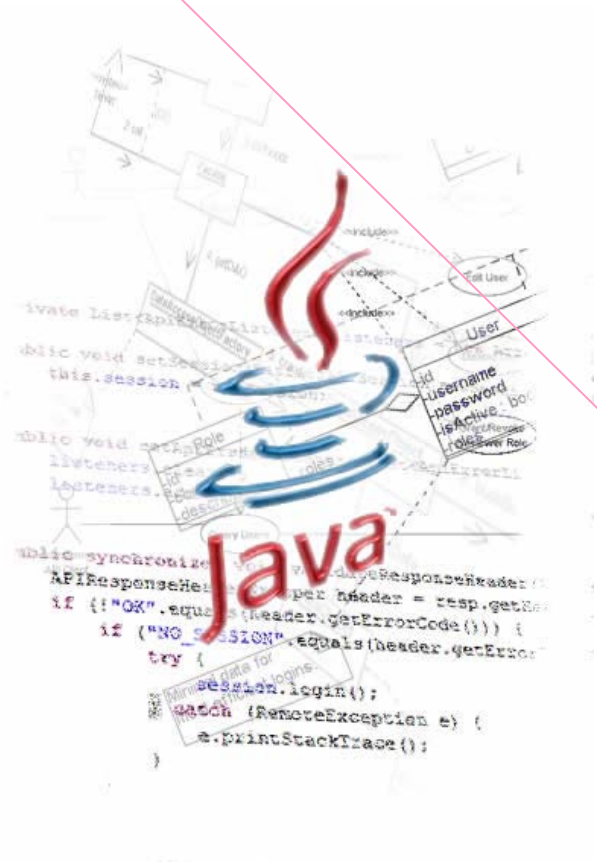
Licencia



Diseño e implementación de un Frameworks de persistencia

Proyecto fin de carrera

Ingeniería Informática
Curso 2010-2011
Consultor: Oscar Escudero Sánchez

Sergio Lovillo Galán
20 de Junio de 2011

A mis padres por hacerme ser lo que soy en la vida, inculcándome el trabajo como forma de vivir. A mi mujer por hacerme tan feliz en la vida sin pensar tanto en el trabajo.

Índice de contenidos

Índice de tablas

Índice de figuras

1. Resumen ejecutivo

Los Frameworks constituyen el nuevo paradigma en cuanto al desarrollo de software se refiere. Entre sus principales características se encuentran la facilidad para la reutilización de código.

En este marco específico proporcionados por la tecnología usaremos la tecnología JAVA y su extensión en cuanto a la persistencia de datos.

El Frameworks de persistencia es el responsable de gestionar la lógica de acceso a los datos en un SGBD (Sistema de Gestión de Bases de Datos), ya sea de entrada o salida, y ocultando los detalles más pesados relativos a la estructura propia de la Base de Datos utilizada, de manera completa y transparentemente.

En conclusión, este proyecto se basa en un análisis del los Frameworks existentes, analizando sus características y profundizando en los detalles concretos de su actividad y manejo en cuanto a la persistencia.

2. Objetivos generales y específicos

Éste estudio nos permitirá alcanzar los siguientes objetivos:

- Definición.
- Características generales.
- Tipos.
- Persistencia de datos.
- Frameworks específicos para la persistencia de datos.
- Open source y propietarios.
- Principales Frameworks y análisis comparativo de rendimiento e implantación de los mismos.
- Diseño de Frameworks de persistencia
    - Análisis de requisitos de un Frameworks de persistencia.
    - Particularidades de diseño:
        - Análisis del dominio
        - Diseño del Frameworks
        - “instanciación” del Frameworks
        - Implementación del Frameworks de persistencia usando tecnología java.
        - Desarrollo de una aplicación de ejemplo.

3. Planificación con los hitos principales

1. Aspectos iniciales correspondientes a la toma de datos y análisis.
    - Comprensión de Frameworks:
        - Búsqueda de información.
        - Análisis de los modelos.
        - Modelos open source y propietarios.

- Análisis de las herramientas generales:
  - Elementos generales del uso.
  - Estructura.
  - Diseño.
- Análisis de la persistencia de datos.
  - Características.
  - Tipos de sistemas.
  - Tecnología.
- Modelos orientados a la gestión de la persistencia:
  - Modelos open source.
  - Modelos propietarios.
  - Comparativa.

2. Aspectos del diseño de la aplicación de persistencia.
   - Recogida y documentación de los requisitos:
     - Diagramas de casos de uso.
     - Documentación de los casos de uso.
   - Análisis:
     - Revisión de los casos de uso de la etapa previa.
     - Representación de las clases. Representación mediante diagramas de colaboración para cada uno de los casos de uso.
     - Identificación de las relaciones entre las clases de entidad y elaboración de un diagrama estático UML.
     - Especificación formal de los casos de uso.
     - Especificación de los diversos diagramas UML necesarios.
   - Diseño:
     - Estudio de patrones adicionales.
     - Análisis de los subsistemas de la aplicación.
     - Preparación de las pruebas para testear cada módulo.
     - Interacción con los Sistemas de Gestión de Bases de Datos.
   - Implementación o codificación para cada uno de los módulos:
     - Implementación de las clases utilizando el lenguaje java.
   - Prueba de la aplicación.
     - Test de aplicación.
     - Elaboración de una aplicación de ejemplo.
3. Instalación de software.
   - Entrega de la aplicación.
   - Integración en el IDE seleccionado.
4. Documentación final del proyecto
   - Manual de usuario
   - Revisión de la documentación interna del software.
   - Documentación en formato javadoc.
5. Elaboración de la memoria del PFC
6. Elaboración de la presentación del PFC

ID	TAREA	Duración	Comienzo	Fin	Predecesoras
1	Toma de datos y análisis	28	10-03-2011	13-04-2011	
1.1	Datos generales	5	10-03-2011	14-03-2011	
1.2	Análisis herramientas	10	15-03-2011	26-03-2011	1.1

1.3	Análisis de la persistencia de Datos	5	27-03-2011	02-04-2011	1.2
1.4	Persistencia	7	04-04-2011	13-04-2011	1.3
2	PEC2	24	14-03-2011	13-04-2011	1
3	Entregar PEC2	1	14-04-2011	14-04-2011	2
4	Diseño Framework	52	07-04-2011	14-06-2011	
4.1	Recogida y documentación de los requisitos	10	07-04-2011	17-04-2011	
4.2	Análisis	15	12-04-2011	30-04-2011	4.1
4.3	Diseño	20	21-04-2011	16-05-2011	4.2
4.4	Implementación o codificación	30	05-05-2011	13-06-2011	4.3
4.5	Pruebas	15	26-05-2011	14-06-2011	4.4
5	PEC3	26	14-04-2011	16-05-2011	2
6	Entregar PEC3	1	23-05-2011	23-05-2011	5
7	Elaboración de la documentación	35	28-04-2011	14-06-2011	4
8	Elaboración memoria PFC	66	18-03-2011	14-06-2011	
9	Elaboración presentación PFC	4	14-06-2011	20-06-2011	
10	Entrega final	1	20-06-2011	20-06-2011	

Tabla 1: Planificación de tareas y plazos

4. Características de los Frameworks

4.1. Que son los Frameworks

En el área de la programación, un Frameworks es un conjunto de código genérico o funciones que han sido diseñadas para realizar tareas frecuentes y comunes en cualquier tipo de aplicación, como por ejemplo, creación de objetos, conexiones con fases de datos, etc. Este tipo de componentes nos permiten disponer a priori de un gran conjunto de utilidades con las que desentendernos de aquellas funciones que dichos paquetes realizan.

En otro orden de cosas, los Frameworks son normalmente construidos en lenguajes orientados a objetos. Dichos lenguajes proporcionan una base muy extendida que permite al reutilización de código y la modularización de los componentes. Además, cada Frameworks implementará patrones de diseño que aseguren su escalabilidad.

Durante el desarrollo de un software, un Frameworks 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.

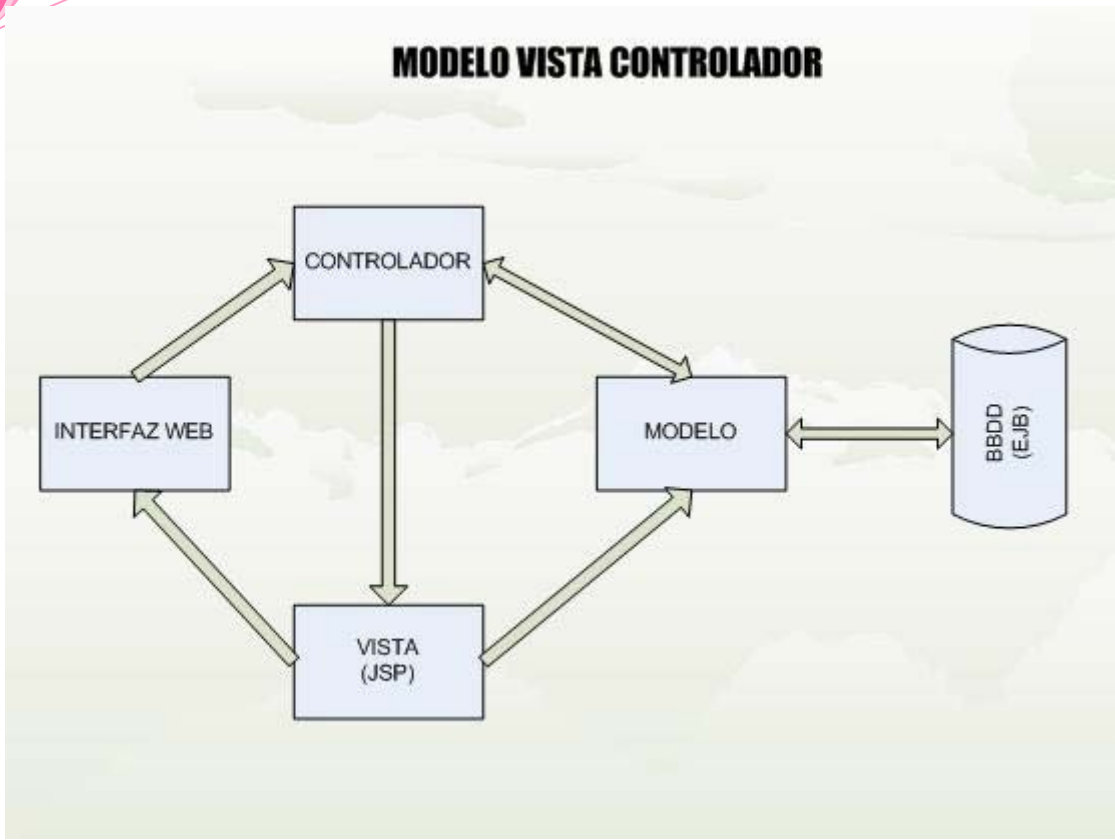


Figura 1: Modelo Vista Controlador

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:

- Ajustar el código a los estándares de desarrollo del uso de patrones
- Reducir los tiempos de desarrollo.
- Aumentar la fiabilidad del código.

4.2. Patrones de diseño: MVC

Es uno de los patrones más extendidos ya que el Modelo-Vista-Controlador es una metodología estructural que define el diseño de arquitecturas de aplicaciones para conseguir obtener una fuerte interactividad con los usuarios.

En dicho patrón podremos encontrar tres capas separadas:

- Capa Modelo: En esta capa se fijan los objetos que interactúan, entre otros, con la base de datos y también es aquí donde fijamos los procesos más pesados pertenecientes a la lógica del negocio.

- Capa Vista: Es la parte más representativa del patrón puesto que es la que interactúa con el usuario. El conjunto de páginas web que la contienen permiten realizar la salida y entrada de información con nuestros usuarios.
- Capa Controlador: Es la capa que se encarga de recibir la petición del usuario a través de la capa vista y derivarla a los procesos adecuados. Es en esta capa donde reside la mayor parte del flujo de ejecución del sistema.

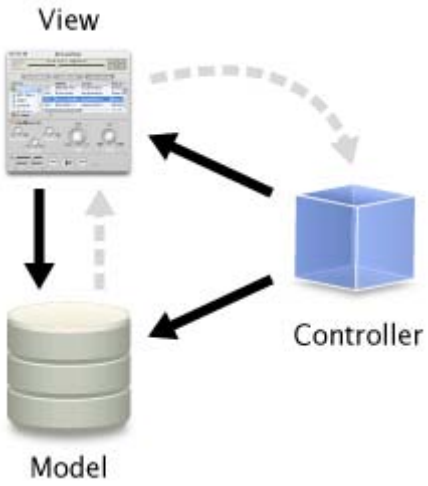


Figura 2: Modelo Vista Controlador

4.3. Empleo del patrón MVC-Framework

A modo de resumen podemos establecer que la interacción entro el patrón MVC y el Framework se realiza de la siguiente manera:

- El procedimiento es llamado desde el Framework
- Se han de implementar interfaces o extender las clases abstractas que proporciona el Frameworks por parte del desarrollador.

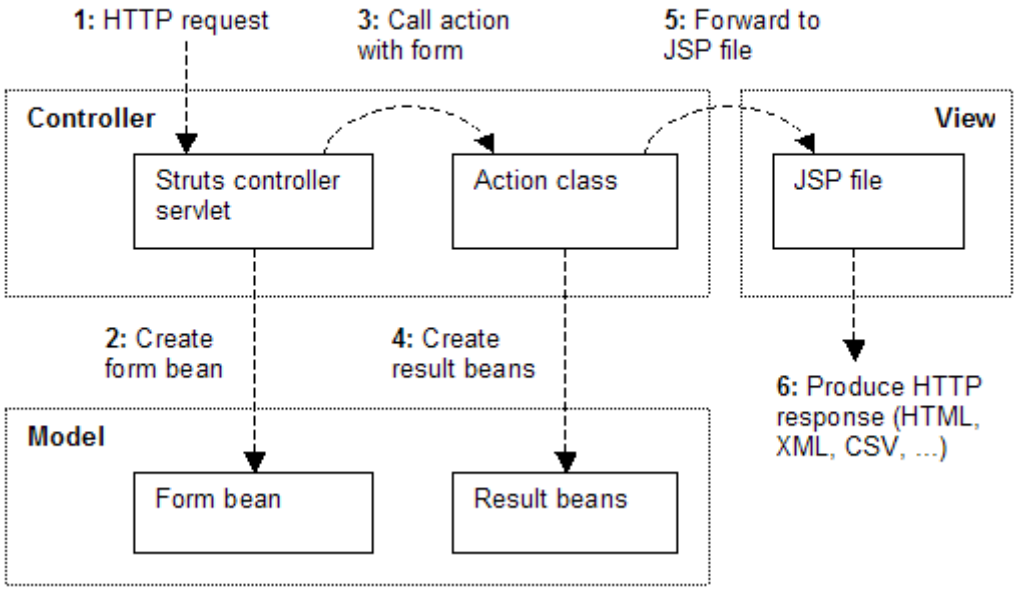


Figura 3: Ejemplo de diseño de aplicativo MVC en Struts

De esta manera, el desarrollador realiza una implementación determinada y específica del patrón MVC adaptándolo a las necesidades que se presenten en la arquitectura a la que se orienta el Framework, el domino que se desea implementar y las particularidades del mismo.

4.4. Arquitectura J2EE

Los Framework no están asociados a ninguna plataforma de desarrollo, pero por su escalabilidad y diversificación sus caminos se encuentran relativamente unidos.

En la actualidad existen otros tipos de Frameworks que no pertenecen a la tecnología Java de Oracle – Sun, como son Ruby on Rail, PHP, etc.

De cualquier forma, la plataforma J2EE reúne unas particularidades que la colocan en primera línea para su uso en cualquier tipo de desarrollo:

- Popularidad del lenguaje JAVA
- Tecnología robusta y fiable
- Preparada para el uso de la mayoría de tecnologías: JSP, funciones avanzadas de la lógica de negocio mediante JAVA, etc.
- Fácil integración con HTML, XML, etc.
- Gran cantidad de información en internet: Manuales, APIs, etc.
- IDE gratuitos de alta calidad de licencia abierta: Eclipse
- Gran cantidad de utilidades existentes portables y fiables.
- Fácil integración con cualquier sistema de base de datos: driver JDBC (Oracle)
- Framework MVC de Struts

J2EE es una gran herramienta enfocada al mundo empresarial con altísima producción y orientada a un desarrollo específico.

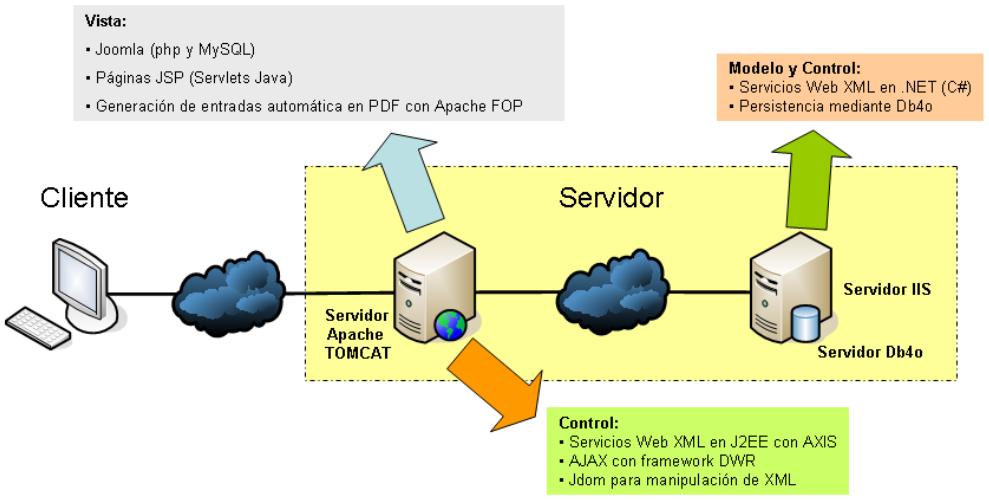


Figura 4: Ejemplo de arquitectura J2EE

En conclusión, J2EE establece una pauta de desarrollo para aplicaciones empresariales multicapa (elaborada por Oracle – Sun Microsystems). Fija de manera muy clara las aplicaciones empresariales basándolas en componentes modulares y estandarizados, proveyendo un completo conjunto de servicios a estos componentes, y manejándolos muchas de las funciones de la aplicación de forma automática, sin necesidad de alta complejidad de programación.

5. Sistemas de Gestión de Bases de Datos (SGBD)

5.1. Historia

A lo largo de la historia de los sistemas de gestión de las bases de datos se ha tratado de obtener el mayor rendimiento de los sistemas, limitados en cierta medida, por la capacidad tecnológica de cada época. Los tiempos de los grandes ordenadores mainframes, cuya información residía en tarjetas perforadas, y una gran manipulación por parte del operador de máquina, ha dejado paso a la actualidad, con sistemas de información distribuidos en el espacio de internet, con fuentes de datos de todo tipo y unas enormes capacidades de almacenamiento de información.

Figura 5: Dispositivos de almacenamiento

Cualquier desarrollo que requiera un procesamiento de datos, se puede dividir en dos niveles: datos persistentes y su procesamiento.

Es importante el uso de los mecanismos para interactuar con los datos, porque es sobre ellos donde radican importantes aspectos para el sistema, por su mantenimiento, diseño, escalabilidad y reusabilidad.

