Patrones de Diseño

Patron Facade

El patrón Facade, o Fachada, sustituye las interfaces de una serie de clases bajo una sola interfaz por lo que provee un acceso unificado a un subsistema de una aplicación.

Los sistemas se estructuran en subsistemas formados por patrones y clases que los implementan, con dependencias entre ellos. Con la utilización de este patrón se intenta evitar que un cliente, al acceder un sistema (o subsistema) necesite acceder a más de una clase provocando dependencia de cada una de ellas.

Este patrón representa una capa controladora entre los clientes y la capa de negocio, abstrae la complejidad de esta última y le presenta al cliente una interfaz sencilla reduciendo el acoplamiento y dependencia entre ellos. Encapsula la complejidad de las interacciones entre objetos exponiendo sólo las interfaces requeridas y proporcionando un acceso uniforme a los clientes.

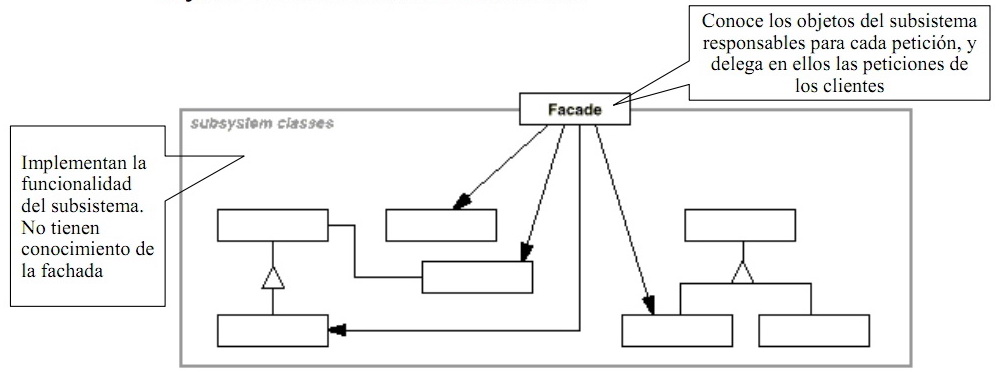


Figura x.x

Patron DAO

El patrón DAO abstrae y encapsula todos los accesos a la fuente de datos aislando los objetos de negocio de una implementación particular de una implementación particular de la persistencia. El objeto DAO maneja la conexión con la fuente de datos para obtener y modificar datos implementando los mecanismos de acceso requeridos para trabaja con la fuente de datos.

Para acceder a datos resistentes en bases de datos relacionales, el API JDBC proporciona acceso y manipulación de datos utilizando sentencias SQL. Sin embargo, incluso en estos entornos de bases de datos relacionales, la actual sintaxis y formato de las sentencias SQL pueden variar dependiendo de la base de datos en particular.

Pero las diferencias son mayores cuando se utilizan distintos mecanismos para el almacenamiento de datos. Para evitar dependencias directas de nuestro código con los mecanismos de almacenamiento de datos es especialmente útil la aplicación de este patrón de diseño.

Este patrón, al evitar estas dependencias, hace mucho menos complicada y tediosa la migración

de la aplicación de un tipo de fuente de datos a otro. Permitirá una fácil configuración para que componentes EJB, JSP o servlets trabajen con la información ubicada en sistemas B2B, LDAP, ya sean del tipo bases de datos relacionales (RDBMS), bases de datos orientadas a objeto (OODBMS), documentos XML, ficheros planos, etc. Esto podrá ser así aun cuando las API para el almacenamiento varíe según el vendedor del producto o sean API no estándares.

El uso de un objeto de acceso a dato (DAO), proporciona una solución frente a la diversidad de almacenamiento persistente abstrayendo y encapsulando todo acceso a la fuente de datos. Este objeto controla y maneja la conexión con la fuente para almacenar y obtener datos.

Este patrón de acceso a datos proporciona una capa que oculta completamente posibles cambios en la implementación de la fuente de datos, permitiéndole adoptar diferentes esquemas de almacenamiento sin afectar a los clientes o componentes de negocio. Actúa como un adaptador entre un componente y la fuente de datos, permitiendo un acceso transparente a

la fuente de datos.

