se trata de objetos de solo lectura salvo que por medio de un CustomClassMapper se le especifique como se persistirán los datos.

- El CustomClassMapper define el nombre de una clase que respeta una interfaz definida por el framework en la cual se codifica en forma manual un DataMapper que especifica la forma de persistir y recuperar los objetos desde y hacia la base de datos

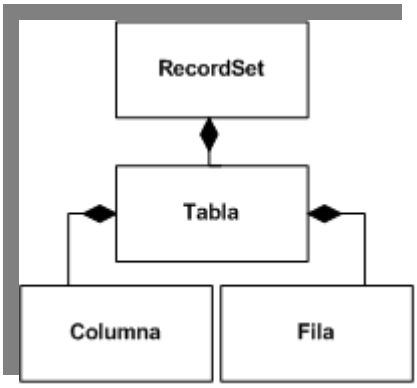
Los mapeos que actúan también a nivel propiedad que son:

- El CommonFieldMap mapéa una propiedad del objeto contra un campo de la base de datos debiendo especificarse el nombre de ambos.
- TransformFieldMap permite mapear un campo de la base contra una propiedad del objeto realizando ciertas transformaciones en el medio.
- OneToOneMapping es el encargado de mapear una relación uno a uno entre los objetos debiendo realizar las consiguientes transformaciones de claves foráneas en el mundo de la bases de datos a referencias en el mundo de los objetos.
- OneToManyMapping corresponde al mapeo uno a muchos donde hay en el medio una colección de objetos involucrados.
- AggregateMapping son objetos simples y pequeños, como por ejemplo un rango de fechas y generalmente no tienen ID. Su ciclo de vida es generalmente controlado por el objeto contenedor en el sentido que solo tiene sentido que existan si son contenidos por alguien (o para algún cálculo interno).
- ConditionalAggregate es un caso extendido del AggregateMapping donde un campo de la base de datos especifica la subclase concreta del AggregateMapping. Este mapeo resulta sumamente útil para mapear un patrón.
- FieldToQueryMap, mapea una propiedad de solo lectura del objeto contra una consulta a la base de datos que será de sólo lectura y no intervendrá en ningún momento en la actualización del objeto. Además de los mapeos, los metadatos de los objetos pueden marcar a las propiedades como mutables o no mutables asumiéndose alguna de ambas posibilidades por defecto. Una propiedad no mutable asegura que el código de su getter asociado no modifica la representación interna.

6.3.5 Capa de abstracción de base de datos (DAL - Database Abstraction Layer)

Los distintos entornos de desarrollos y lenguajes de programación traen frameworks y librerías para permitir la interacción con la base de datos. Estas pueden variar según la base de datos. El objetivo de esta capa es ofrecer una interfaz común y conocida para comunicarse con la base de datos. Debe ser capaz de abstraernos de las pequeñas diferencias propias de cada motor, por ejemplo en las variaciones de la sintaxis de las consultas SQL.
Esta capa recibirá implementaciones alternativas según la base de datos que utilicemos. Por lo tanto en su diseño es importante pensarla en términos de clase e interfaces abstractas que abstraigan esas diferencias. Las subclases pueden variar según la base de datos que utilicemos. Dichas implementación deberán ser intercambiables y se debe prever alguna forma de configurar cual se utilizará. Ponemos esta capa alrededor de tres formas no excluyentes.

- Una es utilizar un objeto que reciba una cadena de caracteres con la sentencia SQL y la ejecute contra el motor de la base de datos.
- El otro camino es generar una jerarquía de objetos que representen una sentencia SQL. Tendríamos el objeto Select, Update, Insert y Delete que tienen propiedades como las tablas y los campos para generar las sentencias SQL mediante estos objetos en lugar de trabajar sobre una cadena de caracteres.
- Por último nos queda la opción de los recordsets. Un recordset es una representación en memoria que replica la estructura de la base de datos. Organiza la información en forma tabular en tablas, filas y columnas. Esta opción puede ser recomendable cuando el entorno provee muchas funcionalidades alrededor del recordset, como por ejemplo la generación automática de las sentencias de SQL. La desventaja es que de esta manera el framework nos queda más dependiente de un lenguaje en particular.



6.3.5.1 Concurrencia

La concurrencia es uno de los problemas más complejos a considerar en el diseño de un framework de persistencia. Dentro de un mismo proceso debemos asegurarnos que no haya dos transacciones actuando sobre el mismo objeto. La cache nos parece un punto adecuado para permitir bloquear objetos. Si nuestro sistema, a través del framework de persistencia, es el único que interactúa con la base de datos iremos bien. Si por otra parte, como mencionamos antes, subimos un nivel y hablamos de concurrencia entre distintos procesos la cosa se suele complicar un poco más. Este es el caso de varias aplicaciones distintas que acceden a la misma base o incluso en una aplicación cliente-servidor donde puede haber varios clientes conectados contra la misma base. Es aquí donde se debe tomar una decisión sobre una estrategia de concurrencia optimista, pesimista o ninguna. Cuando hablamos de concurrencia a este nivel lo que se busca es evitar el fenómeno de updates perdidos donde una aplicación recupera un objeto y mientras está operando con él en memoria una segunda aplicación interviene, recupera el mismo objeto y lo actualiza antes de que la primera aplicación confirme sus cambios. Cuando la primera aplicación en efecto lo haga sobrescribirá los cambios de la segunda, dando como resultado un update perdido. La estrategia de concurrencia más adecuada puede variar según el caso y según el objeto de negocio que se considere. Por este motivo es necesario poder definirla para cada objeto en tiempo de configuración. Esta información se puede especificar dentro de los metadatos de cada clase. Es posible el caso se juzgue que para un objeto determinado la posibilidad de que ocurra un update perdido es muy baja y en el caso de que ocurra no ocasione ningún impacto negativo en el negocio. En ese caso es aceptable no tomar ninguna estrategia siempre y cuando ese juicio sea acertado. La estrategia optimista suele ser una alternativa viable para prevenir conflictos entre transacciones concurrentes detectando el conflicto y haciendo rollback de la transacción. Su implementación puede requerir el agregado de un campo timestamp a la tabla de la base de datos que indique el instante de la última modificación cada registro.
Agregar el timestamp a cada tabla de la base de datos resulta una tarea bastante tediosa por lo que muchas veces se opta por crear una tabla aparte que contenga (Nombre de la tabla, ID del registro, el timestamp). Si esta última opción tampoco resulta adecuada la alternativa es en la sentencia SQL update especificar en el where que el valor de cada uno de los campos coincida con el que tenía antes de realizar las modificaciones. Esta opción no requiere el agregado de tablas adicionales pero puede hacer que los updates sean un poco más lentos y requiere la tarea adicional de almacenar los valores que se tenían originalmente al momento del recuperar el objeto.
La persistencia pesimista es un poco más compleja porque requiere un trabajo en conjunto. Dos aplicaciones no pueden tomar simultáneamente el mismo registro de la base de datos. Si un registro esta cogido se debe esperar que la aplicación que lo bloqueó lo libere. Algunos motores de base de datos proveen alguna forma de implementar una persistencia pesimista. La ventaja de que lo implemente el motor es que si una aplicación falla automáticamente el registro es liberado aunque cada base de datos lo implementa de una forma distinta. Si no hay que hacerlo manualmente con algún campo o tabla donde se marquen los registros tomados. Esto requiere una tarea conjunta entre las distintas aplicaciones que acceden a la base de datos y puede hacer que queden objetos codidos que nunca se liberan si la aplicación que los tomos sufre un desperfecto