Las bases de datos surgen como una solución inicial a la persistencia de datos. Al principio, se accedía a los sistemas de gestión de bases de datos mediante tareas de muy bajo nivel. A medida que se fueron creando sistemas más complejos, se hizo necesario establecer mayores niveles de abstracción. Es de este modo como surgen los interfaces de acceso a las bases de datos y lenguajes de consulta. Aprovechando la aparición de los lenguajes de programación orientados a objetos, y la gran aceptación de las bases de datos relacionales, aparecen los denominados mapeadores como mecanismos de traducción entre los paradigmas subyacentes. A la par del avance tecnológico, los procesos de desarrollo de software evolucionaron integrando nuevas herramientas de abstracción como ser lenguajes de modelado. De igual manera, la experiencia de la comunidad de desarrolladores dio a conocer buenas soluciones para problemas conocidos, a las cuales se les denominaron patrones de diseño. Estos se beneficiaron de los lenguajes de modelado como herramienta para especificar soluciones. Tomando en cuenta que un sistema a menudo requiere la aplicación repetitiva de distintos patrones, se popularizan herramientas de generación automática de código. Aunque en la actualidad un conjunto de tecnologías para persistencia lideran el mercado, sí existen alternativas de menor difusión pero con enfoques diferentes: lenguajes persistentes, orientación a aspectos o prevalencia.

5.2. El progreso de los SGBD

Para almacenar grandes volúmenes de datos, controlar el acceso concurrente a los mismos y asegurar la consistencia y seguridad de la información surge el concepto de Database Management System (SGBD), SGBD – Sistemas de Gestión de Bases de Datos.

Puede definirse un SGBD: paquete o sistema de software diseñado para facilitar la creación y el mantenimiento de bases de datos computarizadas. Asimismo, se entiende Base de Datos como una colección de datos relacionados.

El primer SGBD de propósito general fue Integrated Data Store, diseñado por Charles Bachean en General Electric a principios de la década del 60. Fue la base para el Modelo de Datos de Red e influenció los sistemas de base de datos en esta época. A finales de los 60, IBM presenta el Information Management System (IMS), aun hoy en uso. El IMS constituyó la base para un método de representación de datos alternativo: Modelo de Datos Jerárquico.

Las bases de datos pre-relacionales no contaban con un conjunto de comandos que permitiese trabajar con los datos. Cada base tenía su propio lenguaje o utilizaba programas escritos en COBOL o C para manipular registros. Asimismo, eran virtualmente inflexibles y no permitían ningún cambio estructural sin tener que detener su ejecución y reescribir gran cantidad de código. Esto último funcionó de forma relativamente efectiva hasta fines de los 60 donde las aplicaciones se encontraban basadas estrictamente en procesamientos por lotes. A principio de los años 70, el crecimiento de aplicaciones que requerían interacción por parte de los usuarios demandó la necesidad de algo más flexible. Situaciones en que un campo extra era requerido o un número de subcampos excedían la máxima cantidad permitida en determinado archivo se volvieron más y más comunes. IBM, en 1970, bajo la dirección de Edgar Codd, y trabajando en el San José Research Laboratory, propuso un nuevo modelo de datos: Modelo de Datos Relacional.

Este nuevo modelo produjo el rápido desarrollo de varios sistemas de gestión de bases de datos basados en el mismo, así como importantes resultados teóricos que ubicaron el campo sobre una base firme. Los sistemas de base de datos maduraron como una disciplina académica, y la popularidad de los Sistemas de gestión de bases de datos relacionales se extendió hasta el plano comercial: el uso de Sistemas de gestión de bases de datos para administrar información corporativa se convirtió en una práctica estándar.

A partir de 1980, el Modelo de Datos Relacional se consolidó como el paradigma dominante, y los sistemas de base de datos continuaron ganando aceptación.
A finales de los 80 y principio de los 90, se lograron varios avances en el área de sistemas de base de datos. Se realizó una considerable investigación para obtener lenguajes de consulta más poderosos y modelos de datos más expresivos: gran énfasis en el análisis de datos complejos.

IBM, Oracle e Informix extendieron sus sistemas con la habilidad para almacenar nuevos tipos de datos (p.ej. imágenes) así como con la capacidad de realizar consultas más complejas. Sistemas especializados fueron desarrollados por varias empresas a fin de poder crear Data Warehouses, integrar información de distintas bases, y llevar a cabo análisis especializados de la información.

Concretamente en este tiempo emergen los denominados Sistemas de gestión de bases de datos Orientados a Objetos (SGBDOO) y los Sistemas de gestión de bases de datos Objeto-Relacional (SGBDOR).

La primera generación de SGDBOOs data del año 1986 y su objetivo era el de proveer persistencia para lenguajes orientados a objetos (relacionados con la inteligencia artificial). Se trataban de sistemas standalone, se basaban en lenguajes propietarios y no hacían uso de plataformas estándar a nivel de la industria. El lanzamiento de Ontos en 1989 por la compañía del mismo nombre marcó el comienzo de la segunda generación en el desarrollo de SGDBOOs. Esta generación se basó en utilizar una arquitectura Cliente/Servidor y plataformas conocidas como C++, X-Window y UNIX. Itasca, primer producto de la tercera generación, fue lanzado al mercado en agosto de 1990. Si bien los SGDBOOs de primera generación pueden ser considerados como extensiones a lenguajes orientados a objetos los sistemas de tercera generación pueden ser definidos como Sistemas de gestión de bases de datos con características avanzadas y lenguajes orientados a objetos para la definición y manipulación de datos.

Un aspecto interesante a tener en cuenta es que si bien la orientación a objetos se ha establecido firmemente en el mercado como uno de los paradigmas de desarrollo dominantes, su popularidad no tiene relación con la popularidad de los SGDBOOs. En concreto, la adopción de las bases de datos orientadas a objetos ha sido considerablemente limitada. Actualmente, los SGBD relacionales dominan el mercado: es poco probable que sean remplazados, en el corto plazo, por los SGDBOOs.

A mediados de los 90, con el auge de Internet y la popularidad de HTML como formato para desplegar información en este medio surge XML. Inicialmente desarrollado como estándar para la Web a fin de permitir que los datos estuviesen claramente separados de los detalles de presentación, rápidamente se notó que XML cubría la necesidad de una sintaxis flexible para intercambiar información entre aplicaciones.

XML no fue inicialmente desarrollado como un lenguaje para persistir información, pero al ser adoptado por numerosas organizaciones para llevar a cabo sus procesos de negocios, devino la necesidad de contar con un mecanismo para almacenar los datos en dicho formato.

Alternativas como persistencia de los documentos en el sistema de archivos o en una base relacional presentaron numerosas desventajas. Así surgen bases de datos capaces de entender la estructura de documentos XML, llevando a cabo consultas sobre los mismos de una manera eficiente: Base de Datos XML.

Actualmente se acepta la clasificación de las anteriores en dos categorías:

- Base de Datos XML-Native
- Base de Datos XML-Enabled

XML-Enabled presentan un modelo de datos interno distinto a XML (Modelo Relacional) y un componente de software para la traducción correspondiente.

De forma general, este módulo de traducción entre modelos no puede manejar todos los posibles documentos XML. En términos de lo anterior, reserva su funcionalidad para la subclase de documentos que cumplen con determinado esquema derivado a partir de los datos almacenados en la base.

Finalmente, a diferencia de las XML-Enabled, las bases de datos XML-Native son aquellas que utilizan el modelo de datos XML directamente.

Esto es, utilizan un conjunto de estructuras para almacenar documentos XML, en principio, arbitrarios.

6. La persistencia

6.1. Definición

Una posible definición de persistencia podría ser el hecho de guardar permanentemente la información generada por una aplicación en un sistema de almacenamiento, con el objetivo de poder ser usada con posterioridad.

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 explícita. 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 de encontrar 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.

6.2. Objetivos

¿Qué características debe tener una capa de persistencia para poder llegar a ser un sistema consolidado?

Si tomamos como base a Scott W. Ambler 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.

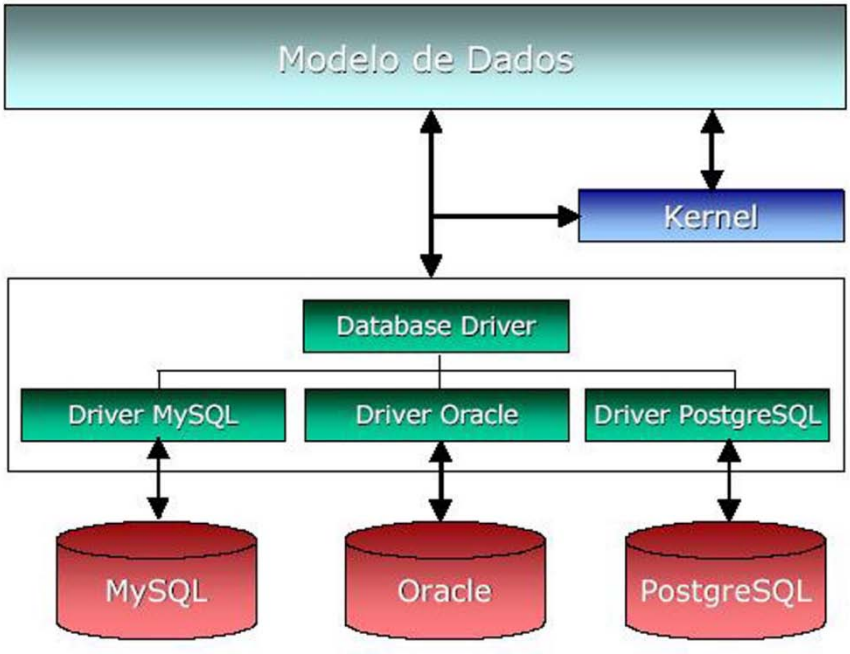


Figura 6: Ejemplo interacción con diversas bases de datos

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.

6.3. Modelos

La necesidad de la persistencia de los objetos ha conseguido que a lo largo del tiempo se hayan desarrollado diferentes sistemas a fin de conseguir este objetivo. Los sistemas más primitivos constituyeron la primera piedra en un proceso que ha permitido dar paso a sistemas más evolucionados, cómo son las capas de persistencia: el último peldaño evolutivo dentro de la escala de la persistencia de objetos. Casi todos de los diferentes sistemas de persistencia todavía se utilizan en mayor o menor grado.

Los principales sistemas de almacenamiento de datos con los que se trabaja habitualmente son las bases de datos. Entre las bases de datos se distinguen dos fundamentales: relacionales y orientadas a objetos. Además está la forma de bases de datos objeto-relacionales, mezcla de los dos sistemas de bases de datos. Otros sistemas con los que nos encontramos, que permiten realizar almacenado de datos, son los sistemas de ficheros, cómo pueden ser ficheros ASCII o los documentos XML.

Otros sistemas clave que también se deben mencionar por su creciente importancia son los que trabajan con arquitecturas de objetos distribuidos como EJB, y las bases de datos XML.

Características	iBatis	Hibernate	JPA
Simplicidad	La Mejor	Buena	Buena
Solución ORM completa	Media	La Mejor	La Mejor
Dependencia de SQL	Buena	Media	Media
Soporte de lenguaje Query	Buena	Media	Media
Rendimiento	La Mejor	La Mejor	(*)
Portabilidad entre diferentes bases de datos relacionales	Media	La Mejor	(*)
Soporte de comunidad y documentación	Media	Buena	Media
Portabilidad entre aplicaciones no Java	La Mejor	Buena	No soporta

Tabla 2: Comparativa de iBATIS, Hibernate y JPA

Dentro de la evolución de los sistemas de persistencia de objetos se ha trabajado sobre todos los diferentes sistemas de almacenamiento. Los primeros principalmente sobre ficheros, y base de datos relacionales. Los más evolucionados, que son las capas de persistencia, además permiten trabajar sobre sistemas distribuidos, y se empieza a hablar seriamente de trabajar sobre bases de datos XML.

No obstante, es preciso destacar que dentro de la diversidad de sistemas de persistencia, cómo podremos ver en la siguiente tabla comparativa, sólo las capas de

persistencia están orientadas, tal y como indica Ambler, hacia múltiples sistemas de almacenado, ficheros, diferentes bases de datos, etc.

La siguiente tabla comparativa enumera algunos de los diferentes sistemas y capas de persistencia.

Requerimiento	Serialización	JDBC-ODBC	ODBMS	EJB	JDO	ODMG	ADO
Diferente tipos de mecanismos	Sistemas de archivos	SGBDR	SGBDOO	SGBDR, EAI, sistemas de archivos, etc.	Sistema de archivos SGBDR, SGBDOO, EAI, BBDD cobol, otros	Sistema de archivos SGBDR, SGBDOO, EAI, BBDD cobol, otros	SGBDR
Encapsulación completa del mecanismos de persistencia	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Acciones multiobjeto	No	Sí	Sí	Sí	Sí	Sí	Sí
Transacciones	No	Sí	Sí	Sí	Sí	Sí	Sí
Extensibilidad	No	No	No	No	Sí	Sí	Solo para tipo de objeto
Identificadores de los objetos	Sí, manual	Sí, manual	Sí	Sí	Sí	Sí	Sí
Cursores	No	Sí	No	Sí	Sí	Sí	Sí
Proxies	No		Sí	Sí	Sí	Sí	Sí
Registros	Sí, manual	Sí, manual	Sí	Sí	Sí	Sí	Sí
Soporte para múltiples arquitecturas	No	No	No	No	Sí	No	No
Posibilidad de trabajar con BBDD de diferentes fabricantes	No	Sí	No	Sí	Sí	Sí	Sí
Conexiones múltiples	No	No	Sí	Sí	Sí	No	No
Uso de controladores nativos o externos	No	No nativos	No	No nativos	No nativos	No nativos	No nativos
Consultas SQL	No	Sí	Sí	Sí	Sí		Sí

Tabla 3: Caracterización de los esquemas de persistencia

Todos ellos son comparados según las características mencionadas por Ambler. Cómo podemos comprobar, los únicos sistemas de persistencia de objetos que cumplen la mayoría de las características son las capas de persistencia.

6.4. Conceptos de objetos

6.4.1. Instancia y transitoria

El concepto de instancia persistente se asocia a aquellas cuyos datos se mantienen a la ejecución del proceso que materializó la instancia. Una instancia temporal o

transitoria, es aquella cuyos datos desaparecen cuando finalizan los procesos que la gestionan. En los dos casos, las instancias desaparecen cuando terminan los procesos asociados a ellas que las crearon.

Como ejemplo podemos tomar la ejecución de un programa que solicita leer nuestro número telefónico, obviamos el tema de ser usado en muchas operaciones de nuestro programa o hilo de ejecución.

Si el dato es usado en instancia transitoria, al volver a ejecutar el programa, tendremos que volver a introducirlo, pero por el contrario, si este programa es asociado a una instancia persistente, el dato puede ser recuperado y usado en la ejecución de nuestro nuevo hilo de programa.

6.4.2. Servicio de objetos

El estatus con el que se califica al concepto de servicio de persistencia no se encuentra del todo definido y/o aclarado en todos los ámbitos. La separación entre el concepto de servicio y el de base de datos no está clara. De algún modo, intentando establecer una aclaración sobre el tema de manera más aceptada posible podemos afirmar que, el servicio de persistencia es un sistema o mecanismo programado para posibilitar una interfaz única para el almacenamiento, recuperación, actualización y eliminación del estado de los objetos que pueden ser persistentes en uno o más sistemas gestores de datos.

La definición considera que la base de datos puede ser de tipo SGBDR, SGBDOO, o similar. El estado podría estar repartido entre varios sistemas, también de tipo diverso. Un servicio de persistencia de objetos aporta los elementos necesarios para efectuar la modificación y la eliminación de los objetos persistentes, además del volcado y recuperación del estado en los sistemas gestores de datos. Y todo ello, debería ser efectuado de acuerdo a la definición hecha más atrás de persistencia, sin necesidad de traducción explicita por parte del programador. En todos los casos, sea cual sea el tipo de gestor datos, los servicios de persistencia de objetos facilitan la ilusión de trabajar con un sistema de bases de datos de objetos integrado con el lenguaje de programación, ocultando las diferencias entre el modelo de objetos del lenguaje y el modelo de datos del sistema empleado como base de datos. A pesar de lo dicho, un servicio de persistencia no es un sistema de gestión de bases de datos orientado a objetos. El servicio de persistencia es un componente esencial de todo sistema gestor de bases de datos de objetos (SGBDOO), que resuelve otros aspectos, además de la persistencia.

De las alternativas estándar para hacer persistir los objetos en Java solo ODMG 3.0 y JDO pueden tener la consideración de servicio de persistencia de objetos Java. También es posible proponer utilizar un servicio de persistencia del OMG, PSS 2.0, para persistir objetos Java, pero este estándar considera opcionales las transacciones y la persistencia transparente, que son funciones necesarias para ofrecer un servicio de persistencia eficaz, conforme a las definiciones vistas de persistencia. JDBC, SQLJ requieren que el programador defina, e implemente las

operaciones de persistencia teniendo en cuenta cómo convertir los objetos en tuplas. Ambos pueden ser utilizados en la construcción de servicios de persistencia para bases de datos relacionales. La serialización es el servicio de persistencia más básico, solo ofrece servicios para guardar y recuperar el estado de un objeto sobre ficheros y flujos de entrada salida.

De otra parte, los SGBD ofrecen servicios de persistencia, que adoptan una o varias de las siguientes aproximaciones de cómo se puede hacer un objeto persistente:

- Por tipo. Un objeto puede llegar a ser persistente cuando es creado de algún tipo (clase) dotado de la capacidad de persistir o de un subtipo (clase descendiente) de estos. Los tipos persistentes son distintos de los tipos cuyas instancias son transitorias. El tipo podría ser identificado como persistente en su declaración, o por herencia de alguno de los tipos predeterminados por el sistema. De forma parecida, la serialización obliga que las clases implementen la interfaz Serializable.
- Por invocación explicita. El programador invoca un método que provoca que un objeto pase a ser persistente. La llamada al método puede ser en la creación del objeto o en cualquier instante, según su implementación.
- Por referencia. Un objeto es hecho persistente al ser referenciado por otro que es persistente. Añadir un objeto a una colección persistente, la asignación de un objeto a un atributo de otro persistente o la asignación a cierto tipo de referencias, provocan que un objeto transitorio pase a ser persistente.

6.4.3. Ortogonal

La característica de ortogonalidad de la persistencia se basa en que el uso de una no afecte a la otra, siendo independientes entre sí. Programas y persistencia serán ortogonales, si la forma en la que los objetos son manipulados por estos programas es independiente de la utilización de la persistencia, que los mismos mecanismos operaran tanto sobre objetos persistentes como sobre objetos transitorios, ambas categorías serían tratadas de la misma manera con independencia de su característica de persistencia. Ser persistente debería ser una característica intrínseca del objeto, soportada por la infraestructura del entorno de programación y persistencia. La persistencia de un objeto debe ser ortogonal al uso, tipo e identificación. Esto es, cualquier objeto debería poder existir el tiempo que sea preciso, ser manipulado sin tener en cuenta, si la duración de su vida, supera al proceso que lo creo, y su identificación no estar vinculada al sistema de tipos, como la posibilidad dar nombres a los objetos.