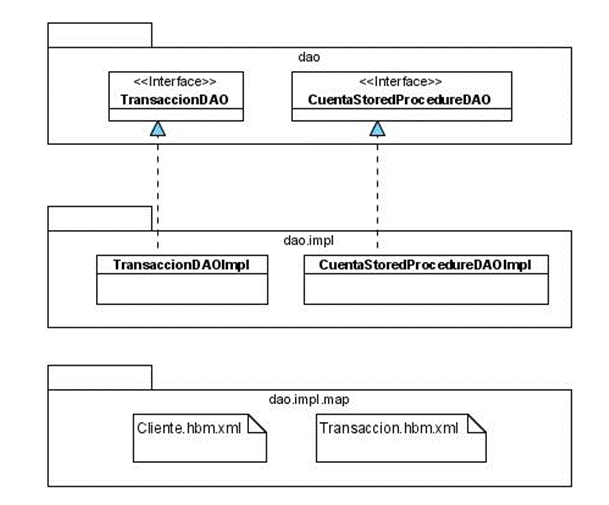


Figura x.x

Los problemas de esta implementación siguen siendo el mantenimiento de la misma así como su portabilidad. Lo único que podemos decir es que tenemos el código de transacciones encapsulado en las clases DAO.

Lo que parece claro es que debemos separar el código de nuestras clases de negocio de la realización de nuestras sentencias SQL contra la BD. Por lo tanto Hibernate es el puente entre nuestra aplicación y la BD, sus funciones van desde la ejecución de sentencias SQL a través de JDBC hasta la creación, modificación y eliminación de objetos persistentes.

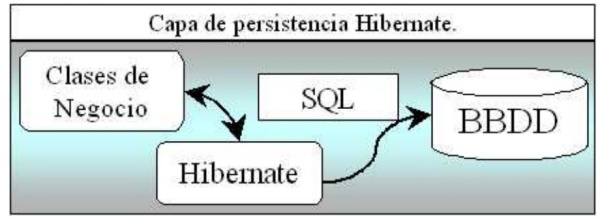


Figura x.x

Con la creación de la capa de persistencia consigue que los desarrolladores no necesiten conocer nada acerca del esquema utilizado en la BD. Tan solo conocerán el interface proporcionado por nuestro motor de persistencia. De esta manera conseguimos separar de manera clara y definida, la lógica de negocios de la aplicación con el diseño de la BD.

Esta arquitectura conllevará un proceso de desarrollo más costoso pero una vez se encuentre implementada las ventajas que conlleva merecerán la pena. Es en este punto donde entra en juego Hibernate. Como capa de persistencia desarrollada tan solo tenemos que adaptarla a nuestra arquitectura.

[Mestras,2004 ] Juan Pavón Mestras," Patrones de diseño orientado a objetos", Facultad de Informática, Universidad Complutense Madrid, España,2004.

http://www.fdi.ucm.es/profesor/jpavon/poo/2.14pdoo.pdf

Consultada: 30-10-2014

Hibernate

Hibernate es una capa de persistencia objeto/relacional y un generador de sentencias sql. Te permite diseñar objetos persistentes que podrán incluir polimorfismo, relaciones, colecciones, y un gran número de tipos de datos. De una manera muy rápida y optimizada podremos generar BBDD en cualquiera de los entornos soportados : Oracle, DB2, MySql, etc.. Y lo más importante de todo, es open source, lo que supone, entre otras cosas, que no tenemos que pagar nada por adquirirlo.

Uno de los posibles procesos de desarrollo consiste en, una vez tengamos el diseño de datos realizado, mapear este a ficheros XML siguiendo la DTD de mapeo de Hibernate. Desde estos podremos generar el código de nuestros objetos persistentes en clases Java y también crear BBDD independientemente del entorno escogido.

Hibernate se integra en cualquier tipo de aplicación justo por encima del contenedor de datos. Una posible configuración básica de hibernate es la siguiente:

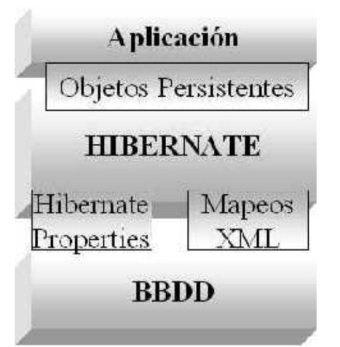


Figura x.x

Podemos observar como Hibernate utiliza la BBDD y la configuración de los datos para proporcionar servicios y objetos persistentes a la aplicación que se encuentre justo por arriba de él.

[Suárez González, 2013] Héctor Suárez González," Manual Hibernate", Documentación de javaHispano , 2013

http://www.javahispano.org/storage/contenidos/ManualHibernate.pdf

Consultada: 30-10-2014