que le impide liberarlo. Esto se subsana con un tiempo máximo durante el cual se puede reservar un objeto aunque es un problema complejo de considerar.

6.3.5.2 Identificación

Al considerar recuperar objetos con el framework de persistencia, el primer caso a tener en cuenta es recuperar un objeto en particular. Este escenario surge de la necesidad de disponer de un mecanismo para poder identificar unívocamente cada objeto de nuestro sistema. La alternativa más común es asignar a cada objeto un ID sin significado de negocios. Esta estrategia nos provee varias ventajas ya que proporciona una forma genérica de identificar un objeto independiente del problema de negocios a resolver. Para soportarla, en la base de datos debe existir un campo para mapear contra ese ID que convenientemente se recomienda que coincida con la clave primaria de la tabla y por consiguiente estar indexada. Si se dispone de un capa superior para todos los objetos de negocio y el mismo sabe de la existencia del framework de persistencia suele resultar cómodo incluir un método estático encargado de recuperar un objeto de una clase de la forma:

```
Cliente c = (Cliente) Cliente.Recuperar(1);
```

Esta forma resulta bastante intuitiva para un desarrollador que desee recuperar el objeto de la clase cliente con el ID número 1. Cuando hablamos de campos ID entran en juegos varias decisiones de diseño. Primero si haremos un ID único por tabla o un ID único para todo el sistema.
Para el primer caso la alternativa de los campos autoincreméntales nos proporciona una solución dependiente de la BDD que utilicemos además debe proveer un mecanismo para obtener el ID del registro a insertar para resolver problemas de claves foráneas.
Si en vez de los campos auto incrementales utilizamos alguna consulta especial de la forma “select max ID) from tabla” debemos prestar atención de que dicha consulta bloqueará la tabla durante lo que dure la transacción.
La tercer opción es disponer de una tabla adicional común que nos provea de la información del último registro que se ha introducido para una tabla.
Para el caso de tener un ID global para todo el sistema, si es un valor entero tiene que ser lo suficientemente grande como para no quedarnos sin ID disponibles. Un entero de 32 bits no suele ser suficiente para las magnitudes de datos que manejan las bases de datos hoy en día. Un entero de 64 bits resulta más adecuado. Esta opción generalmente se suele implementar con una tabla adicional que guarda el ultimo ID global. En este caso hay que tener mucho cuidado de no bloquear dicha tabla durante una transacción ya que en caso contrario no se podrán tener transacciones simultáneas en la base de datos.
Una alternativa posible es el uso de GUIDS (GlobalUnique Identifier) como ID. Los GUID son valores pseudo aleatorios generados por un algoritmo de tal manera que la posibilidad de obtener dos valores duplicados es muy baja. Los GUID son escritos empleando una palabra de cuatro bytes, tres palabras dedos bytes y una palabra de seis bytes, como por ejemplo:

```
{4F15E4EA-4FAA-00D3-000C-E00C0001}
```

Debido a su longitud se suelen almacenar directamente como una cadena de caracteres por lo cual es conveniente verificar cómo se comporta el motor de datos en términos de rendimiento al manejar claves alfanuméricas. Todas estas opciones no cubren el universo de alternativas posibles a la hora de considerar IDs. Es por ello que es conveniente que la parte del framework responsable de la generación de IDs este planteada de forma que el usuario del framework pueda definir su propia forma de generación de IDs.

6.3.5.3 Proxys

Los objetos no suelen vivir aislados sino que interactúan con otros objetos con los que están directamente relacionados. Esto se traduce que al recuperar una instancia debamos traer en cascada otros objetos. Si esto se realiza sin tomar ninguna precaución es posible terminar con toda la base de datos en memoria en el peor de los casos y con más objetos de los que realmente se

necesitan en promedio. Para evitarlo se suele postergar la recuperación del objeto hasta el instante en el que se necesita creando un proxy virtual para cada objeto de negocios de modo que toda interacción con el mismo sólo se realiza a través de su proxy. El patrón proxy es sumamente difícil de implementar. Para empezar los proxys muchas veces imponen restricciones sobre la forma en el que escribimos los objetos de negocio por ejemplo exigiendo que todos los métodos sean virtuales. Como escribir el proxy para cada objetos de negocio es una tarea tediosa y sumamente repetitiva se suelen optar por distintas alternativas que van desde la generación de código, generación automáticas de bytecodes de forma dinámica. Por otra parte, si los proxys se implementan sin miramientos realizando un proxy por cada objeto de negocio puede terminar con problemas graves de rendimiento, sobre todo para el caso de las colecciones.
Otro objetivo de los proxys puede ser interceptar cualquier mensaje que modifique al objeto a quien envuelve y notificar a la transacción que el objeto esta modificado para que la misma lo ponga en la colección papelera. Otra posibilidad es aprovechar los proxys para verificar que una propiedad marcada como no mutable efectivamente lo sea. Para ello el proxy intercepta la llamada al getter y luego de ejecutarla verifica si el objeto ha cambiado su representación lanzando una excepción en caso que así sucediera. No es muy complejo idear una batería de pruebas unitarias para que cada objeto del modelo de negocio verifique que se cumpla el contrato de mutabilidad declarado en los metadatos de cada clase.