A la hora de plasmar el uso de la persistencia en nuestros programas, una persistencia ortogonal ideal, llevaría a no tener que modificar el código de nuestras clases, salvo aquellas donde debamos introducir las operaciones que provocan la persistencia para cualquier objeto que sea duradero. Los beneficios que aporta la persistencia ortogonal son importantes: mayores cotas de facilidad de mantenimiento, corrección, continuidad del código y productividad de desarrollo. Se consigue:

- Menos código. Una semántica para expresar las operaciones de persistencia más simple de usar y entender. Evita la duplicidad de código uno preparado para instancias transitorias y otro para instancias persistentes.
- Evitar la traducción explicita entre el estado de los objetos y su representación en base de datos, que redunda en mayor facilidad de mantenimiento y menos código también.
- Facilitar la integridad y permitir que actúe el sistema de tipos subyacente, que automáticamente podría verificar la consistencia y la correspondencia de tipos, entre estados en base de datos y objetos en programa, la integridad no sería responsabilidad del programador.

Todo lo visto en este apartado apunta la conveniencia de usar persistencia ortogonal. En la práctica conseguir persistencia ortogonal completa no es fácil, habitualmente encontraremos limitaciones.
Habrá clases de objetos que no son soportadas por los servicios de persistencia, bien por compromisos de diseño, como la dificultad de implementación; bien porque cabe pensar que determinados objetos no tienen sentido fuera del contexto de ejecución concreto de un proceso, como por ejemplo un puerto de comunicaciones para IP.

6.4.4. Cierre

La construcción avanzada de objetos complejos suele hacerse a partir de objetos más simples. Esto plantea estructuras de objetos que necesitan una relativa gestión recursiva. Cada objeto puede tener un gran número de objetos dependientes de manera directa e indirecta. Esta relación de dependencias es parte integrante del estado de cada objeto. Cuando el estado de un objeto es salvado o recuperado, sus dependencias también deberían ser guardadas o recuperadas. De otro modo, cuando el objeto fuese recuperado, llegaría a estar incompleto, sería inconsistente con respecto a cómo fue guardado.

Un mecanismo de persistencia que posibilita la persistencia automática de las dependencias de un objeto, que deban persistir, se dice que admite el cierre de persistencia. Cuando el estado de un objeto es almacenado, los estados de los objetos dependientes que tengan que ser persistentes, son también almacenados, y así, sucesivamente. En otro sentido, en la recuperación del estado de un objeto, los estados de los objetos dependientes son recuperados. El cierre de persistencia determina el conjunto de referencias necesario, que ayuda a conseguir la consistencia entre el estado del objeto en el instante de guardar y estado resultante de su recuperación.

6.4.5. Alcance

La persistencia por alcance o persistencia en profundidad es el proceso de convertir automáticamente en persistente todo objeto referenciado directa o indirectamente por un objeto persistente, los objetos del cierre de persistencia de un objeto son

hechos persistentes. Es la aplicación recurrente de la estrategia de persistencia por referencia.

Las clases cuyos objetos se almacenen en la base de datos deben ser descritas en el esquema de objetos asociado con la aplicación, y además deben ser identificadas como capaces de ser persistentes.

En el binding para Java la persistencia es por alcance, con lo que un objeto Java transitorio referenciado por un objeto Java persistente será persistente si es una instancia de una clase capaz de ser persistente, y los dos objetos no están ligados por un campo transitorio. Y tal y como propone el estándar, dentro de una clase persistente puede haber campos transitorios.

Por otro lado, la recuperación de un objeto de la base de datos no implica la recuperación de todos los objetos asociados a éste. Cuando se recupera un objeto, todos los objetos relacionados con éste son representados en la aplicación cliente con Proxies. Cuando se accede a un proxy el objeto correspondiente es cargado en la memoria local.

6.4.6. Transparencia de datos

Cuando un sistema o entorno de programación ofrece transparencia de datos, el conjunto de las clases persistentes y el esquema de la base de datos es uno, las clases definen de hecho el esquema en la base de datos. Los estados almacenados en la base de datos son manejados con el lenguaje de programación elegido, no es necesario otro. Los objetos son recuperados de la base de datos automáticamente, cuando las referencias a estos son accedidas. También, las modificaciones del estado de objetos persistentes son reflejadas en la base de datos automáticamente. Los estados de los objetos son recuperados y actualizados de forma transparente; no hay cambios en la semántica de referencia o de asignación en la aplicación. Objetos transitorios y persistentes son manipulados de igual forma. Las operaciones propias de la persistencia son efectuadas sin la intervención directa del programador, con más código. La frontera entre el lenguaje de programación y los servicios de datos desaparece a los ojos del programador, evitando la falta de correspondencia (impedance mismatch) entre la base de datos y lenguaje de programación.

No debemos confundir transparencia con persistencia ortogonal, la primera es una consecuencia de la segunda. Podemos encontrar transparencia de datos sin disponer de persistencia ortogonal total, es el caso de que determinadas clases no puedan persistir, esto concretamente supone que la solución no sería ortogonal, independiente, respecto el tipo.

6.4.7. Falta de correspondencia entre clases y datos

Cuando se trabaja con sistemas gestores de datos, como bases de datos relacionales, ficheros, bases de datos documentales XML, etc., cuyo modelo datos no

tiene una equivalencia directa con el modelo de objetos del lenguaje de programación usado, hay una falta de correspondencia (impedance mismatch) entre la base de datos y lenguaje de programación, es necesario establecer un modelo de correspondencia, que defina como una clase se convierte en datos sobre el modelo ofrecido por sistema que albergará el estado de los objetos. A esta equivalencia entre la clase y los datos, se denomina aquí correspondencia clase – datos (object mapping). El caso particular de la correspondencia entre clases y tablas en un SGBDR, es la correspondencia objeto - registros. Se puede utilizar el término mapear para señalar al proceso de definición de la correspondencia entre las clases de los objetos persistentes y los modelos de datos, el proceso puede implicar cambios en ambos lados de la correspondencia, en el modelo de clases creado o modificado para asumir ciertos modelos de datos, y al contrario, el modelo de datos puede ser diseñado o cambiado, para permitir una correspondencia más eficiente y eficaz con ciertos modelos de clases.

Las definiciones de persistencia que se han visto invitan a emplear una persistencia que no cambie la forma de trabajar con el lenguaje de programación, que actúe de forma transparente y consistente, donde el modelo de clases y el modelo de datos son la misma cosa, y que sea una persistencia ortogonal. Los beneficios serán un código con menos líneas, más fácil de mantener, más correcto y productivo.

6.5. Mecanismos

Como se puede observar se trata de una categorización en dos niveles que incluye tanto técnicas establecidas como ser acceso directo a bases de datos relacionales (una subcategoría de acceso directo a bases de datos) como también técnicas más recientes y de creciente popularidad (p.ej. generadores de código y mapeadores objeto-relacional).

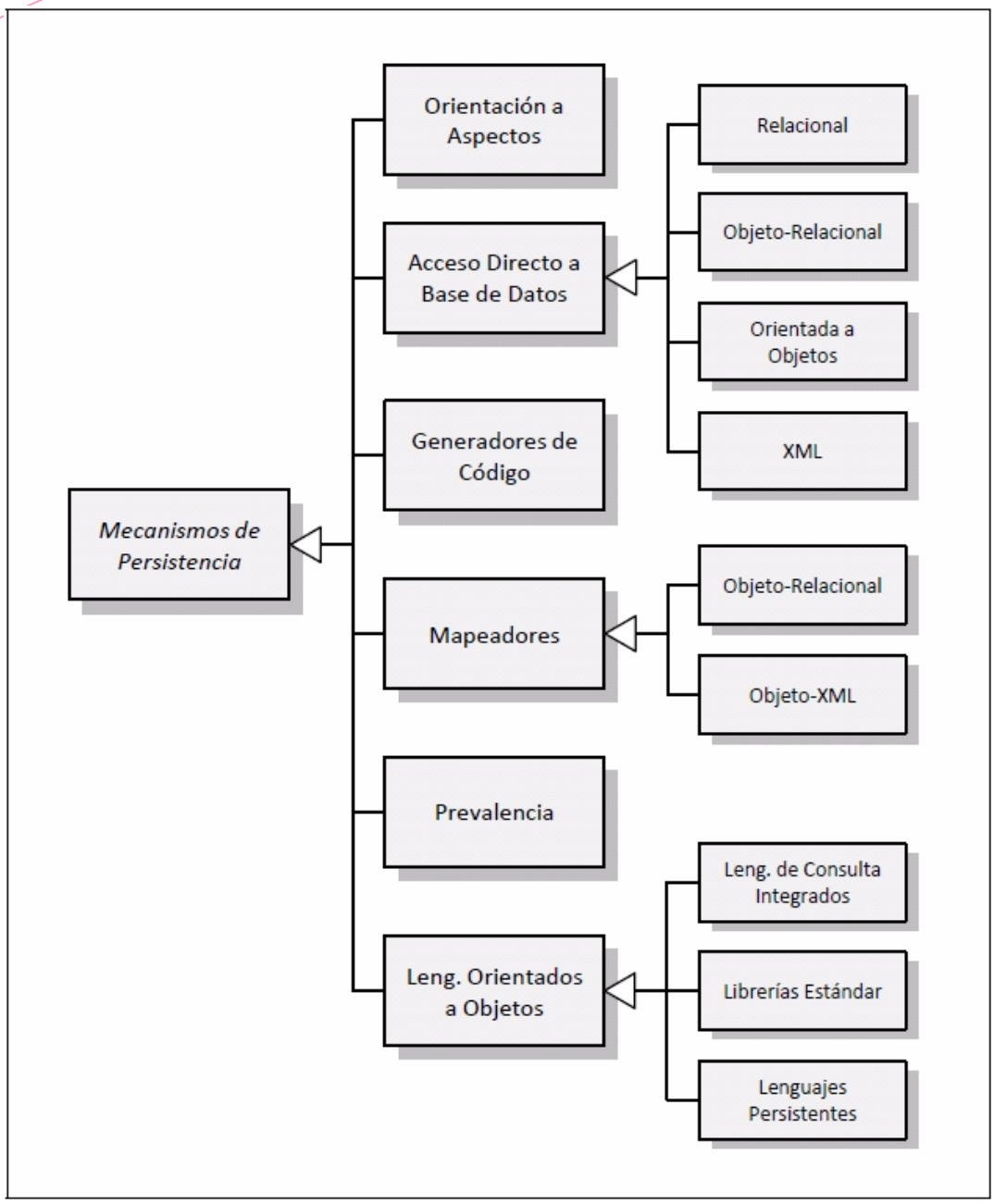


Figura 7: Mecanismos de persistencia, agrupados por herencia

La existencia de un segundo nivel permite separar técnicas que mas allá de compartir una misma técnica base (p. ej. en el caso de los mapeadores) tienen características diferenciadas (el caso de los mapeadores objeto-relacional y objeto-XML). Concretamente, la clasificación propuesta define categorías que contemplan: Mapeadores (mapeo de objetos a otro modelo de datos), Acceso Directo a Base de Datos (interacción directa con una base de datos), Lenguajes Orientados a Objetos (soporte del lenguaje para la persistencia de objetos), Orientación a Aspectos (implementación de persistencia a través de aspectos), Generadores de Código (generación del código para persistencia a través de herramientas) y Prevalencia (persistencia mediante snapshots y fuerte uso de memoria principal).

6.5.1. Acceso directo a base de datos

El mecanismo en cuestión implica el uso directo de la base de datos para implementar la persistencia del sistema. Esto último refiere al acceso a los datos utilizando por ejemplo una interfaz estandarizada en conjunto con algún lenguaje de consulta soportado directamente por el DBMS. Téngase en cuenta que no existe ningún tipo de middleware entre la aplicación y el DBMS utilizado.

- Acceso Directo a Base de Datos Relacional. Haciendo uso de una base de datos relacional como dispositivo de almacenamiento, es la intención fundamental proveer un mecanismo de acceso a datos que maneje el modelo relacional.
- Acceso Directo a Base de Datos Objeto-Relacional. Haciendo uso de una base de datos objeto-relacional como dispositivo de almacenamiento, es la intención fundamental proveer un mecanismo de acceso a datos que brinde un modelo de datos más rico que el relacional.
- Acceso Directo a Base de Datos Orientada a Objetos. Haciendo uso de una base de datos orientada a objetos como dispositivo de almacenamiento, es la intención fundamental proveer un mecanismo de acceso a datos que no implique una diferencia de paradigmas entre el nivel lógico y de persistencia.
- Acceso Directo a Base de Datos XML. Haciendo uso de una base de datos XML como dispositivo de almacenamiento, es la intención fundamental proveer un mecanismo de persistencia que permita almacenar los datos de los objetos en documentos XML.

6.5.2. Mapeadores

En esta categoría se incluyen 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 ha de lidiar con los diferentes problemas que se presentan al mapear los datos entre paradigmas. El conjunto de problemas que se derivan de estas diferencias recibe el nombre de impedance mismatch.

- Mapeadores Objeto-Relacional. La intención de este mecanismo radica en contar con un mecanismo para mapear los objetos de la lógica de un sistema orientado a objetos a una base de datos relacional. Debido a las diferencias entre la representación tabular de información y la encapsulación de los datos en objetos (problema conocido como impedance mismatch objeto-relacional) debe considerarse alguna estrategia (manual o automática) para poder resolver estas diferencias e integrar ambas tecnologías.
- Mapeadores Objeto-XML. Es la intención de este mecanismo permitir el almacenamiento de los datos contenidos en los objetos de la lógica en forma de documentos XML. El mapeo objeto-XML provee un medio por el cual los datos de negocio pueden ser visualizados en su forma persistida (incluso sin la necesidad de un DBMS), al mismo tiempo de que se facilita el intercambio de dichos datos con otros sistemas.

A diferencia de los modelos como DOM y JDOM, los mapeadores objeto-XML considerados toman un enfoque centrado en los datos y no en la estructura del documento XML de forma tal que el desarrollador utiliza objetos de negocio que reflejan el contenido mismo de los documentos.

6.5.3. Generadores de código

El surgimiento de numerosos Frameworks que resuelven parte de la problemática del desarrollo de una aplicación empresarial y el creciente uso de patrones ha permitido el desarrollo de aplicaciones empresariales más robustas en menos tiempo. Sin embargo el programador aun se ve obligado a realizar tareas repetitivas. La idea de que el programador debe concentrarse en el código que representa la lógica de negocio es la principal motivación para el surgimiento de los generadores de código. Concretamente, se trata de herramientas que basándose en metadatos de un proyecto son capaces de generar código correcto, robusto y que aplica los patrones de diseño pertinentes. Este código generado se encarga de resolver parte de la problemática del sistema, en este caso la persistencia del mismo.

6.5.4. Orientación a Aspectos

La Programación Orientada a Aspectos (AOP) es un paradigma que permite definir abstracciones que encapsulen características que involucren a un grupo de componentes funcionales, es decir, que corten transversalmente al sistema (p.ej. seguridad, comunicación, replicación, etc.). En la programación orientada a aspectos, las clases son diseñadas e implementadas de forma separada a los aspectos requiriendo luego una fusión. Es intención de este mecanismo manipular concretamente la persistencia como un aspecto ortogonal a las funcionalidades, desacoplando el código correspondiente al resto del sistema. Otro objetivo de este mecanismo es, a su vez, permitir modularizar la persistencia para luego poder reutilizar el código generado.

El principal objetivo de la POA es la separación de las funcionalidades dentro del sistema:

- Por un lado funcionalidades comunes utilizadas a lo largo de la aplicación.
- Por otro lado, las funcionalidades propias de cada módulo.

Cada funcionalidad común se encapsulará en una entidad.

6.5.5. Lenguajes orientados a objetos

La categoría de lenguajes orientados a objetos como mecanismo de persistencia implica resolver la persistencia de datos utilizando funcionalidades provistas por el propio lenguaje de programación.

- Librerías Estándar. La idea fundamental de este mecanismo yace en utilizar las funcionalidades básicas incluidas en la infraestructura de un lenguaje de programación orientado a objetos a fin de resolver la persistencia de un

sistema. Se trata luego de una estrategia simple que evita la inclusión de Frameworks u otras herramientas ajenas al lenguaje. Tomando como ejemplo el lenguaje de programación Java, se tienen librerías para el manejo de archivos que permiten persistir los datos de una aplicación en forma de texto plano, CSV (Comma Separated Value) o XML. Asimismo se pone a disposición del desarrollador la posibilidad de utilizar técnicas de serialización de objetos.

- Lenguajes Persistentes. El objeto de este mecanismo implica resolver de manera transparente la persistencia de datos haciendo uso de funcionalidades provistas por el lenguaje de programación, cumpliendo con los tres principios de persistencia ortogonal (o en su defecto, una relajación de los mismos): ortogonalidad de tipo, persistencia por alcance e independencia de persistencia. En concreto, dichos principios implican respectivamente que: (1) Cada objeto, independiente de su tipo, tiene el mismo derecho a ser persistido. (2) Se persisten solo aquellos objetos que son alcanzables desde la raíz de persistencia. (3) La introducción de persistencia no puede introducir cambios semánticos en el código fuente.
- Lenguaje de Consulta Integrado. Existen múltiples interfaces de acceso a datos que son utilizadas desde numerosos lenguajes de programación como método para administrar y consultar los datos persistentes de un sistema. El uso de interfaces de este tipo tiene como consecuencia una disminución en la claridad de la solución. Esto debido no solo al agregado en el código fuente de llamadas a operaciones definidas por el API en particular, sino también por la inclusión, en forma ajena al lenguaje de programación, de lenguajes de consulta como ser SQL, OQL, HQL, etc. Tomando en cuenta lo anterior, el mecanismo propone la utilización de un lenguaje de programación que integre como parte del mismo un lenguaje de consulta y administración de datos persistentes.

6.5.6. Prevalencia

La prevalencia de objetos surge como una propuesta diferente a utilizar bases de datos para lograr la persistencia. La mayoría de los programas operan sobre datos en memoria principal, ya que de esa forma logran procesar los datos más rápido que en un escenario donde se trabaja directamente en memoria secundaria. Este mecanismo permite mantener las instancias de objetos que componen a un sistema en memoria principal, añadiendo en forma periódica, la persistencia del estado de los anteriores mediante técnicas de serialización. La persistencia del estado de los objetos se denomina snapshot de los objetos en memoria, y representa el último estado consistente de los datos. Asimismo, permite el manejo de transacciones y la capacidad de recuperar un estado consiste del sistema tras la caída del mismo.

7. Framework de persistencia

La persistencia de datos en Java viene facilitada por mapeadores Objeto/Relacionales (O/R). Estas tecnologías conforman técnicas de programación para enlazar un lenguaje de programación orientado a objetos con una base de datos relacional. Los mecanismos de mapeo O/R permiten al programador mantener

una perspectiva orientada a objetos y cuidar que se consiga aportar una solución a la lógica de negocio, eliminando el obstáculo que supone compatibilizar objetos con modelos relacionales. El Frameworks para la persistencia de datos se ocupa de todos los detalles que conllevaría desarrollar un mecanismo de mapeo personalizado.

Hoy en día, uno de los asuntos más debatidos y discutidos más apasionadamente en la industria del software es: ¿cuál de las tecnologías de persistencia (o Frameworks para el mapeo Objeto/Relacional) merece ser la alternativa dominante? Esta discusión ha creado una división en la comunidad Java.

Mientras esto siga así, dicho debate seguirá en pie y los desarrolladores de software deberán elegir la tecnología que mejor se adapte a sus necesidades, y esto se debe extender a cualquier componente que conforme la arquitectura del sistema completo, incluyendo el driver JDBC.

Consecuentemente, la capa de mayor criticidad y por consiguiente en la cual más trabajo se ha desarrollado en los últimos años es la de persistencia. Debido al choque de impedancia que se produce entre los objetos del modelo de negocio y los datos persistidos en una base de datos relacional, es que esta capa requiere un tratamiento particular.