6.3.5.4 Rendimiento

El rendimiento a la hora de diseñar nuestro framework de persistencia cobra especial importancia debido a que se espera que el mismo resuelva las consultas abstrayendo al usuario del mismo de las sentencias SQL subyacentes. Los beneficios de esto tienen como contrapartida que generalmente el usuario tendrá poco control de este proceso viéndose limitado en posibilidades a la hora de tunear las sentencias SQL. Si bien es aceptable que la rendimiento de los SQLs generados por el framework no sea óptima, si debe tener ciertas consideraciones y seguir ciertas reglas.
Para empezar debe tratar de minimizar el número de llamadas interproceso realizadas contra el motor de la base de datos. Esto se logra aprovechando cada una para traer la mayor cantidad de datos útiles potencialmente utilizables siempre dentro de cierto límite. Esto se puede ver claramente planteando un escenario. Supongamos que tenemos un objeto factura que a su vez posee una colección de 20 ítems que corresponde al detalle de la misma. Si generamos un proxy para la factura y un proxy para cada ítem terminaremos realizando 21 consultas sobre la base de datos lo cual evidentemente no es optimo. Intuitivamente si quisiéramos resolver esa consulta seguramente resulte en un solo select que contiene un join entre la tablas que tienen la información sobre la factura y los ítems lo cual resulta enormemente más óptimo. Como estrategia intermedia podemos tener un proxy para la factura y un proxy para la colección de ítems de modo que el problema se resolvería en dos consultas, uno para la factura y otro para los ítems. Esta estrategia si bien es menos eficiente que la anterior suele ser más útil en los casos donde se tiene una colección numerosa de facturas y solo se accederá a los ítems en un caso particular.
Como regla general se puede decir que para el caso de los mapeos uno a muchos conviene traer toda la información haciendo join entre las tablas involucradas siendo deseable la posibilidad de configurar que trabaje como en el último caso especificando proxys para las colecciones.
Para las relaciones uno a uno donde hay más de una tabla involucrada la cosa es más abierta dependiendo del negocio si conviene o no hacer join entre estas tablas.
Para las jerarquías es necesario hacer join entre las distintas tablas comprendidas en la jerarquía pudiendo llegar a requerir hacer left joins o union para el caso de los mapeos condicionales.
Salvo para el caso de la jerarquía siempre conviene imponer un límite en la cantidad de tablas sobre las que se hacen los join estando los motores de bases de datos generalmente optimizados para 3 o 4 tablas. Dar la posibilidad al usuario del framework de tener cierto control de cómo se realizan las búsquedas es una característica muy anhelada.

6.3.5.5 Consultas

El framework de persistencia debe proveer un mecanismo de consultas para recuperar colecciones de objetos bajo algún criterio. Esto será responsabilidad del motor de consultas del framework cuya complejidad variará según la potencia y opciones de consultas que se deseen ofrecer.
Al momento de diseñar las consultas es importante analizar si existe algún estándar de consultas para trabajar con objetos en el entorno sobre el cual se está desarrollando. En caso afirmativo tal vez resulte una buena opción adherirse al mismo. En caso que optemos por una solución propietaria podemos seguir tres caminos: definir nuestro propio lenguaje de consultas, definir un objeto predefinido para hacer consultas, construir las consultas como un compositor de objetos provistos por el framework.
Para definir nuestro propio lenguaje de consultas es importante que sea fácil de comprender por parte de quien lo va a usar y por ello se recomienda elegir una sintaxis similar al SQL. El desafió es asegurar que dicho lenguaje sea igual de potente, es decir que al ejecutar varias veces la misma consulta devuelva siempre lo mismo. Para ello se asume que solo se pueden realizar consultas que permitan especificar propiedades no mutables como parte del criterio de búsqueda. En el caso que optemos por ofrecer un objeto para hacer consultas este se tratará de un objeto al cual se le definen ciertas propiedades como el nombre de la clase sobre el cual realizaremos las búsquedas y algún criterio. Realizaremos las consultas de la siguiente forma:

```
Selection s = new selection(typeOf(Animal));
s.criteria = new FieldEquals(Nombre,'Perro');
```

6.3.5.6 Colecciones

Será tarea intrínseca del framework trabajar con colecciones de objetos. Lo que en el mundo relacional se lee como relaciones uno a muchos y claves foráneas entre tablas, dentro del desarrollo tendremos objetos en memoria que a su vez podrán referenciar colecciones de otros objetos en memoria. El problema de las colecciones ocurre cuando la cantidad de elementos de la misma es muy grande y ocupan más memoria de la que podríamos desear atentando contra el desempeño de la aplicación. Para esos casos lo mejor es construir un mecanismo de paginación.
El mecanismo de paginación nos permite evitar tener la colección completa en memoria y solo disponer de un conjunto de la misma. No obstante para el que utiliza la colección esto será transparente y parecerá como si estuvieran todos los datos disponibles.

6.4 Análisis y Diseño del framework

6.4.1 Conexión y relación de la Base de Datos

Esta es una parte muy importante del proyecto, ya que es sobre la base de datos donde recaerá toda la funcionalidad final. Es importante definir que en este framework se van a relacionar las tablas con los objetos con los que trabajemos.
Independientemente del nombre del objeto que creemos, este deberá guardar correspondencia con un fichero de properties, que será el encargado de definir el mapeo de las tablas de Base de Datos con los atributos de las entidades que vayamos a utilizar. Además, este properties se utilizará para poder obtener la información necesaria para la construcción de las sentencias SQL y las cadenas de conexión. Cada tabla tendrá su properties en concreto y cada una de las entradas de dicho properties, tendrá la siguiente estructura:

nombre_tabla.nombre_campo = etiqueta_campo

Puesto que estamos hablando en este apartado de la Base de Datos con la que se interrelacionará el framework de persistencia, es necesario que hagamos mención de la conexión que utilizamos y como configurarla para diferentes sistemas. El sistema elegido es, al igual que lo comentado anteriormente, gestionar la configuración de la conexión mediante fichero de properties. En concreto el fichero de properties debe estar compuesto por las siguientes variables:

- nombreDriver={nombre del driver que nos permitirá establecer conexión con la Base de Datos}
- dbURL={cadena de conexión con la Base de Datos}
- nombreUsuario={nombre de usuario con el que acceder a la Base de Datos}
- password={contraseña del usuario introducido anteriormente}

Datos conexión MySQL (datos sacados de un fichero de pruebas).