Gran parte de los productos que se han generado atacan el problema del mapeo y acceso a los datos persistentes. Algunos de los más conocidos son:

- EJB Entity Beans
- JDBC
- SQLJ
- TopLink
- CocoBase
- Hibernate / nHibernate
- JPOX (JDO)
- Versant (JDO)
- OBJ
- Object Spaces

Debido a algunas limitaciones de EJB Entity Beans han surgido las otras alternativas. Básicamente los Entity Beans presentan la característica de ser usables cuando los modelos de dominio son simples, deben ser distribuidos y conectarse a otros sistemas. Su utilización es buena cuando existe un mapeo uno a uno con las tablas de la base de datos. Es decir cuando la granularidad es baja. No admiten herencia entre clases componentes. Por esta razón y debido a esta limitación han surgido otros productos. Debido a esto también es que se desaconseja su utilización.
Por ejemplo, TopLink es un producto muy utilizado en el mercado, ahora integrado al servidor de aplicaciones Oracle 9i AS. Permite entre otras cosas:

- Integración de EJB CMP
- Mapeo de objetos a múltiples tablas
- Uso de herencia

- Soporta bloqueo optimista de tablas
- Transacciones anidadas
- Permite adaptar el mapeo de datos entre objetos y tablas
- Permite realizar el diseño en ambas direcciones, objetos / tablas y tablas / objetos.
- Permite adaptar código SQL generado
- Permite el uso de Store Procedures
- Administra pool de objetos
- Tiene una desventaja, es dependiente de APIs propietarias.

Hibernate también es muy utilizado tanto en el ambiente java como .net.

JDO es una especificación java que se espera se convierta en el estándar de administración de bases de datos orientadas a objetos. No exige la implementación de interfaces. Presenta menor sobrecarga en la creación de objetos y su configuración se realiza en un archivo XML, siendo más simple que la de los EJB CMP.

Presenta ventajas que heredó de EJB tales como el encapsulamiento de datos de la aplicación, el mapeo a tablas, trabaja en transacciones delimitadas por los Session Beans, administra pool de objetos. También posee ventajas propias como trabajar con clases java genéricas y las hace persistentes, requiere menor infraestructura.

7.1. Situación actual

7.1.1. Hibernate

Hibernate busca solucionar el problema de la diferencia entre los dos modelos usados hoy en día para organizar y manipular datos: 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 todas las 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.

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.

- Herramienta libre bajo licencia LGPL
- Es una herramienta madura, creada en el 2001, y es una de las más extendidas.
- Es un motor de persistencia para Java, aunque hay una versión para la plataforma .net llamada Nhibernate.
- Compuesto de muchos proyectos como: core (proyecto principal), annotations, Entity manager, shards, validator ...
- Tiene un montón de plugins para eclipse y tareas de Ant para ayudar a la utilización y automatización de la persistencia. Están dentro del proyecto hibernatetools
- Hibernate soporta paradigmas de la orientación a objetos como son el polimorfismo y las asociaciones.
- Ofrece un lenguaje de consultas de datos llamado HQL (Hibernate Query Language).
- Se pueden crear filtros definidos por el usuario.
- Gracias al proyecto HibernateAnnotations se pueden utilizar anotaciones de JDK 5.0 junto con los ficheros XML para el mapeo de objetos.

7.1.2. OJB

ObJectRelationalBridge (OJB) es una herramienta de correspondencia de objeto / Relational que permite mantener una transparencia en la persistencia contra SGBS para objetos Java. OJB ha sido diseñado para uno conjunto grande de aplicaciones, de sistemas empotrados a aplicaciones 'rich client', hasta arquitecturas J2EE multinivel. OJB se integra con facilidad en servidores de aplicación J2EE. Soporta la consulta de JNDI datasources. Se pone a la venta con JTA completo y la integración de JCA. OJB puede ser usado dentro de JSPs, Servlets y SessionBeans. OJB provee el soporte especial para Bean Managed EntityBeans (BMP).

7.1.3. 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.

7.1.4. Castor

Castor es un Frameworks de datos Open Source para Java. Es la ruta más breve entre objetos de Java, documentos de XML y tablas de BBDD relacionales. Castor provee un vínculo de gestión de persistencia entre Java hacia XML, Java hacia SQL, y otros.

7.1.5. Cayenne: Professional Object Relational Mapping.

Cayenne es un Frameworks poderoso y completo para correspondencia entre Objetos Java y Mapeo Relacional. Es open Source y totalmente libre. Una de las diferencias de Cayenne principales son que viene con herramientas de modelado GUI, para plataformas cruzadas. Esta característica coloca a Cayenne en su propia liga, haciéndolo una elección muy atractiva sobre tantos productos de comerciales de código cerrado.

- Herramienta de código abierto de la fundación Apache.
- Portabilidad entre las bases de datos JDBC.
- Funcionalidad de cache, para hacer el acceso a la base de datos más eficiente
- Capacidad de paginación para cargar de la base de datos solo los objetos que se necesitan en el momento.
- Tutorial web paso por paso para aprender el manejo de esta herramienta.

7.1.6. TriActive JDO (TJDO)

Como características principales presenta:

- Implementación libre de la especificación JDO 1.0.1 de Sun.
- Implementa el lenguaje de consultas JDOQL, aunque permite consultas SQL directas.
- Auto-creación de todos los elementos del esquema necesarios (tablas, claves, índices).
- Auto-validación de la estructura del esquema en tiempo de ejecución.
- Capacidad de mapear clases de Java a Vistas SQL.

7.1.7. Jaxor

Jaxor es una herramienta de generación de código o de mapeo que toma la información definida en XML relacionando las entidades del modelo relacional para mapearlas y generar clases, interfaces y objetos y buscar objetos que puedan ser usados en cualquier aplicación Java (incluyendo JFC / Swing, J2EE y herramientas en línea de comando). Su gran velocidad de generación de código se debe al uso estructurado de plantillas y la generación mediante mecanismos de conversión fijos.

7.1.8. JDBM

JDBM es un motor de persistencia transaccional para Java. Aspira a ser para Java lo que GDBM es para otros lenguajes (C / C + +, Python, el lenguaje perl, etc.): Un motor de persistencia rápido y simple.

Todas actualizaciones son hechas en una manera transaccional segura. JDBM también provee estructuras de datos escalables, como HTree y B+Tree, para soportar la persistencia de grandes colecciones de objetos.

7.1.9. PBeans

PBeans es una capa de mapeo de BBDD Objeto/Relacional (O/R). Está diseñado para ser simple de usar y automatizado totalmente. La idea es que se ahorre tiempo y esfuerzo concentrándose solo en el desarrollo de clases Java, y no preocupándose por el mantenimiento o correlación de los scripts SQL, esquemas XML basados en, ni siquiera en la generación de código. El Frameworks pBeans se cuida de tomar muy poca ayuda del desarrollador para concretar la persistencia de los JavaBeans.

7.1.10. Simple ORM

SimpleORM es un proyecto Open Source para mapeo Objeto / Relacional en Java (la licencia de estilo de Apache). Provee una implementación simple pero eficaz de mapeo O/R sobre JDBC, de bajo costo.

No necesita de configuración mediante archivo XML.

Más que Java puro, SimpleORM es Java 100% depurado. Sin pre-procesamiento ni post-procesamiento de bytecode. Este Frameworks simple y elegante no requiere ningún truco ingenioso.

SimpleORM tiene mucho cuidado con semántica de BBDD. Bloqueo, calidad de la información, y almacenamiento son siempre tratados apropiadamente.

7.1.11. Java Ultra-Lite Persistence (JULP)

Un Frameworks de muy pequeño tamaño.

- Open Source (GPL)

- 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 7.3, Sybase ASE 12.0, MckoiDB 1.02, hsqdb 1.7.1)
- Herencia
- Muchas clases por tabla
- Muchas tablas por clase
- Asignaciones simples: la asignación lo único que necesita es catálogo>> <schema> table.column = fieldName.

7.1.12. Prevayler

El concepto de prevalencia implica el mantenimiento de la totalidad de las instancias de objetos que componen un sistema en memoria principal. La persistencia se realiza mediante snapshots del estado del sistema que se efectúan de forma periódica y que utilizan la técnica de serialización de los objetos en memoria. La primera implementación de prevalencia se desarrolló como un proyecto Java de código abierto, denominada Prevayler.

Prevayler que incluye funcionalidades como ser rollback automático ante fallos, volcados XML, y mejoras en la escalabilidad global.

7.1.13. 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.

7.1.14. IBatis SQL Maps

Maps de SQL de iBATIS provee unos medios muy simples y flexibles de cambiar de lugar los datos entre sus objetos de Java y una BBDD relacional. El mapeo de SQL realizado por el Frameworks ayuda a reducir la cantidad de código Java que normalmente hay que realizar para acceder a una BBDD relacional. Este Frameworks traza un mapeo entre JavaBeans a sentencias de SQL usando un descriptor de XML muy simple. La sencillez es la ventaja más grande del mapeo SQL sobre los otros Frameworks y otras herramientas de correspondencia O/R. Para usar SQL solo necesita estar familiarizado con JavaBeans, XML y SQL. Hay muy poco más para

aprender. No hay ningún plan complicado requerido para unir tablas o ejecutar consultas complicadas. Usando mapeo SQL se tiene el pleno poder de SQL.

El Frameworks de Mapeo de SQL puede trazar un mapa de casi cualquier base de datos a cualquier modelo de objeto y es muy tolerante con los diseños anticuados, o incluso diseños malos. Esto se consigue sin tablas de base de datos especiales o generación de código.

7.1.15. 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.

7.1.16. 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.

7.1.17. XORM

XORM es una capa de correspondencia O/R extensible para aplicaciones Java. Suministra la interfaz para persistencia basada en SGBDs mientras permite que desarrolladores se concentren en el modelo de objeto, no la capa física.

7.1.18. 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

7.1.19. JDO Genie

Genie de JDO es una implementación de JDO de máximo rendimiento que soporta BBDD relacionales de comerciales y open source. JDO Genie soporta JBoss, WebLogic, WebSphere, Tomcat, otros motores servlet y aplicaciones de 2-niveles.

7.1.20. LiDO

A diferencia de otras herramientas de correspondencia O/R, LiDO no sólo provee el acceso a BBDD relacionales, sino a cualquier otra fuente de datos a las que las demás no pueden acceder, como XML, Mainframe, ODBMS y texto plano.

7.1.21. JDO Toolkit

JDO es un componente básico fundamental para cualquier aplicación. Es la manera usual de realizar la persistencia de objeto en Java, tal y como los servlets son la manera usual de hacer request/response.

7.1.22. FrontierSuite

FrontierSuite, uno de los mejores Frameworks de persistencia para guardar objetos de Java en RDBMS.

Especializado en el manejo de la persistencia para aplicaciones de empresa, proporcionando motores de persistencia, mapeo O/R y soluciones de almacenamiento ObjectFrontier es diferente en la singularidad de su producto. Y la especialidad está sostenida por su compatibilidad objeto/relacional, una estrategia con la que pocos han sido exitosos.

7.1.23. JRelay

Almacena objetos gráficos de Java en SGBD relacionales con JRelay. JRelay usa las tecnologías más avanzadas en correspondencia O/R para conseguir máximas prestaciones y gran rapidez en la persistencia de objetos (JDO). JRelay combina la experiencia en capas objeto / relacional con la universalidad de un estándar abierto, que asegura la escalabilidad, la robustez y el futuro soporte para uno de las partes más críticas en toda aplicación: el manejo de la persistencia de datos de una aplicación.

7.1.24. IntelliBO

- Soporte para servidores de aplicaciones: WebLogic, WebSphere, SAP WAS, Oracle, JBoss, Apache Gerónimo, Tomcat ...
- Soporte para bases de datos JDBC genéricas, y además: Oracle, DB2, MS SQL, Informix, HSQLDB, postgreSQL, mySQL ....

- Tiene muchas opciones avanzadas para el mapeo de objetos, como división de un objeto simple en múltiples tablas, estrategias de mapeo de herencia (TablaSubclase, Tablasuperclase, Tablanueva), mapeo serializado, mapeo agregado.
- Opción de verificar las clases y las instrucciones de mapeo.
- Opción para comprobar la consistencia de la estructura de tabla actual.
- Generador automático de código fuente de las clases e instrucciones de mapeo a partir de las estructuras de tablas.
- Generador automático de las estructuras de tablas a partir de las clases e instrucciones de mapeo.
- Integración con Borland Jbuilder, Apache Ant, Eclipse e IBM WebSphere Application Developer.
- Ejemplos de proyectos y documentación comprensible.
- Edición profesional (de pago), edición comunidad (gratuita), plugins para eclipse.

7.1.25. KodoJDO

- Motor de persistencia para Java basado en el código de OpenJPA de Apache (JPA= Java Persistence Api).
- Soporte para la especificación JDO 2.0 y EJB 3.0 JPA.
- Herramienta comercial.
- Es capaz tanto de crear un esquema nuevo a partir de las clases, las clases a partir del esquema, o mapear clases existentes en un esquema existente.
- Diseñado para trabajar con o sin aplicaciones servidor.
- Soporte para múltiples bases de datos: Oracle, IBM DB2 y Informix, Microsoft SQL server …
- Integración con Eclipse, Borland Jbuilder, IBM WSAD y NetBeans.
- Soporte para servidores de aplicaciones J2EE: WebLogic, WebSphere, JKBoss…

7.1.26. Oracle

- Motor de persistencia para POJO (Plain Old Java Objects) ejb 2.1, cmp and 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.

7.2. Comparativa que implementa JDO

Características	FrontierSuite	Kodo	Lido	OpenFusion	TJDO	OJB
Especificación	1.0	2.0	1.0	1.0	1.0.1	1.0
RDBMS	Si	Si	Si	Si	Si	Si
ODBMS	No	Si	Si	No	No	No
Opciones especificación	Si	Si	Algunas	Algunas	Algunas	No
Entorno Administrativo	Si	Si	Si	Si	No	Si
Adaptable a entornos de programación	Si	Si	Si	No	No	No
Correspondencia reversible	Si	Si	No	No	No	Si
Versión evaluación	Completa	Completa	No es mantenida	No	Si	Si
Opciones propias	Si	Si	Si	Si	No	Si
MultiBase	Si	Si	Depende del fabricante	Si	Si	Si

Tabla 4: Implementación de JDO, comparativa

El mapeo objeto-relacional (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

A continuación establecemos un cuadro comparativo de los mapeadores de Hibernate, JPA/TopLink y LLBLGen Pro.

Criterio	Hibernate	JPA/TopLink	LLBLGen Pro
Nivel de Productos			
Última Versión	v3.2.1	v2.0b41	v2.0
Fecha de Lanzamiento	16-11-2006	11-05-2006	02-07-2006
Licencia	LGPL	CDDL	Comercial
Plataformas	Múltiples (Java)	Múltiples (Java)	Múltiples (.NET)
Nivel de Mapeadores			
Arquitectura de Cache	2 Niveles	2 Niveles	2 Niveles
Bulk Data Manipulation	Si	Si	Si
Consultas Nombradas	Si	Si	Si
Claves Primarias Compuestas	Si	Si	Si
Construcción Manual de SQL	No	No	No
Construcción GRUOP BY	Si	Si	No
Construcción por Defecto	Si	Si	Si
Consultas Dinámicas	Si	Si	Si
Consultas Polimórficas	Si	Si	Si
Creación Automática de Esquema	Si	Si	No
DBMSs Soportados	JDBC-compliant	JDBC-compliant	SQL Server, Oracle, PostSQL, Firebird, DB2 UDB, MySQL, MS Access
Estrategias de Fetching	{eager, lazy}	{eager, lazy[1]}	{eager, lazy}
Estrategias para Herencia	{table-per-class, table-per-hierarchy, table-per-subclass}	{table-per-class, table-per-hierarchy, table-per-subclass}	{table-per-class, table-per-hierarchy, table-per-subclass}
Funciones Agregadas	{avg, count, max, min, sum}	{avg, count, max, min, sum}	{avg, count, max, min, sum}

Generación Automática de ID	Si	Si	Si
Lenguaje de Consulta OO	Si (HQL)	Si (JPQL)	No
Mapeo de 1 Clase a N Tablas	Si	Si	Si
Mapeo de N Clases a 1 Tabla	Si[2]	Si[2]	Si
Objetos Asociados con Outer Joins	Si	Si	Si
Ortogonalidad de Tipo	Si	Si	Si
Persistencia de EJBs	Si	Si	No
Persistencia con Campos Privados	Si	Si	Si
Persistencia con get/set	Si	Si	No
Proceso/Generación de Código	No	No	Si
Relaciones Soportadas	{1-1, 1-n, m-n}	{1-1, 1-n, m-n}	{1-1, 1-n, m-n}
Soporte de Annotations	Si	Si	No
Soporte para Cascading	Si	Si	Si
Soporte para Detached Object	Si	Si	Si
Soporte para Optimistic Locking	Si	Si	Si
Soporte para Paginación	Si	Si	Si
Tablas Extra en la Base de Datos	Si	Si	Si
Tipos Personalizados	Si	Si	Si
Uniquing	Si	Si	Si

[1] Depende de si el proveedor de persistencia utilizado lo soporta.
[2] Sólo una clase puede modificar los datos, el resto se restringe a sólo lectura.

Tabla 5: Comparación de los Mapeadores de O/R principales

7.3. Evaluación comparativa

7.3.1. 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.

7.3.2. 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.

7.3.3. 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.

8. Diseño de un Framework de persistencia

8.1. Descripción general

Para el desarrollo del Frameworks de persistencia se ha utilizado como base el libro “The Design of a Robust Persistence Layer For Relational Databases” [Scott W. Ambler], así como la evaluación tanto de los requisitos como del código (cuándo este ha estado disponible) de algunos de los principales Frameworks de código open source en la actualidad.

Como todo debe tener una denominación, a este trabajo he decidido denominarlo SLGFP, es decir, Framework de Persistencia UOC.

Básicamente se ha perseguido desarrollar un Frameworks sencillo que abstraiga las características de la BBDD a la que se conecta y permita el encapsulamiento de las sentencias SQL hacia/desde la SGBD partiendo de un modelo genérico de comportamiento.