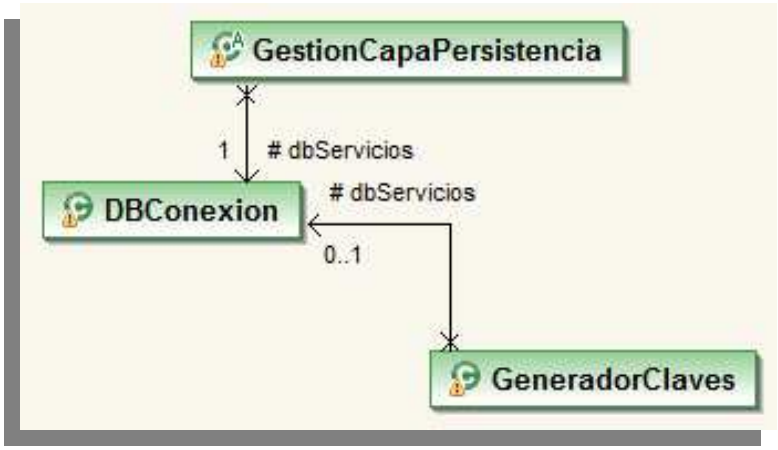
Resultado_01_MYSQL_buscarContrato.log

```
1 controlador: com.mysql.jdbc.Driver
2 dbURL: jdbc:mysql://free-mysql.BizHostNet.com:3306/1337755750
3 usuario: 1337755750
4 password: 040404
5 ruta: C:\UOC\PFC\Source\PFC_BCNDANIEL\prueba\MYSQL.sql
```

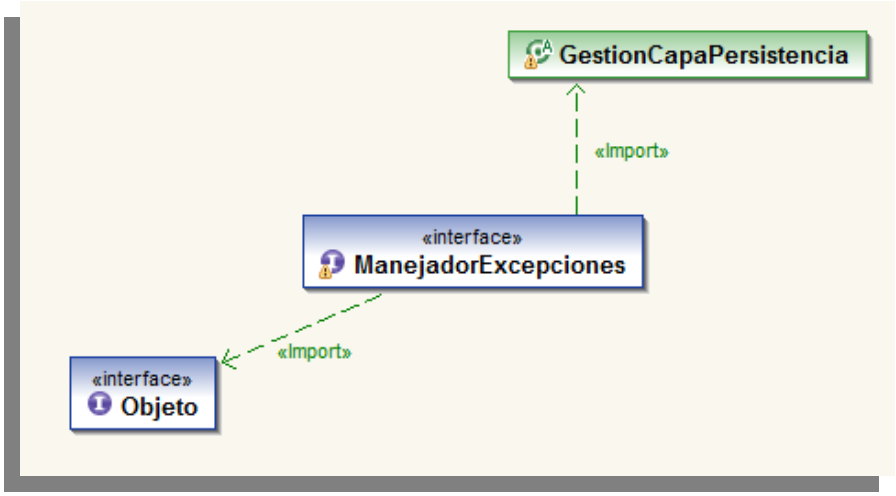
Datos conexión ORACLE (datos sacados de un fichero de pruebas).

Resultado_01_ORACLE_buscarContrato.log

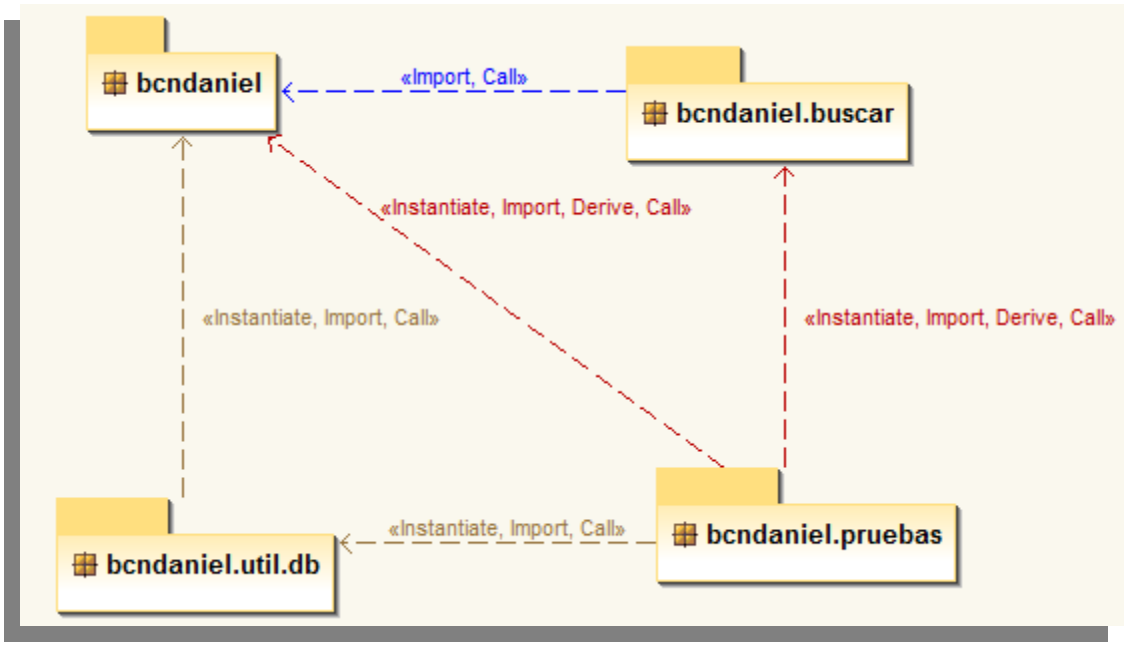
```
1 controlador: oracle.jdbc.driver.OracleDriver
2 dbURL: jdbc:oracle:thin:@localhost:1521:XE
3 usuario: PFC
4 password: 0404
5 ruta: C:\UOC\PFC\Source\PFC_BCNDANIEL\prueba\ORACLE.sql
```



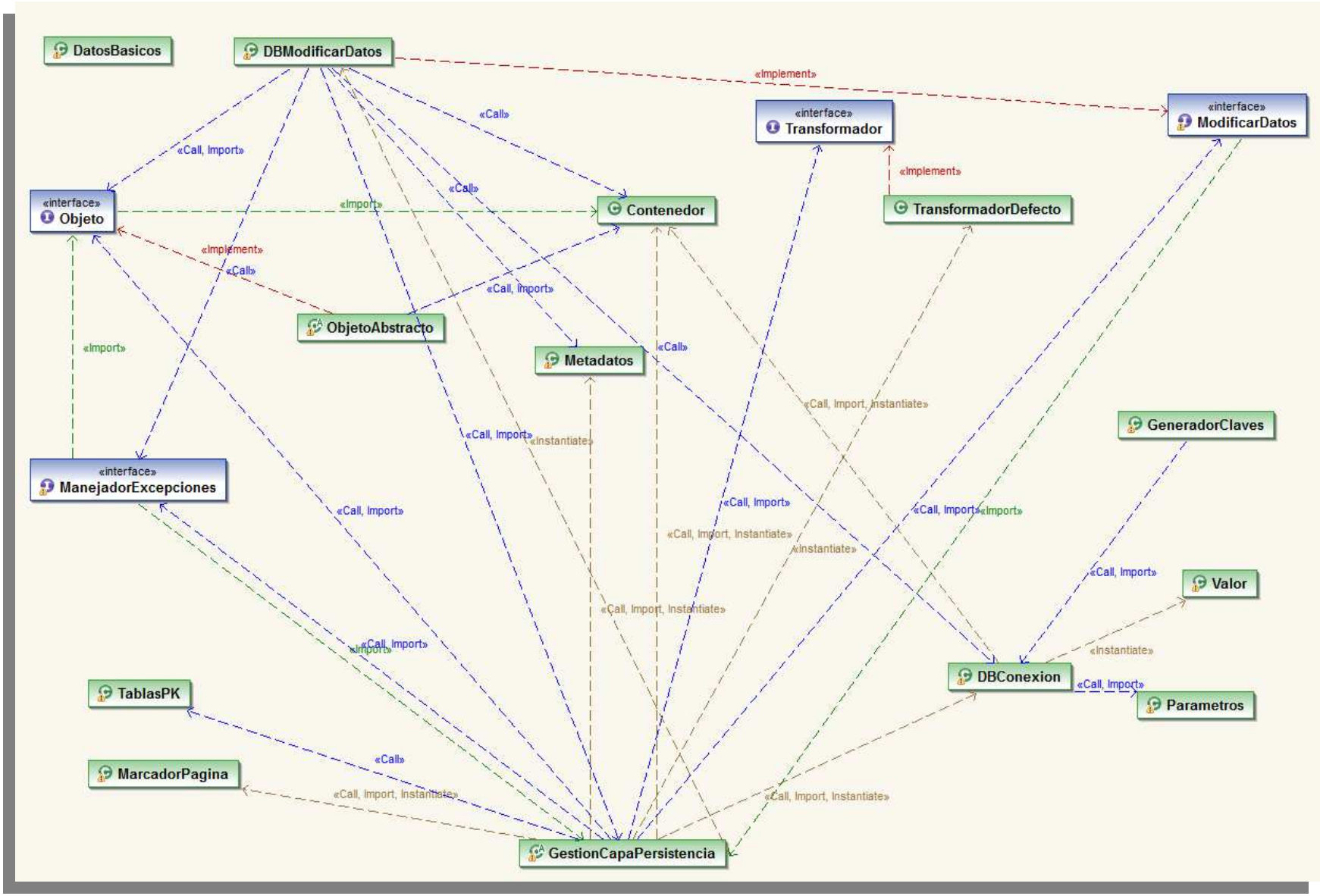
6.4.2 Sistema de excepciones



6.4.3 Diagrama de paquetes

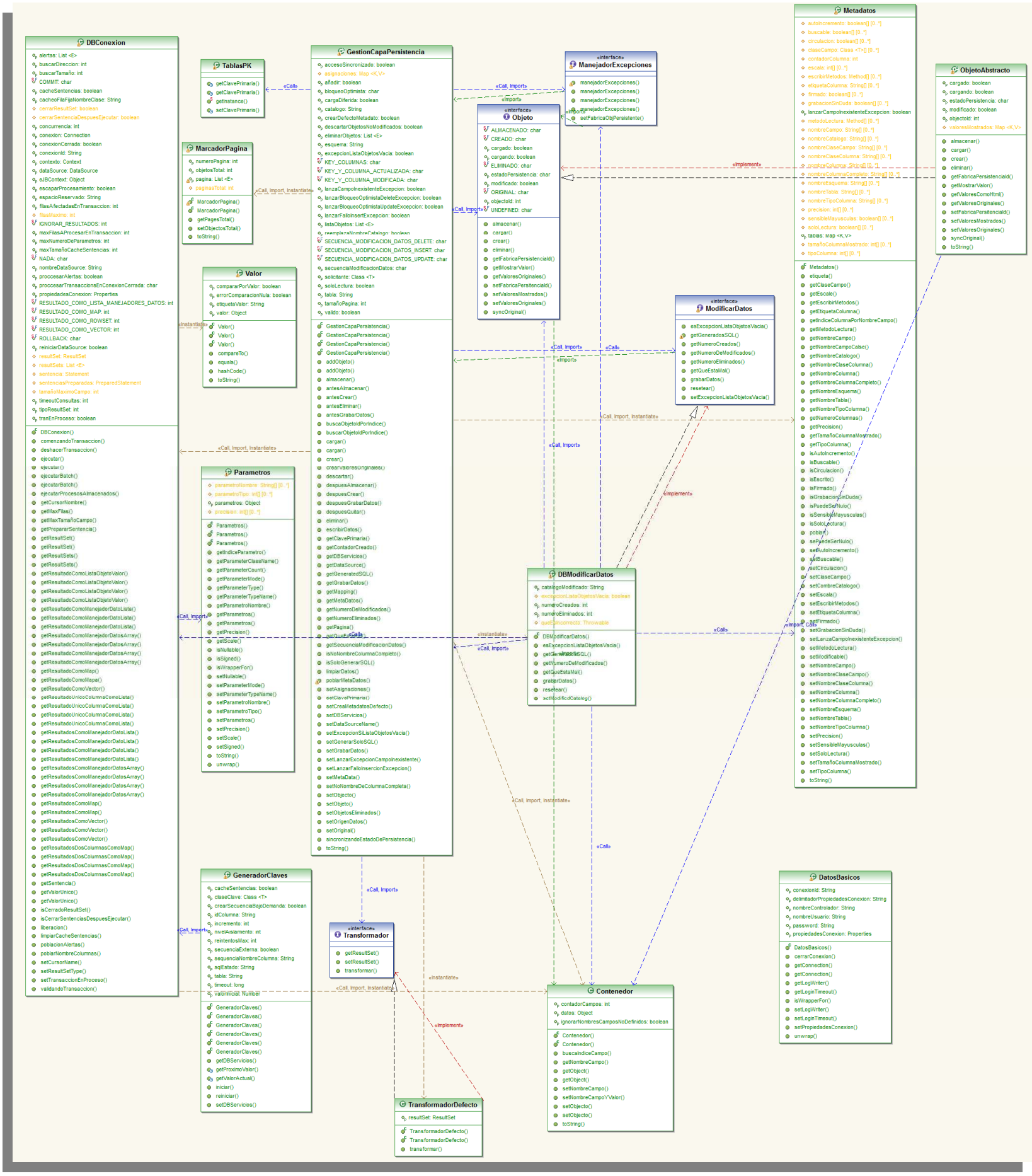


6.4.4 Diagrama de dependencias



6.4.5 Diagrama de clases

Este es el diagrama global de clases del framework de persistencia.