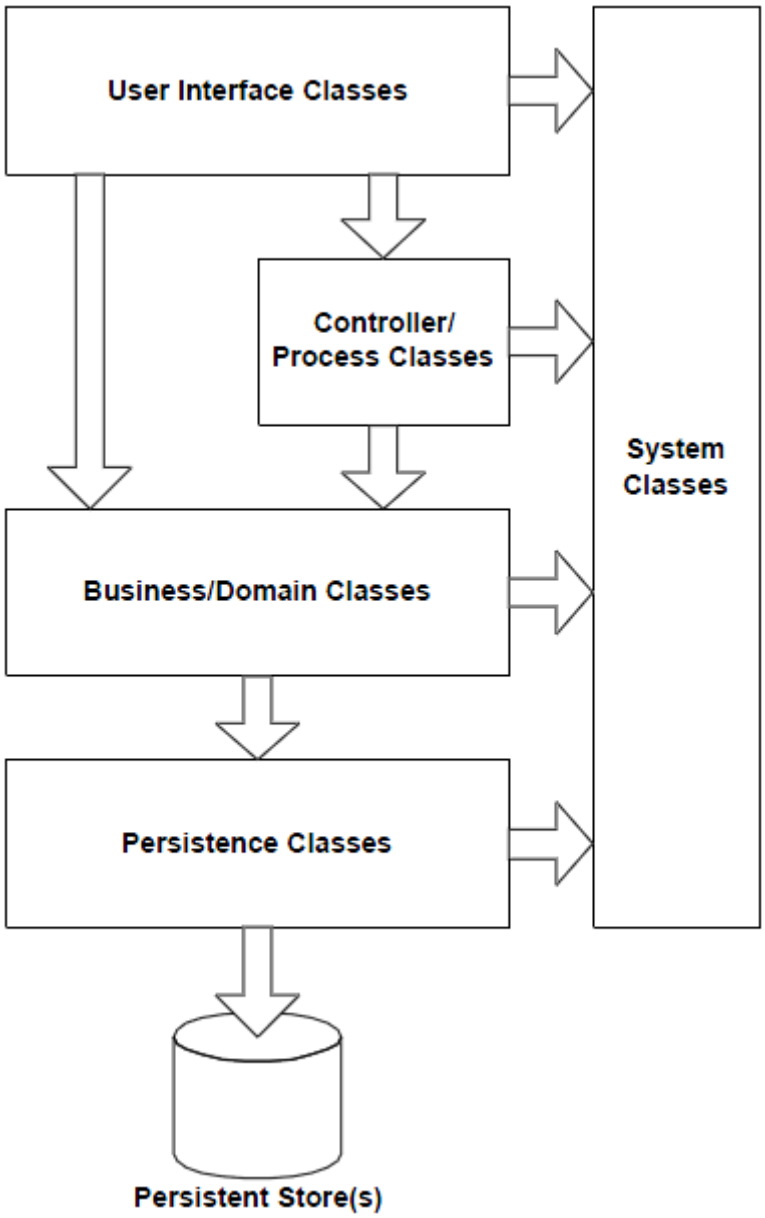


Figura 8: Arquitectura básica Framework persistencia [Scott W. Ambler]

8.2. Características principales del SLGFP

Las características que se han definido para este proyecto de desarrollo de Frameworks de persistencia han sido las más sencillas y básicas, contando entre ellas con:

- Habilidad para utilizar persistencia mediante POJO.
- Uso del patrón DAO como elemento estructural del diseño.
- 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 (como MySQL, ORACLE, SyBase,...).
- 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.
- El mapeo entre la BBDD y el Frameworks deberá ser muy simple:

<catalogo>.<esquema>.tabla.columna=nombreCampo

Por ejemplo, el mapeo sería de la siguiente forma:

```
CREATE TABLE PRODUCTO(ID INTEGER NOT NULL PRIMARY KEY,NOMBRE VARCHAR,PRECIO
DECIMAL)
...
public class Producto{
Integer x_producto;
String nombreProducto;
double importe;
...
}
...
ProductoFactory factory = new ProductoFactory(Producto.class);
Map mapping = new HashMap();
mapping.put("PRODUCTO.ID", "x_producto");
mapping.put("PRODUCTO.NOMBRE", "nombreProducto");
mapping.put("PRODUCTO.IMPORTE", "importe");
factory.setMapping(mapping);
...
```

8.3. Presentación de las clases del SLGFP

El paquete SLGFP contiene 17 clases únicamente, de las cuales 4 son interfaces.

El modelo general de interacción, la arquitectura de comportamiento de las clases es el que se muestra en la figura:

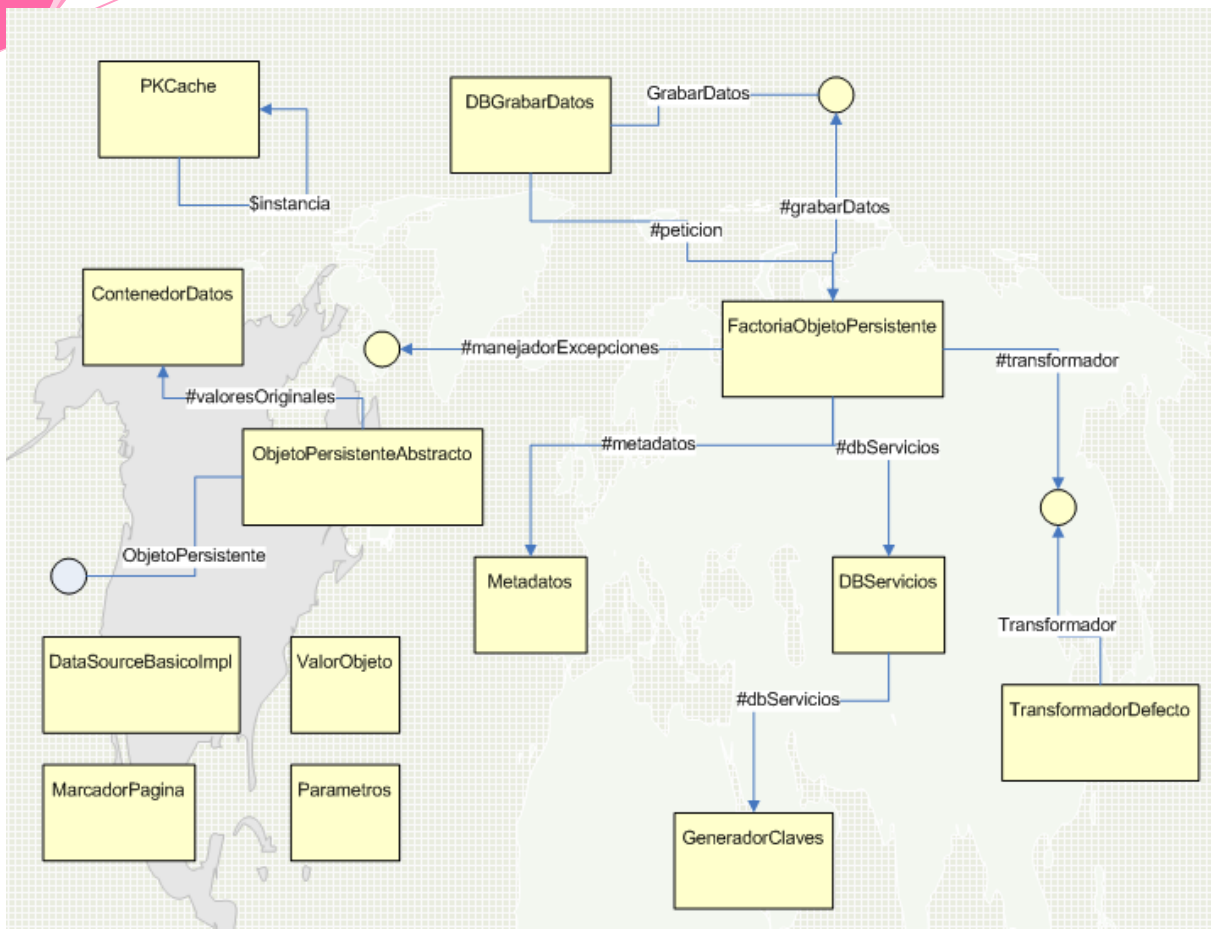


Figura 9: Arquitectura general del paquete SLGFP

8.4. Diseño de las clases del SLGFP

8.4.1. Diagramas de Jerarquía

Figura 10: Jerarquía de herencia de DBServicios

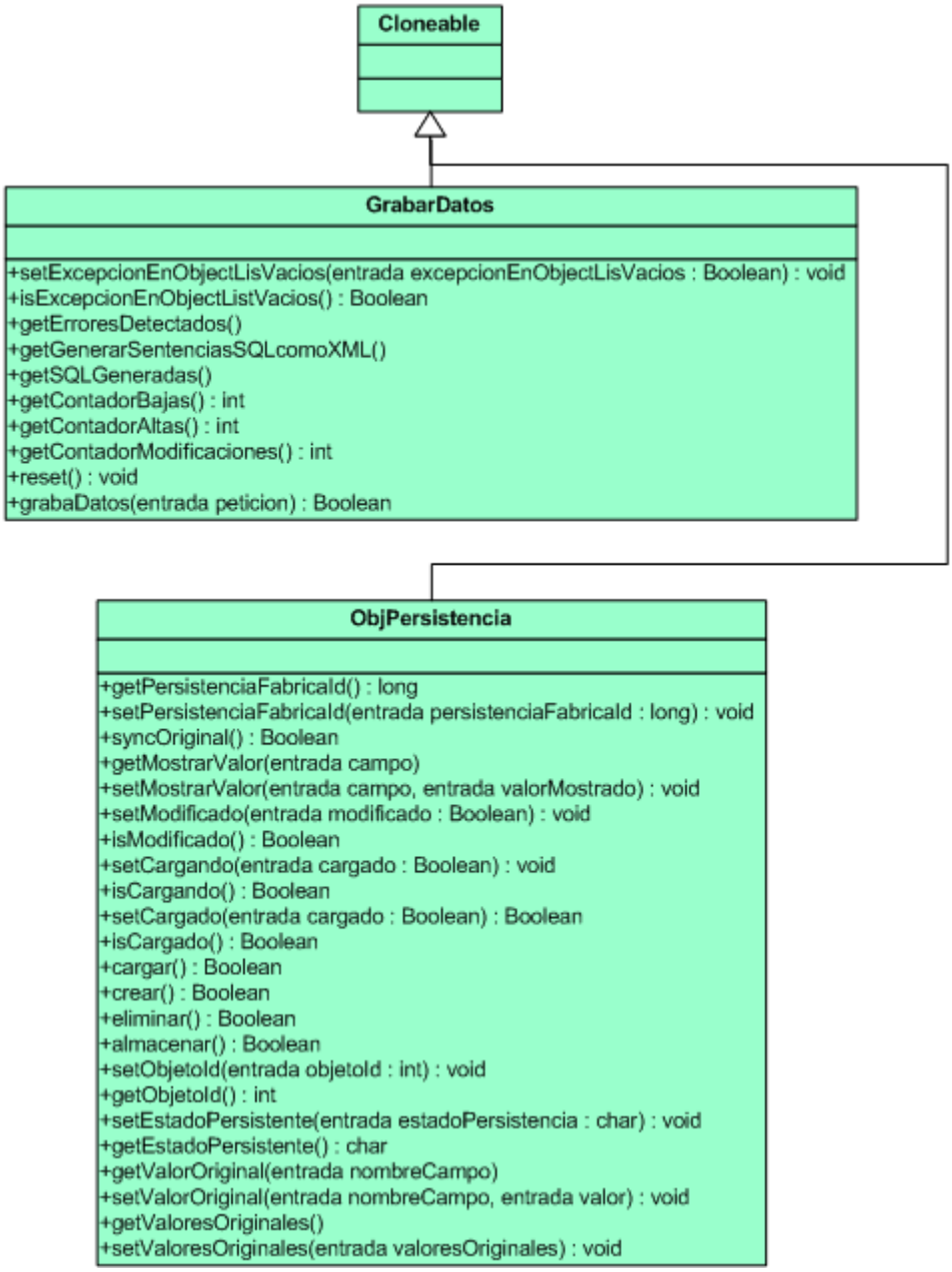


Figura 11: Jerarquía de herencia de ObjPersistencia y GrabarDatos

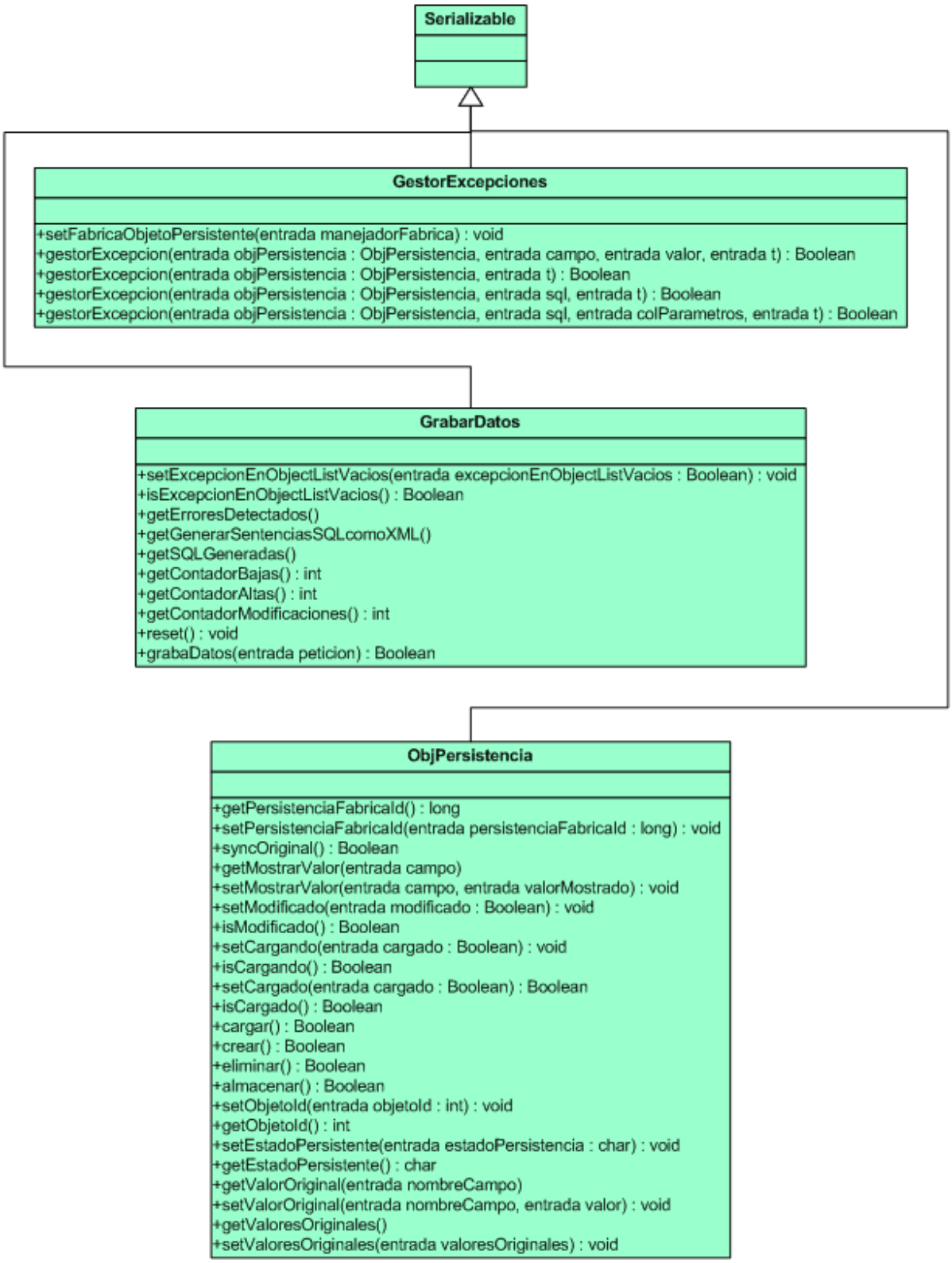


Figura 12: Jerarquía de herencia dependiente de Serializable

8.4.2. Diagramas de paquetes

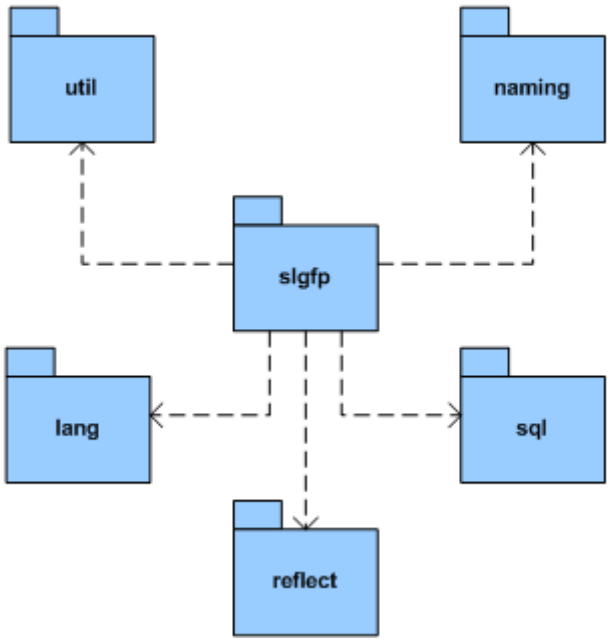


Figura 13: Estructura de paquetes en torno al SLGFP

8.4.3. Diagramas de clases

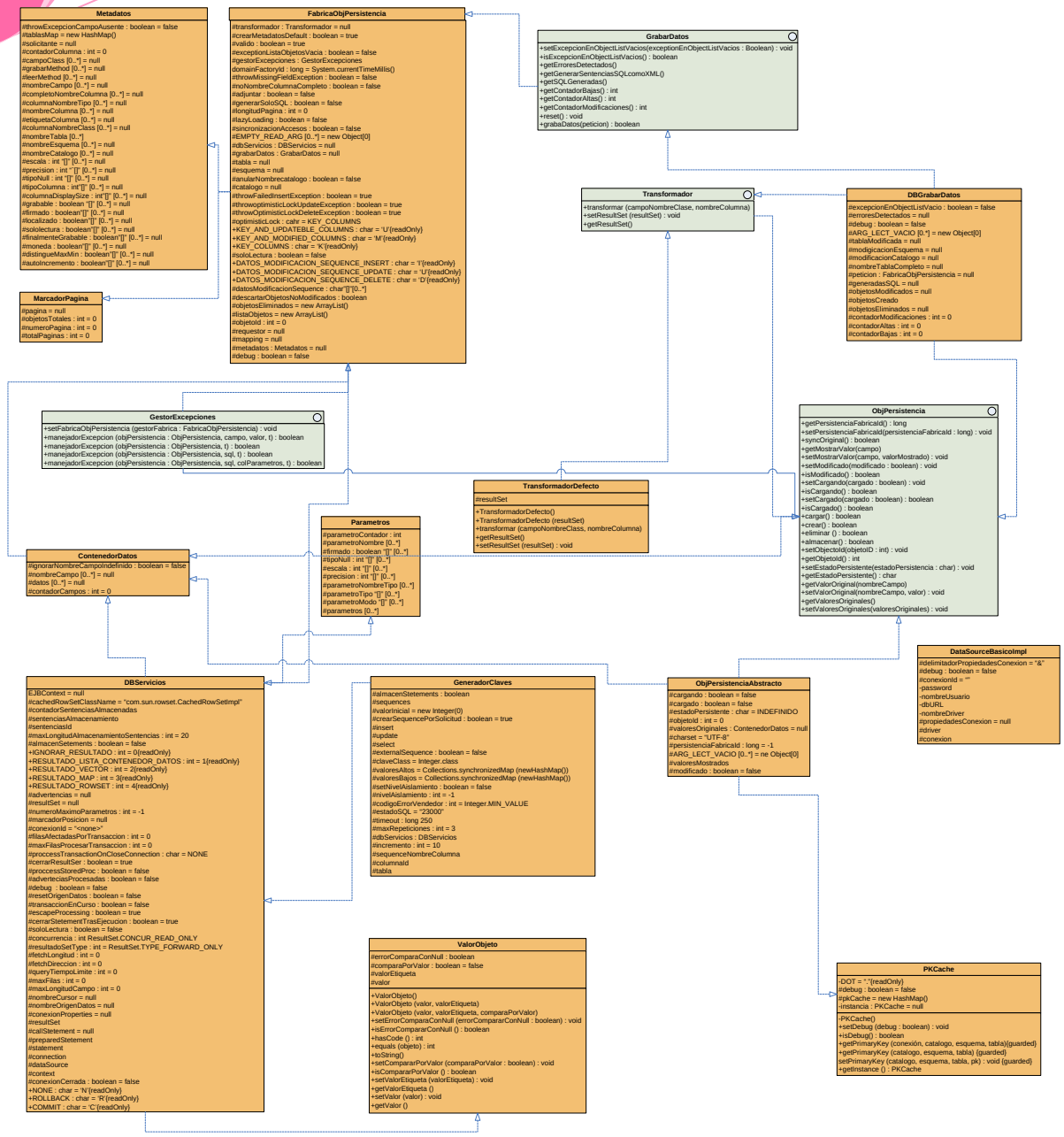


Figura 14: Diagrama general de dependencia de clases

8.4.4. Diagramas de componentes

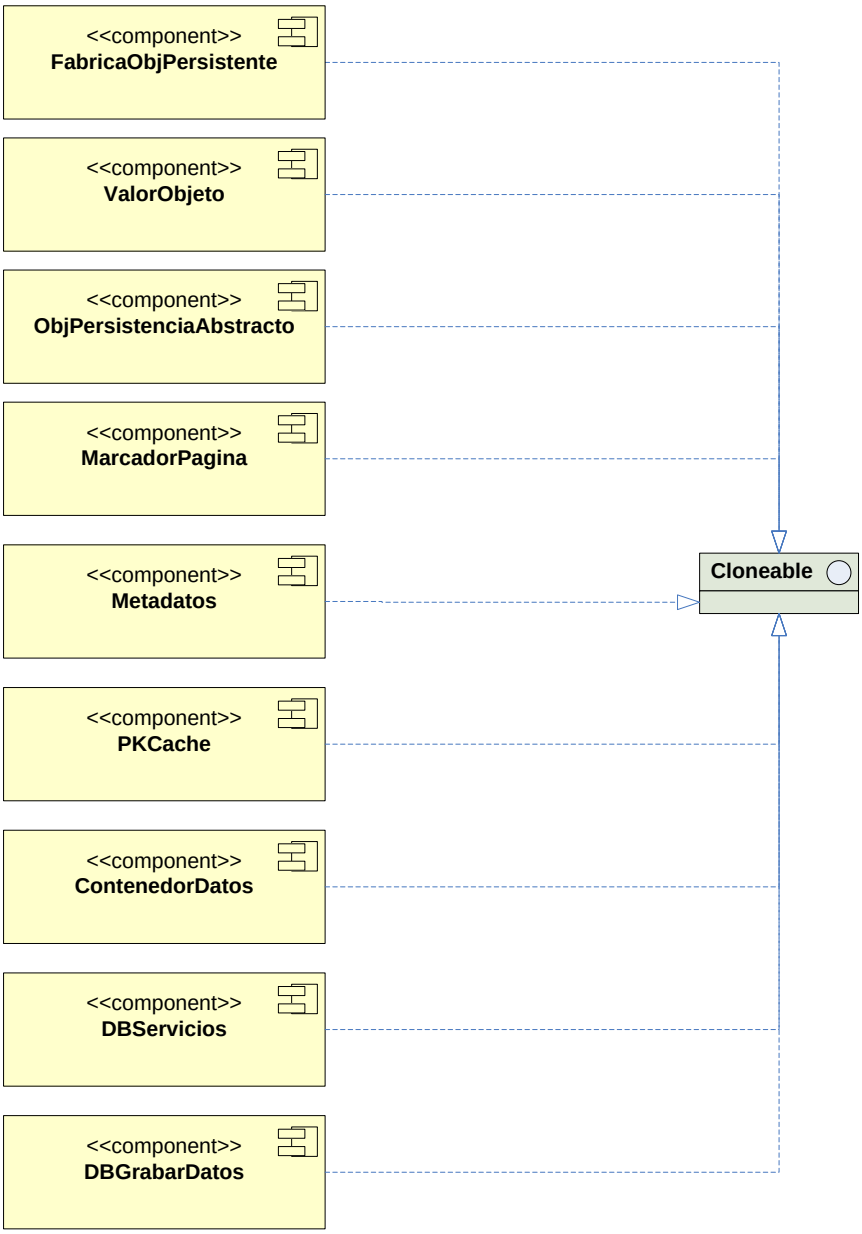


Figura 15: Diagrama general de componentes

8.4.5. Diagrama de realización de algunas clases

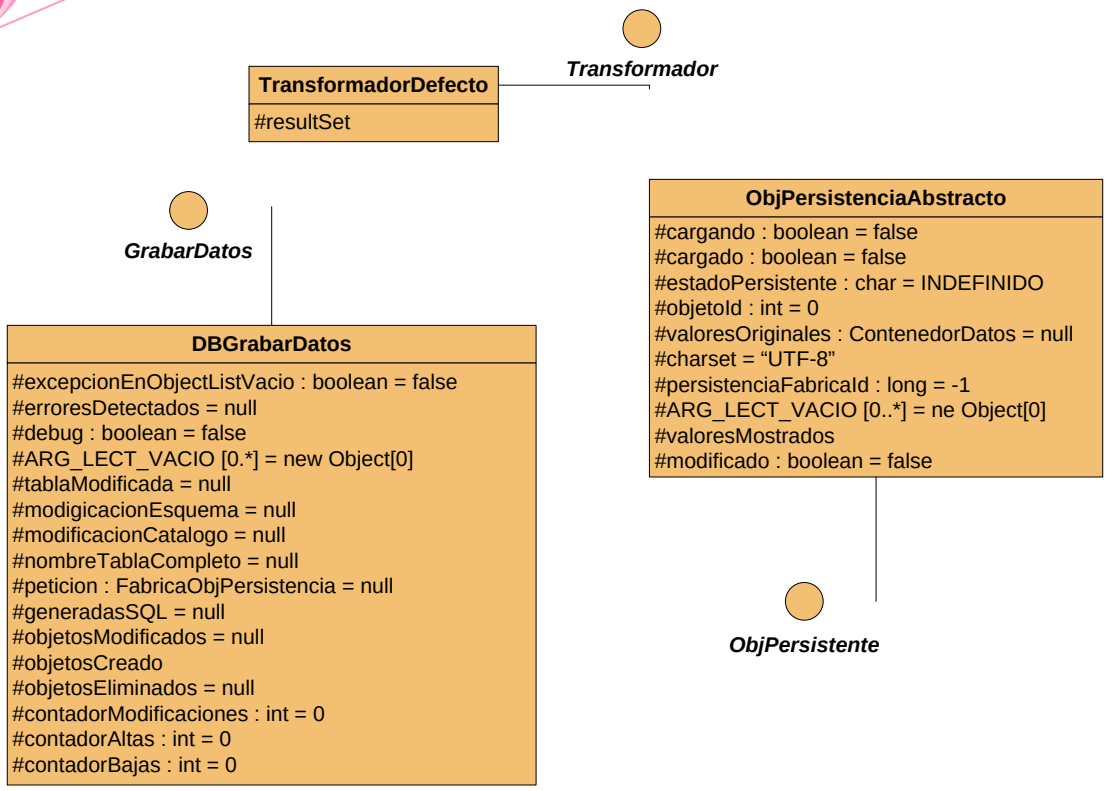


Figura 16: Ejemplo del diagrama general de una clase

8.5. Uso del SLGFP

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

1. La tabla que queramos leer debe ser referenciada por un objeto que se debe extender de la clase slgfp.ObjPersistenciaAbstracto.
2. Para su uso crearemos un objeto que extienda de la clase slgfp.FabricaObjPersistencia.
3. Por último, crearemos un fichero con el nombre_de_la_tabla.propiedades que es te tipo properties, donde le daremos la correlación entre el nombre de los campos de la tabla y el nombre de los campos del objeto.

En el paquete slgfp.pruebas podemos encontrar varios ejemplos que explican su uso. Uno de ellos CLIENTES.JAVA es el siguiente:

```
package slgfp.pruebas;

@SuppressWarnings("serial")
public class Clientes extends slgfp.ObjPersistenciaAbstracto implements
                java.io.Serializable, Cloneable {

        public Clientes() {
        }

        protected java.lang.Integer idCliente;
        protected java.lang.String nombre;
        protected java.lang.String apellidos;
        protected java.lang.String direccion;
        protected java.lang.String ciudad;
```

```
	public java.lang.Integer getIdCliente() {
		return this.idCliente;
	}

	public void setIdCliente(java.lang.Integer idCliente) {
		if (!isCargando()) {
			if (idCliente == null) {
				throw new IllegalArgumentException("Falta el campo:
clientesId");
			}
			if (!idCliente.equals(this.idCliente)) {
				this.modificado = true;
			}
		}
		this.idCliente = idCliente;
	}

	public java.lang.String getNombre() {
		return this.nombre;
	}

	public void setNombre(java.lang.String nombre) {
		if (!isCargando()) {
			if (nombre == null) {
				throw new IllegalArgumentException("Falta el campo: nombre");
			}
			if (!nombre.equals(this.nombre)) {
				this.modificado = true;
			}
		}
		this.nombre = nombre;
	}

	public java.lang.String getApellidos() {
		return this.apellidos;
	}

	public void setApellidos(java.lang.String apellidos) {
		if (!isCargando()) {
			if (apellidos == null) {
				throw new IllegalArgumentException("Falta el campo: apellidos");
			}
			if (!apellidos.equals(this.apellidos)) {
				this.modificado = true;
			}
		}
		this.apellidos = apellidos;
	}

	public java.lang.String getDireccion() {
		return this.direccion;
	}

	public void setDireccion(java.lang.String direccion) {
		if (!isCargando()) {
			if (direccion == null) {
				throw new IllegalArgumentException("Falta el campo: direccion");
			}
			if (!direccion.equals(this.direccion)) {
				this.modificado = true;
			}
		}
		this.direccion = direccion;
	}

	public java.lang.String getCiudad() {
		return this.ciudad;
	}

	public void setCiudad(java.lang.String ciudad) {
		if (!isCargando()) {
			if (ciudad == null) {
				throw new IllegalArgumentException("Falta el campo: ciudad");
			}
			if (!ciudad.equals(this.ciudad)) {
				this.modificado = true;
			}
```

```
			}
			this.ciudad = ciudad;
		}
}
```

Y del mismo modo, su fichero asociado FABRICACLIENTE.JAVA:

```
package slgfp.pruebas;

import java.util.*;
import slgfp.*;
import slgfp.buscar.*;
import java.sql.*;

@SuppressWarnings({ "serial", "rawtypes", "unchecked" })
public class FabricaDeClientes extends slgfp.FabricaObjPersistencia implements
		java.io.Serializable, Cloneable {

	// El objeto principal con las asignaciones de CAMPO_OBJETO y CAMPO_TABLA
	protected Properties sqlMap = null;

	// Constructor principal sin parámetros
	public FabricaDeClientes() {
		/*
		 * No es necesario cargar las asignaciones de esta forma, podría ser
		 * cualquier cosa: XML, JNDI, bases de datos, etc..
		 */
		sqlMap = cargaAsignaciones("Clientes.sql");
		initFactory();
		setSolicitante(Clientes.class);
	}

	// Inicializamos el acceso a la base de datos, el interfaz
	protected void initFactory() {
		try {
			DataSourceBasicoImpl ds = new DataSourceBasicoImpl();
			ds.setNombreDelDriver("org.hsqldb.jdbcDriver");
			ds.setDbURL("jdbc:hsqldb:hsql://localhost");
			ds.setNombreUsuario("sa");
			ds.setPassword("");
			this.dbServicios = new DBServicios();
			dbServicios.setDataSource(ds);
			// Fichero con las asignaciones de tipo PROPERTIES
			setAsignaciones(cargaAsignaciones("Clientes.propiedades"));
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public Properties cargaAsignaciones(String path) {
		java.io.InputStream inStream = null;
		Properties props = new Properties();
		try {
			inStream = this.getClass().getResourceAsStream(path);
			props.load(inStream);
		} catch (java.io.IOException ioe) {
			throw new RuntimeException(ioe);
		} finally {
			try {
				inStream.close();
			} catch (java.io.IOException ioe) {
				throw new RuntimeException(ioe);
			}
		}
		return props;
	}

	public void seleccionarComo() {
		System.out.println("\n====================== Este es un test de u"
				+ "so de ciertos métodos para rellenar las colecciones "
				+ "==========================\n");
		try {
			setAsignaciones(cargaAsignaciones("Clientes.propiedades"));
			String sql = sqlMap
					.getProperty("buscarTodosLosClientesConTelefono");
```

```
                        System.out.println("============== Como la lista de valores de "
                                        + "los objetos se muestran: (primer valor de la "
                                        + "columna y la segunda columna) ==================");
                        List list = dbServicios.getResultadoComoListaObjetoValor(sql);
                        System.out.println(list);
                        System.out.println("============== como vector de vectores "
                                        + "==================");
                        Vector vector1 = dbServicios.getResultadoComoVector(sql);
                        System.out.println(vector1);
                        System.out.println("============== En el mapa de las Listas ("
                                        + "key=nombre de columna, el valor=lista de "
                                        + "valores de la columna) ==================");
                        Map map = dbServicios.getResultadoComoMapa(sql);
                        System.out.println(map);
                        System.out.println("============== Como array de Contenedor"
                                        + "Datos ==================");

                        ContenedorDatos[] contenedorDatos = dbServicios
                                        .getResultAsDataHoldersArray(sql, false);
                        for (int i = 0; i < contenedorDatos.length; i++) {
                                System.out.println(contenedorDatos[i]);
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                        throw new RuntimeException(e);
                } finally {
                        try {
                                dbServicios.liberacion(true);
                        } catch (Exception ex) {
                                ex.printStackTrace();
                        }
                }
        }

        public void clausulaIn() {
                System.out
                                .println("\n======================== Probando la cláusula IN"
                                                + "======================== y usando SELECT con
numero de parámetros\n");
                try {
                        dbServicios.setMaxNumberOfParams(3);
                        dbServicios.setPlaceHolder(":#");
                        dbServicios.setCacheInstrucciones(true);
                        setSolicitante(Clientes.class);
                        setAsignaciones(cargaAsignaciones("Clientes.propiedades"));

                        List listaArg = new ArrayList();
                        String sql = sqlMap.getProperty("inClause");

                        List cliId = new ArrayList();
                        cliId.add(new Integer(0));
                        cliId.add(new Integer(1));
                        cliId.add(new Integer(2));
                        cliId.add(new Integer(3));
                        cliId.add(new Integer(4));
                        cliId.add(new Integer(5));
                        cliId.add(new Integer(6));
                        cliId.add(new Integer(7));

                        listaArg.add("Sergio Perez");
                        listaArg.add(cliId);

                        load(dbServicios.getResultSets(sql, listaArg));

                        Iterator it1 = getObjectList().iterator();
                        while (it1.hasNext()) {
                                System.out.println(it1.next());
                        }
                        dbServicios.liberacion(false);
                        System.out.println("==================================");

                        List list = dbServicios.getResultsAsDataHoldersList(sql, listaArg,
                                        false);

                        Iterator it2 = list.iterator();
                        while (it2.hasNext()) {
                                System.out.println(it2.next());
```

```
			}
		} catch (Exception e) {
			e.printStackTrace();
			throw new RuntimeException(e);
		} finally {
			try {
				dbServicios.liberacion(true);
			} catch (Exception ex) {
				ex.printStackTrace();
			}
		}
	}

	public int buscarTodosLosClientes() {
		int records = 0;
		try {
			String sql = sqlMap.getProperty("buscarTodosLosClientes");
			records = this.load(this.dbServicios.getResultSet(sql));
			imprimirTodosLosClientes();
		} catch (SQLException sqle) {
			throw new RuntimeException(sqle);
		}
		return records;
	}

	public int buscarTodosLosClientesConTelefono() {
		System.out.println("\n======================= Prueba en la que se "
				+ "usa la herencia ==========================\n");
		this.setSolicitante(ClientesConTelefono.class);

		setAsignaciones(cargaAsignaciones("ClientesConTelefono.propiedades"));

		int records = 0;
		try {
			records = this.load(this.dbServicios.getResultSet(sqlMap
					.getProperty("buscarTodosLosClientesConTelefono")));
			imprimirTodosLosClientes();
		} catch (SQLException sqle) {
			throw new RuntimeException(sqle);
		}
		return records;
	}

	public void crearYAlmacenarClientes() {
		System.out.println("\n======================= Prueba de creación y"
				+ " actualización de objetos (INSERT & UPDATE) ==="
				+ "=======================\n");
		this.setSolicitante(ClientesConTelefono.class);
		setAsignaciones(cargaAsignaciones("ClientesConTelefono.propiedades"));
		this.getDBServicios().setCacheInstrucciones(true);
		int records = buscarTodosLosClientes();
		ClientesConTelefono clientesConTelefono = new ClientesConTelefono();
		clientesConTelefono.setIdCliente(new Integer(records + 1));
		clientesConTelefono.setNombre("Sergio");
		clientesConTelefono.setApellidos("Sanchez");
		clientesConTelefono.setDireccion("Gran Avenida.");
		clientesConTelefono.setCiudad("Madrid");
		clientesConTelefono.setTelefono("987654321");

		this.crear(clientesConTelefono);
		System.out.println("\nCliente creado: " + clientesConTelefono + "\n");

		ListIterator li = this.objectList.listIterator();
		while (li.hasNext()) {
			ClientesConTelefono customer = (ClientesConTelefono) li.next();
			String lastName = customer.getApellidos();
			if (lastName.equals("Miller")) {
				customer.setTelefono("912345678");
				customer.almacenar();
			}
		}

		System.out.println("\n======================= Se han realizado las "
				+ "modificaciones de los datos ==========================\n");
		imprimirTodosLosClientes();

		try {
```

```
                this.dbServicios.comenzandoTransaccion();
                boolean exito = this.escribirDatos();
                Throwable t = this.getQueEstaMal();
                if (t != null) {
                        throw t;
                }
                if (exito) {
                        this.dbServicios.validandoTransaccion();
                } else {
                        throw new SQLException("Modificacion de los datos con errores");
                }
                this.sincronizandoEstadoDePersistencia();
        } catch (Throwable t) {
                try {
                        this.dbServicios.deshacerTransaccion();
                } catch (SQLException sqle) {
                        sqle.printStackTrace();
                        throw new RuntimeException(sqle);
                }
                t.printStackTrace();
        } finally {
                try {
                        this.dbServicios.liberacion(true);
                } catch (SQLException sqle) {
                        sqle.printStackTrace();
                        throw new RuntimeException(sqle);
                }
        }

        System.out.println("\n======================= Paso despues "
                        + "de COMMIT y sincronizandoEstadoDePersistencia "
                        + "o despues de ROLLBACK ============================\n");
        imprimirTodosLosClientes();
}

public void buscarClientes() {
        System.out.println("\n======================= Prueba usando criterios"
                        + " ============================\n");
        setAsignaciones(cargaAsignaciones("Clientes.propiedades"));
        this.poblarMetaData();
        List manejador = new ArrayList(2);
        ManejadorCriterioBusqueda mcb1 = new ManejadorCriterioBusqueda();
        mcb1.setFieldName("apellidos");
        mcb1.setOperator(ManejadorCriterioBusqueda.LIKE);
        mcb1.setFunctions("UPPER(${})");
        mcb1.setSearchValue("MI%");
        mcb1.setBooleanCondition(ManejadorCriterioBusqueda.OR);
        manejador.add(mcb1);
        ManejadorCriterioBusqueda sch2 = new ManejadorCriterioBusqueda();
        sch2.setFieldName("apellidos");
        sch2.setOperator(ManejadorCriterioBusqueda.LIKE);
        sch2.setFunctions("UPPER(${})");
        sch2.setSearchValue("S%");
        manejador.add(sch2);
        ClientesCriterios criteria = new ClientesCriterios();
        criteria.setMetaData(this.metadatos);
        criteria.setManejadorCriterioBusqueda(manejador);
        criteria.construirCriterio();
        String select = this.sqlMap.getProperty("buscarTodosLosClientes");
        criteria.setSelect(select);
        String sql = criteria.getConsulta();
        Collection p = criteria.getArgumentos();

        try {
                this.load(this.dbServicios.getResultSet(sql, p));
        } catch (SQLException sqle) {
                throw new RuntimeException(sqle);
        }
        imprimirTodosLosClientes();
}

protected void imprimirTodosLosClientes() {
        List products = this.getObjectList();
        Iterator iter = products.iterator();
        while (iter.hasNext()) {
                Clientes clientes = (Clientes) iter.next();
                System.out.println(clientes);
```

```
                }
        }
}
```

8.6. Características suplementarias del SLGFP

Es posible transmitir solamente los objetos que sea imprescindibles, aquellos que contengan los datos de las actualizaciones, de este modo se minimiza el tráfico de red y se optimiza el rendimiento.

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

```
// Construccion de objeto
FabricaDeCliente cf = new FabricaDeCliente(Cliente.class);
// Fijamos el parámetro a TRUE
cf.setSoloSQLGenerado(true);
...
// Realizamos la operación
cf.escribirDatos();
...
// Podemos ver el resultado final en esta lista
List todosCambios = cf.getSQLGenerado();

// Otro modo podría ser
// Creamos una lista de objetos
Object[] cambios = (Object[]) todosCambios.get(0);
String sql = (String) cambios[0];
Object arg = cambios[1];
// Lanzamos la orden tal cual.
PreparedStatement ps = connection.prepareStatement(sql, arg);
...
// Si quisiéramos usar el frameworks SLGFP para las actualizaciones:
DBServicios dbServicios = new DBServicios();
...
// Se realiza el cambio en BBDD
dbServicios.ejecutar(sql, arg);
```

Otra manera de rellenar los objetos sería:

```
...
FabricaDeCliente cf = new FabricaDeCliente (Cliente.class);
ResultSet rs = ...;
cf.carga(rs);
...
```

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.

```
...
FabricaDeProductos pf = new FabricaDeProductos(Producto.class);
List arg = new ArrayList();
arg.add(new Double(77.50));
arg.add("%CARD%");
```

```
String sql = "SELECT * FROM PRODUCTOS WHERE PRECIO > ? AND NOMBRE LIKE ?";
DBServicios dbServicios = new DBServicios();
...
pf.carga(dbServicios.getResultSet(sql, arg));
dbServicios.liberacion(true);
List productos = pf.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 SLGFP concebidos (con métodos especiales para ello) 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 slgfp.ObjPersistenciaAbstracto, aunque no es la única manera.
- 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:
    - KEY_COLUMNAS
    - KEY_Y_COLUMNAS_MODIFICADAS
    - 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):

```
...
FabricaDeProductos pf = new FabricaDeProductos(Producto.class);
...
char[] sMD = new char[3];
sMD[0] = FabricaDeProductos.SECUENCIA_MODIFICACION_DATOS_DELETE;
sMD[1] = FabricaDeProductos.SECUENCIA_MODIFICACION_DATOS_UPDATE;
sMD[2] = FabricaDeProductos.SECUENCIA_MODIFICACION_DATOS_INSERT;
pf.setSecuenciaDeModificacionDeDatos(sMD);
...
```

- 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:

```
FabricaDeProductos pf = new FabricaDeProductos(Producto.class);
...
pf.carga(...
...
```