7. Detalles de la aplicación

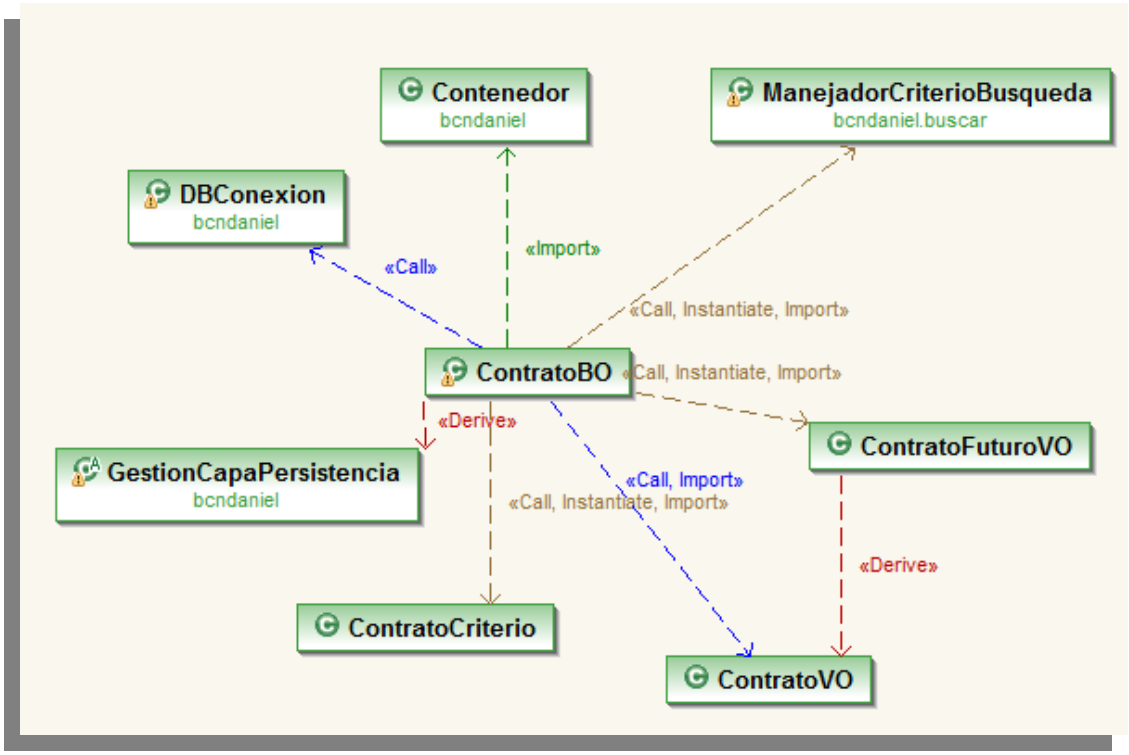
7.1 Funcionamiento del framework

Como método para el uso del software desarrollado en el PFC tenemos que tener en cuenta las siguientes operaciones:

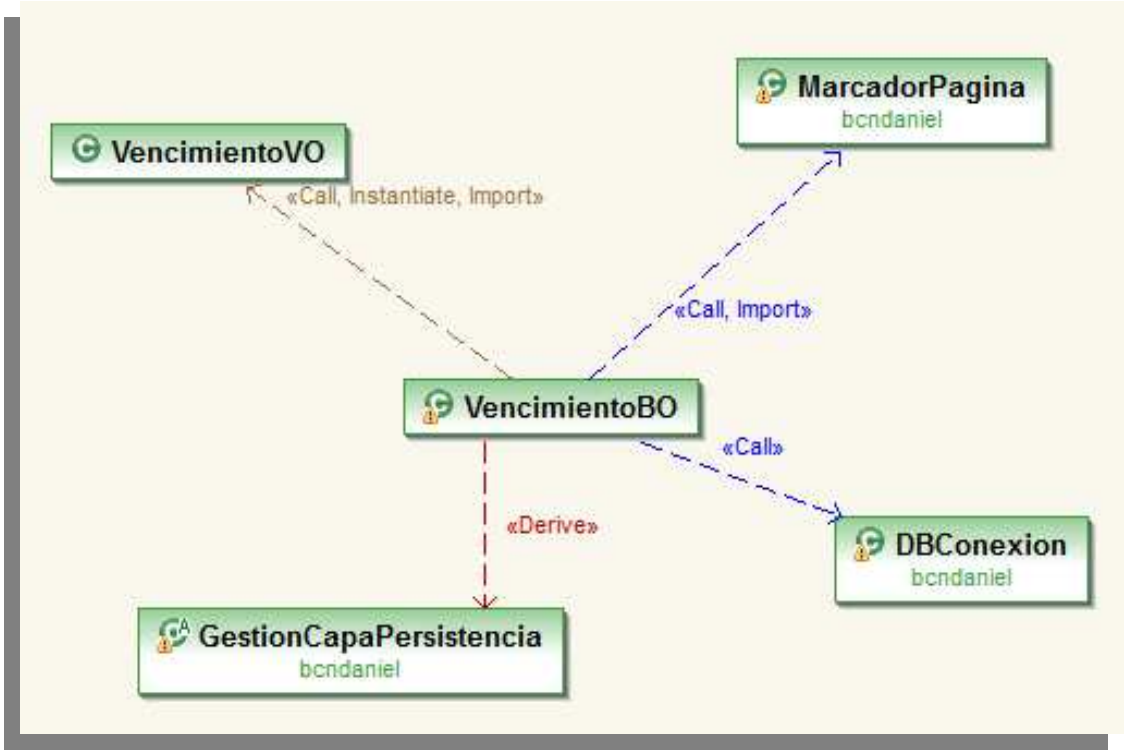
- La tabla que queramos leer debe ser referenciada por un objeto que se debe extender de la clase bcndaniel. ObjetoAbstracto.
- Para su uso crearemos un objeto que extienda de la clase bcndaniel. GestionCapaPersistencia.
- Por último, crearemos un fichero con el nombreTabla.properties, donde le haremos la correlación entre el nombre de los campos de la tabla y el nombre de los campos del objeto.

En el paquete bcndaniel.pruebas podemos encontrar varios ejemplos que explican su uso. Uno de ellos es ContratoBO.java (El juego de pruebas se basa en datos referidos a la gestión de derivados).

7.1.1 Pruebas ContratoBO.java



7.1.2 Pruebas VencimientoVO.java



7.2 Conceptos básicos de utilización

Es posible transmitir solamente los objetos que sea imprescindibles, aquellos que contengan los datos de las actualizaciones, de este modo se disminuyen las peticiones a BDD y resulta mucho más óptimo.