```
MarcadorPagina mp1 = this.getPagina(1);
System.out.println("\n=============== Total registros: "
                + mp1.getObjetosTotales() + ", Pagina " + mp1.getNumeroPagina()
                + " de " + mp1.getTotalPaginas() + " ===============\n");
Iterator iter = mp1.getPagina().iterator();
while (iter1.hasNext()) {
        Producto producto = (Producto) iter.next();
        System.out.println(producto);
}
...
```

- La arquitectura de SLGFP está basada en la aplicación del patrón DAO, así que si algún día se precisa cambiar a otro Frameworks el uso de SLGFP no debe ser un inconveniente, y la conversión deberá ser relativamente simple.

9. Funcionamiento del SLGFP

9.1. Presentación

Para la verificación del correcto funcionamiento del Frameworks de persistencia SLGFP 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 en el mayor grado posible la flexibilidad, manejo y sencillez de codificación.

9.2. Características

El SGBD utilizado ha sido HSQLDB http://hsqldb.org/, una sencilla BBDD, en su versión 1.8.0, que cumple los requerimientos mínimos para el ejemplo. Para la instalación según los requerimientos particulares, se remite a la documentación de la página web de referencia.

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

Se ha utilizado este SGBD en vez de otros como ORACLE o MySQL debido a la sencillez de instalación y la flexibilidad en la portabilidad de los elementos de la BBDD y el uso de estándares JDBC usuales.

El entorno de desarrollo ha sido utilizando los componentes:

- Eclipse Helios v3.6.2: utilizada para el desarrollo completo de todas las partes de la codificación y prueba. La entrega se realiza con el formato de proyecto eclipse por lo que su importación y supervisión no debe suponer mayor inconveniente para su corrección.

- MagicDraw, versiones 11.5 y 15.5, para el desarrollo de los diagramas básicos de trabajo y arquitectura.

- Rational Rose Enterprise 2003, para la verificación de loa diagramas, y la confección de algunos diagramas de prueba adicionales.

En cuanto a la instalación de las pruebas, no se requiere más que seguir las instrucciones y el uso de los lanzadores (la aplicación solo está diseñada para su uso en entornos Windows y con java 1.5 o superior, preferiblemente la 6.0, para la que se han compilado en la versión 1.6.0_24).

Los archivos .jar que se encuentran en el directorio ../prueba son los únicos necesarios para la aplicación, junto a las extensión de la carpeta ../lib.

9.3. 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 SLGFP.

El código empleado es muy sencillo, utilizando un patrón fábrica, y permite la verificación de la generación de objetos para las tablas y el uso de los comando SQL y las respuestas esperadas.

Tal y como se puede verificar, no se necesitan fichero complejos de instalación, estilo .xml, para establecer la relación entre el Framework y la BBDD. Únicamente se utilizan ficheros .properties, con las características iniciales de la BBDD y comando de uso, así como un fichero .xml, con el esquema DTD de empleo del mapeo de las SQL par la comprensión por el Framework.

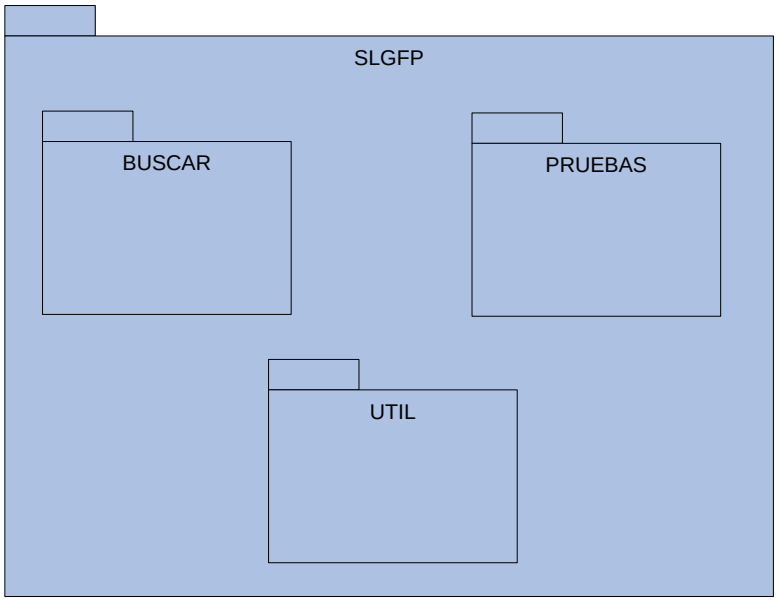


Figura 17: Esquema general de paquetes

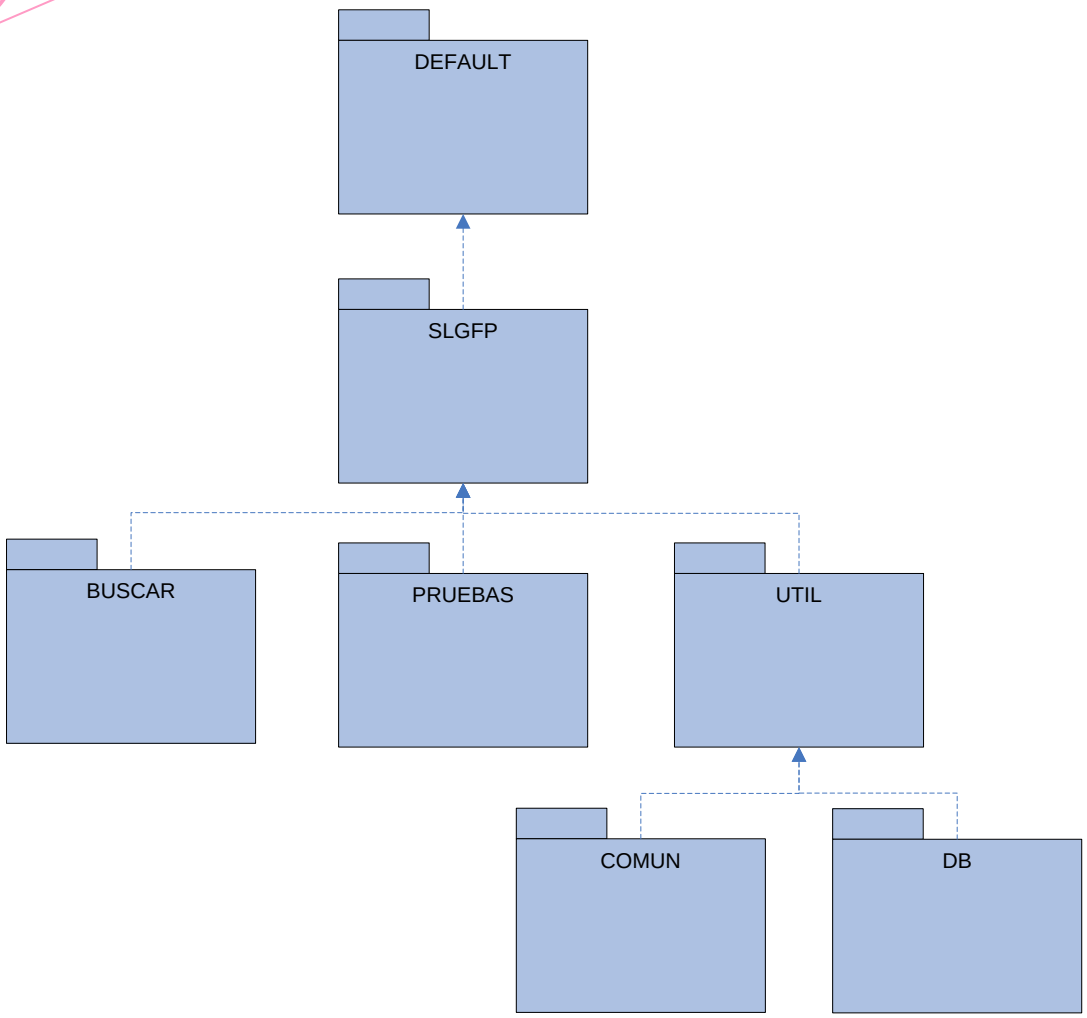


Figura 18: Diagramas de dependencias de paquetes

9.4. Diagrama de la aplicación prueba

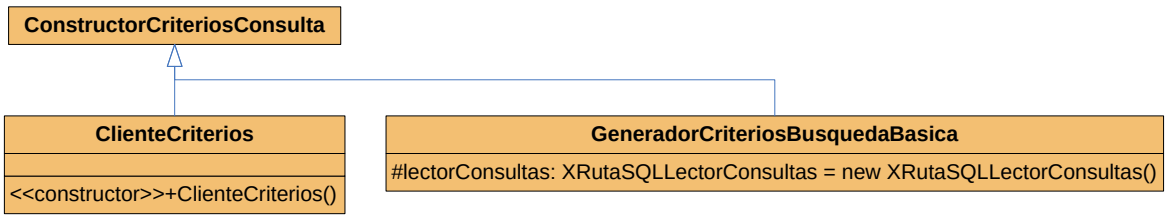


Figura 19: Jerarquía de las clases de consulta

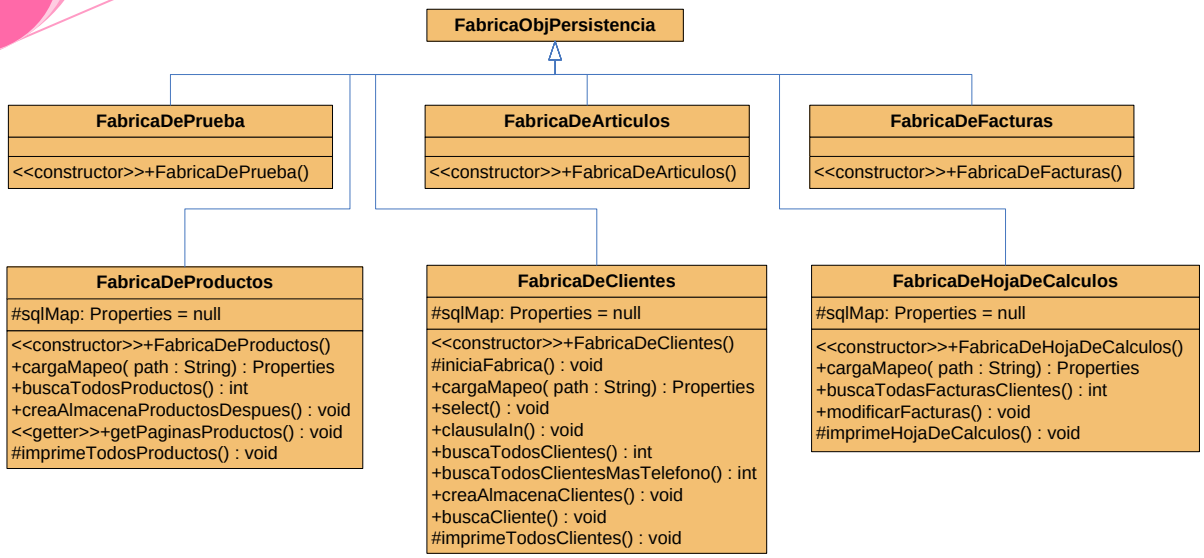


Figura 20: Jerarquía de los objetos fábrica

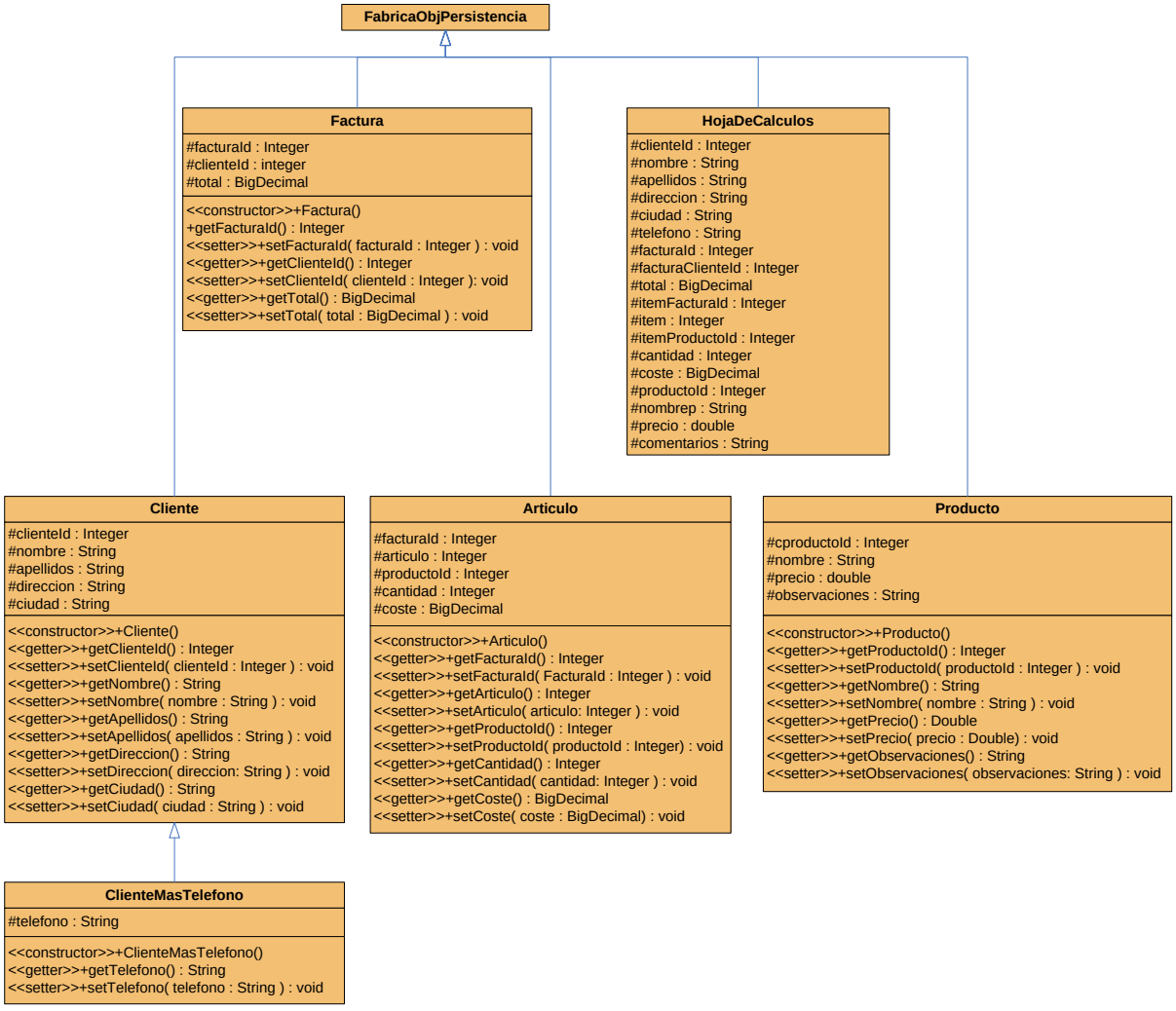


Figura 21: Jerarquía de los objetos de relación con las tablas de la base de datos

9.5. Instalación y uso de la aplicación de ejemplo

Para el uso de la aplicación la única condición es que esté instalado el kit de ejecución de Java, versión 5 o superior (preferiblemente la última ya que se ha compilado para la versión 1.6.0_24-b07, aunque siempre está la opción de recompilar el código).

La aplicación viene con una estructura de directorios tanto para su instalación en el Eclipse Helios v3.6.2 como para su ejecución.

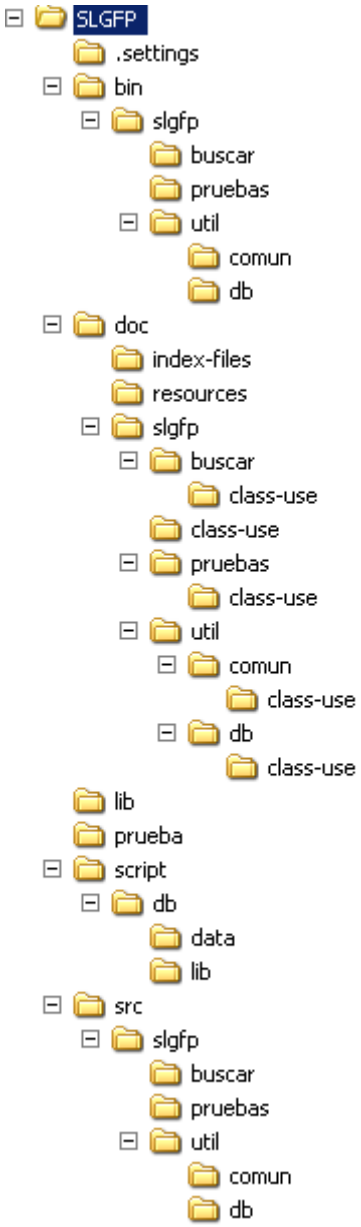


Figura 22: Estructura de directorios

La versión de prueba se ha desarrollado para entornos Win32 por lo que ha seguido el patrón de ficheros por lotes (.bat).

Para ejecutar los ejemplos seguiremos los siguientes pasos:
1. Inicio/ejecutar/CMD
2. El usuario debe colocarse en el directorio SLGFP\prueba.

3. Iniciamos la base de datos ejecutando **iniciarServidor.bat**
4. Abriremos otro CMD.
5. Nos colocaremos en el directorio SLGFP\prueba.
6. Ejecutar cada ejemplo o todos a la vez. Cada uno de los ejemplos restituye los valores de la base de datos a un estado inicial (SLGFP\script\db\configuracion.sql).
   6.1. Para ejecutar un ejemplo ejecutaremos **PRUEBA NN**, donde NN es un valor comprendido entre 01 y 11. Por ejemplo, PRUEBA 01.
   6.2. Para ejecutar todos los test de manera consecutiva ejecutaremos **PRUEBA TODOS**.

Los valores de las pruebas NN se definen en la siguiente tabla:

Valor NN	Prueba que se ejecuta
01	buscarTodosProductos
02	crearYAlmacenarProductosDespues
03	buscarTodosLosClientes
04	buscarTodosLosClientesConTelefono
05	crearYAlmacenarClientes
06	getPaginasDeProductos
07	buscarCliente
08	hojaDeCalculos
09	modificarFacturas
10	clausulaIn
11	seleccionarComo

Tabla 6: Relación de número de prueba con prueba que se ejecuta.

De manera opcional, se puede ejecutar el GUI Database Manager (runManager.bat) para visualizar los datos almacenados.

Cuando se abra el Dialogo de Conexión seleccione HSQLDB Database Engine Server, en ningún caso debe usar HSQLDB Database Engine In-Memory o cualquier otro.

Usuario: sa
Password: <no tiene>

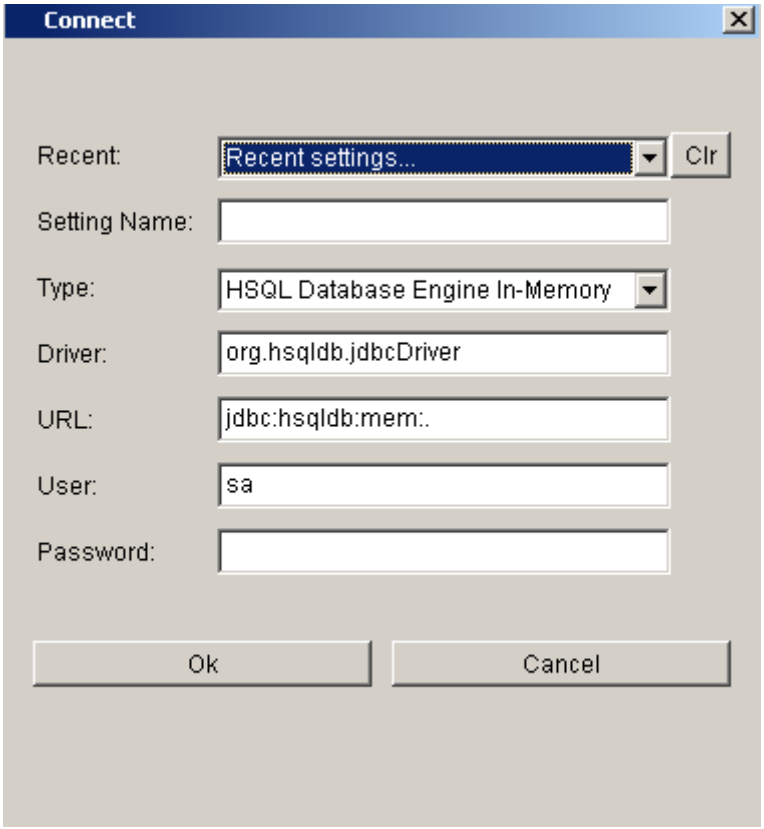


Figura 23: Administrador de la base de datos

Por último, y por expreso requerimiento de la entrega del producto se definen que librerías externas tienen que tener el proyecto y en que lugar deben colocarse para que los ejecutables diseñados puedan funcionar:

Librería	Ruta	Descripción	Descarga
Hsqldb.jar	SLGFP\SCRIPT\DB\LIB	Base de datos	
Servlet.jar	SLGFP\SCRIPT\DB\LIB	Base de datos	
Jaxen-core.jar	SLGFP\LIB	Jaxen-core	
Jaxen-jdom.jar	SLGFP\LIB	Jaxen-jdom	
Jdom.jar	SLGFP\LIB	Jdom	
Jxl.jar	SLGFP\LIB	Jxl	
Saxpath.jar	SLGFP\LIB	Saxpath	

Tabla 7: Relación de librerías con su ubicación y descripción.

10. Glosario

- **Base de datos**: son la representación integra de los conjuntos de entidades o instancias de información y sus interrelaciones.
- **Caché**: memoria que permite mejorar el rendimiento del sistema al almacenar los datos más comunes y repetidos. De acuerdo composición es muy rápida, hecho que justifica su uso.
- **Capa**: también conocida como API (Interface de Programación de Aplicaciones). Es un conjunto de convenciones de programación que definen como se debe invocar un servicio desde un programa.
- **Ciclo de vida**: conjunto de etapas de un objeto o instancia.
- **Clase**: tipo de datos definido por el usuario que especifica un conjunto de objetas que comparten las mismas propiedades.
- **Clave de identificación**: Llave que garantiza la unicidad de las instancias o diferentes elementos.
- **Consulta**: cuestión realizada a una base de datos, en la que pide información o informaciones concretas en función de unos criterios definidos.
- **Commit**: término que define la acción de confirmar la transacción actual. Todos los cambios realizados en la transacción se trasladan de forma permanente en la base de datos. El término tiene un cierto paralelismo con el propio de las bases de datos.
- **Controlador**: también conocidos como controladores, son los encargados del trabajo Conjunto entre hardware y software.
- **Correspondencia**: término que en inglés traduciríamos como “mapping”, aun cuando su traducción literaria sería cartografía. Hace referencia al hecho que se produce cuando dos elementos son el mismo en aspecto pero de diferente naturaleza.
- **Encapsulación**: una de las 3 propiedades principales de los objetos. La encapsulación se define como la propiedad de los objetos que permite el acceso a su estado únicamente a través de su interfaz o de las relaciones preestablecidas con otros objetos.
- **Encriptación**: técnica por la que la información se hace ilegible a terceras personas. Para poder acceder a la información es siempre necesaria una llave que sólo conocen emisor y receptor.
- **Especificación**: es un documento que especifica cómo debe ser un sistema, un lenguaje, una capa, etc.
- **Excepción**: término propio de los lenguajes de programación, y hace referencia a los errores que se pueden producir durante la ejecución de una aplicación.
- **Herencia de objetos**: una de las 3 propiedades principales de los objetos. Permite generar clases y objetos a partir de otros ya existentes, manteniendo algunas o todas sus propiedades.
- **Incrustado**: en inglés ‘embedded’. Viene asociado al término lenguaje incrustado. Permite diferenciar un lenguaje que se encuentra dentro de otro.
- **Instancia**: El término instancia hace referencia siempre a un objeto creado por la clase en tiempo de ejecución conservando siempre las propiedades y métodos.

- **Implementación**: hace referencia a programar una aplicación a partir de un modelo de aplicación teórico o esquemático.
- **Java**: lenguaje de programación orientado a objetos, creado por Sun Microsystems para generar aplicaciones exportables a Internet.
- **J2EE**: paquete del lenguaje Java que recoge sobre todo librerías orientadas a servidores de aplicaciones.
- **J2SE**: paquete del lenguaje Java que recoge las librerías básicas de este lenguaje.
- **Metadato**: término que define un documento que describe el contenido, la calidad, la condición y otras características un dato.
- **Modelo de datos**: término que hace referencia al sistema formal y abstracto que nos permite describir los datos a partir de una serie de reglas y convenios predeterminados.
- **Objeto**: caso concreto de una clase. Es la representación de un objeto real. Es producto de una Clase y cumple las propiedades requeridas del mismo objeto real que creemos que lo definen y que son necesarios por la nuestro aplicación. Se compone de atributos, métodos y relaciones con otros objetos.
- **Persistencia**: propiedad que hace referencia al hecho de que los datos sobre los que trata un programa no se pierdan al acabar su ejecución.
- **PersistenceManager**: Agente administrador de la persistencia de los objetos.
- **PersistenceManagerFactory**: Administrador de todos los agentes de persistencia de cualquier aplicación.
- **Rollback**: literalmente quiere decir marcha atrás. Hace referencia al término propio de base de datos que nos permite recuperar el estado anterior a una operación sobre la información en una base de datos. No se debe confundir con el mismo término propio de las bases de datos.
- **Servidores de aplicaciones**: cualquier servidor que contiene la mayor parte de la lógica de una aplicación y la manipulación de datos. No contiene ninguna información en sí, sino el tratamiento y distribución de estas.
- **SGBDR**: nomenclatura que hace referencia a las bases de datos de tipo relacional.
- **SGBDOO**: nomenclatura que hace referencia a las bases de datos orientadas a objetos.
- **SQL**: lenguaje de consulta creado por IBM que facilita el tratamiento de datos de cualquier base de datos relacional.
- **TAD**: tipo abstracto de datos. Estructura de datos que permite representar tipos de datos avanzados.
- **Transacción**: es una acción. Mueve un dato de un lugar a otro. Se da entre bases de datos, de una aplicación a una base de datos, a la inversa, o bien entre aplicaciones. No se debe confundir con el término propio de las bases de datos.
- **Transparencia**: término que hace referencia al hecho que cualquier interacción entre dos aplicaciones, si depende de una tercera, estas trabajarán como si no existiera. Esto quiere decir que no resta rendimiento ni propiedades.

- **Unicidad**: una de las 3 propiedades principales de los objetos. Permite diferenciar como entidad un objeto en sí de otro de idénticas propiedades y valores en sus atributos.
- **XML**: también conocido como Extensible Markup Language, lenguaje extensible de marcas. Es un lenguaje dirigido al etiquetado de las partes que forman un documento y es además un metalenguaje que permite definir otros lenguajes, y que facilita la comunicación entre lenguajes de diferente naturaleza.

11. Bibliografía

M.ATKINSON R.MORRISON, Orthoganally Persistent Object Systems. The VLDB. Journal 4(3) pag 319-401. 1995
KENT BECK, Extreme Programming Explained, Embrace Change, Addison-Wesley Octubre, 1999
ELIZA BERTINO, LOREZON MARTINO. Sistemas de bases de datos orientados a objetos. Addison Wesley 1993
GRADY BOOCH, JAMES RUMBAUGH, IVAR JACOSON, El lenguaje Unificicado de Modelado, Adison Wesley,1999
BROWN, K. y WHITENACK, B. "Crossing Chasms: A Pattern Language for ObjectRDBMS Integration" dentro del Pattern Languages of Program Desing Vol 2, Addison-Wesley, 1996
R.G.G. CATTELL, Object Data Management Addison Wesley, 1994
C.J. DATE, Introducción a los sistemas de bases de datos, Addison-Wesley, 1986.
BRUCE ECKEL, Thinking Java. Prentice Hall.
ERICH GAMMA, RICHARD HELM, RALPH JONSON, JOHN VLISSIDES, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison Wesley, 1995
RON JEFFIRIES, ANN ANDERSEN, CHET HENDRICKSON, Extreme Programming Installed, Addison-Wesley Pub Co; Octubre, 2000
DAVID JORDAN, CRAIG RUSSELL Java Data Objects. O 'Relly. April 2003
HENRY F.KORTH, ABRAHAM SILBERSCHATZ, Fundamentos de Bases de Datos, MgGraw Hill.
GRAIG LARMAN Uml y Patrones, Prentice Hall, 1999
BERTRAND MEYER, Object Oriented Software Construction 2nd Editon, Prentice Hall, 1997
NICHOLAS NEGROPONTE, Beeing Digital. Vintage Books. 1996
OMG Persistent State Service Specification 2.0, formal/02-09-06
R. ORFALI AND D. HARKEY, Client/Server Programming with Java and CORBA, John Wiley and Sons,
New York, 1998. Client/Server Programming with Java and CORBA, 2nd Edition, Wiley, Marzo 1998
[24] CLAUS P. PRIESE, "A flexible Type-Extensible Object-Relational Database Wrapper- Architecture"
RIEL, ARTHUR J. Object-Oriented Design Heuristics. Addison-Wesley, 1996.
ROGER S. PRESSMAN, Sotware Engineering, McGraw Hill 3ed. 1991
JAMES R RUMBAUGH, MICHAEL R. BLAHA, WILLIAM LORENSEN FREDERICK EDDY , WILLIAM
PREMERLANI, Object-Oriented Modeling and Desing, Prentice Hall.1990

SZYPERSKY, C. Component Software: Beyond Object-Oriented Programming. Addison -Wesley. Massachusetts. 1997.
SCOTT W. AMBLER Design of Robust Persistence Layer.
http://www.ambysoft.com/persistenceLayer.html
DOUGLAS BARRY, DAVID JORDAN, ODMG: The Industry Standard for Java Object Storage
http://www.objectidentity.com/images/componentstrategies.html
DOUGLAS BARRY, TORSTEN STANIENDA Solving the Java Object Storage Problem
http://csdl.computer.org/comp/mags/co/1998/11/ry033abs.htm
DAVID JORDAN, Comparison Between Java Data Objects (JDO), Serialization an JDBC for Java Persistence
http://www.jdocentral.com/pdf/DavidJordan_JDOversion_12Mar02.pdf
David Jordan, New Features in the JDO 2.0 Query Language
http://jdocentral.com/JDO_Articles_20040329.html
DAVID JORDAN, The JDO Object Model, David Jordan, Java Report, 6/2001
http://www.objectidentity.com/images/jdo_model82.pdf
ARTHUR M. KELLER y otros, Architecting Object Applications for High Performance with Relational
Databases, 1995 www.hep.net/chep95/html/papers/p59/p59.ps
WOLFGANG KELLER, Object/Relational Access Layers A Roadmap, missing Links and More Patterns
http://www.objectarchitects.de/ObjectArchitects/orpatterns/
Productos de persistencia
http://www.persistence.com/products/edgextend/index.php,
http://www.objectmatter.com/, http://www.chimu.com/products/form/index.html,
Cocobase(http://www.thoughtinc.com/cber_index.html),http://www.secant.com/Products/OI/index.html,Toplink(http://otn.oracle.com/products/ias/toplink/htdocs/sod.html) Hibernate
(http://www.hibernate.org) ObjectRelationalBridge (OJB) http://db.apache.org/ojb/

12. Páginas Web de referencia

http://64.233.183.104/custom?q=cache:T_K1lqW6tq0J:dis.um.es/~jmolina/jisbd2000.ps+crear+un+framework+persistencia+en+java&hl=en&ct=clnk&cd=3&client=pub-5127604934005033
http://ammentos.biobytes.it/
http://cc.borland.com/Item.aspx?id=22982
http://code.google.com/p/jlynx-persistence-framework/
http://code.google.com/p/persevere-framework/
http://developer.apple.com/documentation/MacOSX/Conceptual/BPFrameworks/Frameworks.html
http://diegumzone.spaces.live.com/?_c11_BlogPart_BlogPart=blogview&_c=BlogPart&partqs=amonth%3D6%26ayear%3D2006
http://dis.um.es/~jmolina/
http://en.wikipedia.org/wiki/Software_framework
http://floggy.sourceforge.net/

http://java-source.net/open-source/persistence
http://julp.sourceforge.net/
http://kuropka.net/
http://nuestro.homelinux.org/clases_gal/Objetos/frameworks/
http://pegasus.javeriana.edu.co/~fwj2ee/
http://soaagenda.com/journal/articulos/que-son-los-frameworks/
http://sourceforge.net/projects/ammentos
http://sourceforge.net/projects/gopf
http://st-www.cs.uiuc.edu/users/johnson/cs497/notes98/
http://today.java.net/pub/a/today/2007/12/18/adopting-java-persistence-framework.html
http://today.java.net/pub/a/today/2007/12/18/adopting-java-persistence-framework.html
http://today.java.net/pub/a/today/2007/12/18/adopting-java-persistence-framework.html?page=last
http://web.fi.uba.ar/~dmontal/index.html
http://wiki.typo3.org/index.php/MVC_Framework
http://wiki.typo3.org/index.php/Object_Persistence_Framework
http://www.acm.org/crossroads/espanol/xrds7-4/frameworks.html
http://www.agiledata.org/essays/implementationStrategies.html
http://www.alistapart.com/articles/frameworksfordesigners
http://www.ambysoft.com/essays/persistenceLayer.html
http://www.ambysoft.com/scottAmbler.html
http://www.assembla.com/wiki/show/kumbia/Indice_wiki_kumbia
http://www.codejava.org/detalle_nota.htm?idxnota=69479&destacada=1
http://www.codeplex.com/Proetus
http://www.codeplex.com/queryframework
http://www.codeplex.com/spf
http://www.codeproject.com/KB/architecture/WhatIsAFramework.aspx
http://www.cs.wustl.edu/~schmidt/rules.html
http://www.devx.com/Java/Article/33768/1954?pf=true
http://www.dmoz.org/Computers/Programming/Languages/Java/Enterprise_Edition/Libraries_and_Frameworks/
http://www.google.com/search?ie=UTF-8&oe=UTF-8&gfns=1&sourceid=navclient&rls=com.google:es-ES:official&q=Open-Source+Web+Framework
http://www.hibernate.org/
http://www.iscripting.net/smf/javascript/(js-framework)-create-your-own-mini-framework/0/
http://www.javaskyline.com/database.html
http://www.javaworld.com/javaworld/jw-10-2005/jw-1003-mvc.html
http://www.jdbaccess.com/
http://www.juguy.org/content/view/157/1/
http://www.karvonite.com/
http://www.mail-archive.com/asnativos@5dms.com/msg10206.html
http://www.manning.com/
http://www.martincordova.com/
http://www.matshelander.com/Weblog/DisplayLogEntry.aspx?LogEntryID=42

http://www.nabble.com/Floggy:-framework-para-persist%C3%AAncia-em-J2ME-MIDPto12305612.html
http://www.reviews.com/reviewer/quickreview/frameset_toplevel.cfm?bib_id=351360
http://www.roseindia.net/enterprise/persistenceframework.shtml
http://www.simplewebportal.net/host/1018.htm