Otra particularidad es que se pueden enviar al servidor de aplicaciones, además de los parámetros, sentencias adicionales DML:

7.2.1 Construccion de objeto

```
ContratoBO contratoBO = new ContratoBO(ContratoVO.class);
```

Fijamos el parámetro a TRUE

```
contratoBO.setSoloSQLGenerado(true);
```

7.2.2 Realizamos la operación

```
contratoBO.escribirDatos();
```

7.2.3 Ver el resultado final en lista

```
List todosCambios = contratoBO.getSQLGenerado();
```

7.2.4 Realizar modificación

```
DBConexion dbConexion = new DBConexion();
```

Se realiza el cambio en BBDD

```
dbConexion.ejecutar(sql, arg);
```

7.2.5 Obtener resultados

Mantenemos las estructuras lo suficientemente abiertas para que el desarrollador emplee aquella manera que considere más adecuada para obtener los resultados en el ResultSet: de otro Frameworks concurrente, de sentencias SQL directamente escritas por usuario, etc.

```
ContratoBO contratoBO = new ContratoBO(ContratoVO.class);
List arg = new ArrayList();
arg.add(new BigDecimal(100));
arg.add("F");
String sql = "SELECT * FROM CONTRATO WHERE MULTIPLICADOR  >= ? AND TIPO = ?";
DBconexion dbConexion = new DBConexion();
/************************/
contratoBO.carga(dbConexion.getResultSet(sql, arg));
dbConexion.liberacion(true);
List contratos = contratoBO.getObjetoLista();
```

Generalmente se pueden almacenar las sentencias SQL en archivos de tipo .properties o bien en fichero .xml y añadir clausulas tipo WHERE basadas en las selecciones del usuario. Hay varios objetos en el framework para realizar esta actividad y facilitarla la labor.

- Ejecución en modo desconectado: se pueden rellanar los objetos de la BBDD o cualquier otra fuente, modificarlos, crear otros nuevos y luego serializarlos. Después se pueden restaurar, conectar la BBDD o enviar los objetos a un servidor de aplicaciones y aplicar los cambios.
- Mantiene la capacidad de utilizar POJO para la persistencia (Plain Java Objects): solo es preciso extender (por herencia) desde la clase bcndaniel. ObjetoAbstracto.
- Se pueden generar actualizaciones (UPDATE) solamente para las columnas que han sido modificadas.
- Bloqueo optimista de concurrencia: la clausula WHERE para realizar actualización (UPDATE) y borrado (DELETE) se genera usando las propiedades:
  1. KEY_COLUMNAS
  2. KEY_Y_COLUMNAS_MODIFICADAS
  3. KEY_Y_COLUMNAS_ACTUALIZABLES

- Secuencia para modificación de datos: cuando se modifican muchos objetos al tiempo, se puede especificar el orden de la modificación, por ejemplo se puede especificar que se realice primero un borrado (DELETE), luego una actualización (UPDATE), luego añadir datos (INSERT) (cualquier orden):

```
ContratoBO contratoBO = new ContratoBO(ContratoVO.class);
/************************/
char[] orden = new char[3];
orden [0] = ContratoBO.SECUENCIA_MODIFICACION_DATOS_DELETE;
orden [1] = ContratoBO.SECUENCIA_MODIFICACION_DATOS_UPDATE;
orden [2] = ContratoBO.SECUENCIA_MODIFICACION_DATOS_INSERT;
contratoBO.setSecuenciaDeModificacionDeDatos(orden);
```

• Se pueden añadir los resultados de una sentencia SELECT a la lista de objetos en lugar de reemplazar el resultado previo, mediante el uso del método correspondiente, y se realiza mediante el uso de UNION ALL

• Soporte de paginación: se puede enviar al cliente o visualizar únicamente la cantidad específica de registros:

```
ContratoBO contratoBO = new ContratoBO (ContratoVO.class);
contratoBO.carga(*****)
MarcadorPagina mp1 = this.getPagina(1);
Iterator iter = mp1.getPagina().iterator();
while (iter1.hasNext()) {
ContratoVO contratoVO = (ContratoVO) iter.next();
System.out.println(contratoVO);
}
```

• La arquitectura está basada en la aplicación del patrón DAO, así que si algún día se precisa cambiar a otro Frameworks el uso de nuestro framework no debe ser un inconveniente.

8. Funcionamiento del framework

8.1 Presentación

Para la verificación del correcto funcionamiento del Frameworks de persistencia se presenta una sencilla colección de ejemplos de uso de los mismos. Se ha tratado que la aplicación fuera lo más sencilla posible y al tiempo lo más completa para verificar.

8.2 Características

La base de datos utilizadas han sido Oracle10 (una de las bases de datos más potente y utilizadas en el mercado) y MySQL.

La BDD Oracle la he creado localmente y no es posible hacer pruebas online. Para MySQL he encontrado una web que suministraba acceso a su BDD, dándonos un usuario y una password para poder trabajar con ella (el único inconveniente es que es un poco lenta). Adjunto datos de conexión:

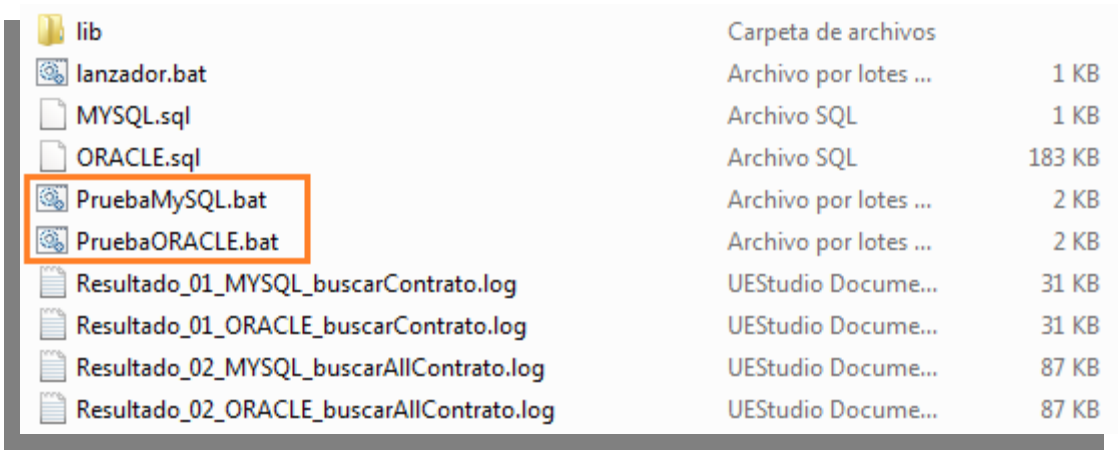
controlador: com.mysql.jdbc.Driver

dbURL: jdbc:mysql://**free-mysql.BizHostNet.com:3306**/1337755750

usuario: **1337755750**

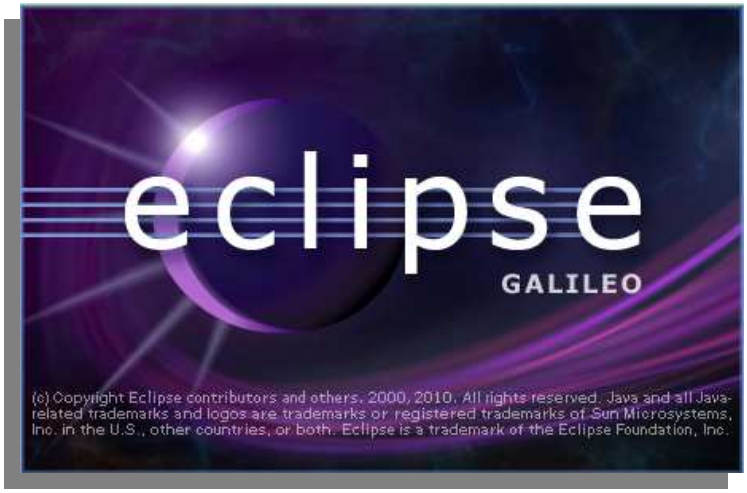
password: **040404**

En el caso de este ejemplo, ya viene preconfigurada para su uso, y solo es preciso utilizar los lanzadores que acompañan la distribución.



El entorno de desarrollo ha sido utilizando los componentes:

• Eclipse Galileo v3.5: utilizada para el desarrollo completo de todas las partes de la codificación y prueba.



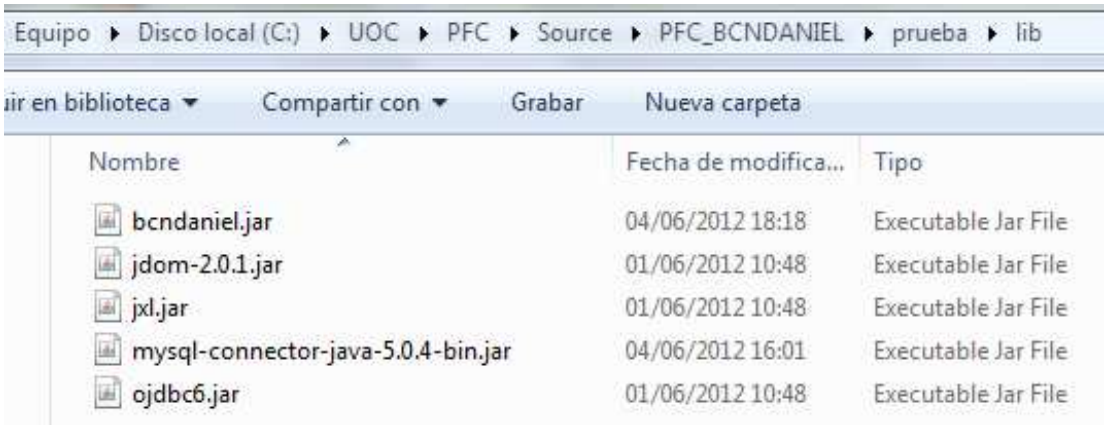
En cuanto a la instalación de las pruebas, solo se requiere crear el workspace en la siguiente ruta:

Siendo la PFC_BCNDANIEL la carpeta del proyecto.

Se ha utilizado java 1.6.0_01.

```
C:\Users\0>java -version
java version "1.6.0_01"
Java(TM) SE Runtime Environment (build 1.6.0_01-b06)
Java HotSpot(TM) Client VM (build 1.6.0_01-b06, mixed mode, sharing)
```

Todos los jars necesarios están en la siguiente ruta:



Bcndaniel.jar contiene todo nuestro proyecto y se reconstruirá cada vez que lancemos uno de los .bat que ejecutan las pruebas.

8.3 Ejemplos

El paquete de ejemplos está diseñado para cargar un sencillo conjunto de datos (creando la BBDD en cada ejecución de los ejemplos) de forma que cada lanzamiento pueda verificar un uso de los métodos y objetos de la capa de persistencia.

Tal y como se puede verificar, no se necesitan fichero complejos de instalación, estilo .xml, para establecer la relación entre el Framework y la BDD. Únicamente se utilizan ficheros .properties, con las características iniciales de la BBDD y comando de uso.

9. Bibliografía

HIBERNATE - Persistencia relacional para Java idiomático
http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/preface.html
Autor: Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, y Steve Ebersole

Patrones Enterprise (parte 1)
http://elblogdelfrasco.blogspot.com.es/2008/12/patrones-enterprise-parte-1.html
Autor: Adrián Paredes

Patrones Enterprise (parte 2)
http://elblogdelfrasco.blogspot.com.es/2008/12/patrones-enterprise-parte-2.html
Autor: Adrián Paredes

http://es.wikipedia.org/wiki/Hibernate

Descripción de iBatis: ¿Qué es?, ¿para qué sirve?
http://mikiorbe.wordpress.com/2009/06/24/descripcion-de-ibatis-%C2%BFque-es-%C2%BFpara-que-sirve/
Autor: Miguel Orbegozo

Thinking in Patterns
http://www.tutok.sk/fastgl/download/books/Thinking%20in%20Patterns%20with%20Java.pdf
Autor: Bruce Eckel

Choose your programming language to compare web frameworks
http://www.bestwebframeworks.com

A Practical Introduction to Enterprise Java Beans
http://www.comptechdoc.org/docs/kanti/ejb/bmpentity.html
Autor: Kantimahanti Prasad

Java Persistence API (JPA)
http://www.coplec.org/?q=book/export/html/240

¿Qué es un framework web?
http://www.cssblog.es/guias/Framework.pdf
Autor: Javier J. Gutiérrez.

ANÁLISIS Y USO DE FRAMEWORKS DE PERSISTENCIA EN JAVA
http://www.iit.upcomillas.es/pfc/resumenes/450955e7368ca.pdf
Autor: Alfredo Payá Martín

Persistencia de Objetos Java Utilizando JDO
http://www.programacion.com/articulo/persistencia_de_objetos_java_utilizando_jdo_310
Autor: Fernando Santos

Building Java Programs: A Back to Basics Approach
Autor: Marty Stepp