REPASO PATRONES ARQUITECTÓNICOS

MVC y N-CAPAS

Índice

Introducción

En este tutorial sobre arquitectura del software, vamos a construir una aplicación web empleando el patrón N-Capas de Brown hibridado con el MVC.

La funcionalidad de la implementación que presentamos aquí es tan sencilla como un listado de alumnos a partir de una base de datos. Como motor de base de datos vamos a emplear Hypersonic, un gestor de bases de datos de juguete que trae integrado JBOSS para sus operaciones internas de persistencia, pero que además nosotros vamos a instanciar de modo independiente.

Recuerda lanzar la base datos que va incluída en cada proyecto que te suministramos para poder probar las aplicaciones. (hsqldbServer.bat)

Te suministramos tres proyectos acabados y funcionales correspondientes a tres modelos de desarrollo de la aplicación:

- En la primera fase desarrollaremos una aplicación basada únicamente en una página JSP que implementará la lógica de negocio, la vista y la persistencia de forma mezclada. (tew-gestioneitorv5_0.zip)
- En una segunda fase y partiendo de la versión anterior diseñaremos una arquitectura clásica de 3 capas que está fusionado con un patrón MVC. (tew-gestioneitorv5_4.zip)
- Y ya en una tercera fase desacoplaremos las capas anteriores empleando el patrón Fachada para obtener un patrón de N-Capas. (tew-gestioneitorv5_5.zip)

Aunque este tutorial está pensando para desarrollar cada proyecto a partir del anterior partiendo de la versión 5_0. Se suministran los proyectos ya acabados con el fin de que sean analizados por el alumno explicándolos en este documento.

Patrones arquitectónicos

Hay que aclarar que existe una gran similitud entre el patrón clásico de 3 capas y el MVC.
Las capas del patrón de 3-capas: persistencia, negocio y presentación se asimilan al MVC de la siguiente forma:

- Persistencia = Modelo
- Negocio = Modelo
- Presentación = Vista/Control

Veamos como implementaremos nosotros el patrón de 3-capas fusionado con el MVC:

1. Persistencia. Capa encargada del acceso a la base de datos. Habitualmente en esta capa se emplean diferentes patrones de diseño para el acceso lo más transparente posible (patrones DAO/DTO).
2. Negocio. Esta capa se encarga del cálculo de las reglas de negocio particulares del problema. En este caso el listado de los alumnos.
3. Presentación/Control. En la capa de presentación tenemos dos partes la que propiamente se ejecuta en el servidor que en nuestro caso se trata de un servlet que hace también de

controlador (MVC) y el código generado que es renderizado en el navegador (habitualmente se emplea un servlet que realiza el control, instancia un Bean y redirige el control a una JSP que renderiza la vista y accede al Bean creado en el servlet/controlador).

Verás en esta implementación un paquete extra, que es la clase tew.com.Model. Aunque este paquete no suele ser una capa por si solo, podemos decir que forma parte de la capa de negocio. Aunque también es usado (como paquete) por el resto de capas. Implementa lo que se denomina un patrón de diseño DTO (Data Transfer Object).

Modelo de 1-Capas

1. Descomprimir e importar el proyecto tew-gestioneitorv5_0.zip.
2. Dentro del proyecto tew-gestioneitorv5_0 encontraremos una carpeta **data** con la información necesaria para arrancar la base de datos. Lánzala ejecutando desde el interfaz de comandos el bat hsqldbServer.bat.
3. Ahora ya puedes desplegar el proyecto y ejecutar el JSP index.jsp.
4. Este proyecto sólo disponse de un componente software: index.jsp que incluye todas las partes de la aplicación aplicación.

Modelo de 3-Capas

1. Descomprimir e importar el proyecto tew-gestioneitorv5_4.zip.
2. Dentro del proyecto tew-gestioneitorv5_4 encontraremos una carpeta **data** con la información necesaria para arrancar la base de datos. Lánzala ejecutando desde el interfaz de comandos el bat hsqldbServer.bat.
3. Para probar la aplicación tenemos dos opciones:
    a. Lanzamos la versión indexv2.jsp que crea el Bean de scope "page" GestioneitorBean y accede a la propiedad listado genera la lista de alumnos.
    b. O bien lanzamos el servlet ListadoGestioneitorServlet que un bean "alumnos" (lista de alumnos) de scope "request" y redirecciona el control a la página index.jsp (Fíjate que si lanzas index.jsp directamente te dará un error, porque el JavaBean no existe).
4. A continuación, vamos a aplicar el patrón de separación de capas. Se trata de repartir la lógica de visualizar la lista de alumnos entre varias clases y paquetes, de tal forma que al final, una clase o JSP no implementen responsabilidades de distinta naturaleza. Seguimos los siguientes pasos:
    a. Creación de paquetes. A la hora de separar en capas, se suele utilizar un paquete distinto para cada una de ellas. Se trata de que si en un futuro alguien tiene que hacer una modificación de, por ejemplo, una sentencia SQL, sea capaz de localizarla con facilidad. Creamos, colgando de src, los paquetes:
        a. **com.tew.persistence**
        b. **com.tew.business**

c. **com.tew.presentation**

d. **com.tew.model**

b. A continuación, creamos la clase DTO com.tew.model.Alumno que contendrá las siguientes propiedades:

```
private String nombre;
private String apellidos;
private String idUser;
private String email;
```

c. Para hacer esto con eclipse, declaramos los atributos y, una vez seleccionados, **Source/Generate Getter and Setter**.

d. La clase com.tew.persistence.AlumnoDAO y le añadimos el método getAlumnos():

```
public class AlumnoDAO {

public Set<Alumno> getAlumnos() throws Exception {
   PreparedStatement ps = null;
   ResultSet rs = null;
   Connection con = null;

   Set<Alumno> alumnos = new HashSet<Alumno>();
         try {
         String SQL_DRV = "org.hsqldb.jdbcDriver";
         String SQL_URL = "jdbc:hsqldb:hsql://localhost";
         // Obtenemos la conexión a la base de datos.
         Class.forName(SQL_DRV);
         con = DriverManager.getConnection(SQL_URL, "sa", "");
         ps = con.prepareStatement("select * from alumno");
         rs = ps.executeQuery();
         while (rs.next()) {
               Alumno alumno = new Alumno();
               alumno.setNombre(rs.getString("NOMBRE"));
               alumno.setApellidos(rs.getString("APELLIDOS"));
               alumno.setEmail(rs.getString("EMAIL"));
               alumno.setIdUser(rs.getString("IDUSER"));

               alumnos.add(alumno);
         }
   }
   finally  {
         if (rs != null) {
                     try{ rs.close(); } catch (Exception ex){}
                  };
         if (ps != null) {
                     try{ ps.close(); } catch (Exception ex){}
                  };
         if (con != null) {
                     try{ con.close(); } catch (Exception ex){}
                  };
   }
```

```
    return alumnos;
}

}
```

e. Creamos la clase com.tew.business.AlumnosListado y le añadimos el método getAlumnos(). Esta clase está en la capa de lógica.

```
public class AlumnosListado {

public Set<Alumno> getAlumnos() throws Exception {
// Aquí iría lógica de negocio que ejecutase algún proceso ...
//... el ejemplo es tan sencillo que no hay nada que hacer
      AlumnoDAO dao = new AlumnoDAO();
      return  dao.getAlumnos();
      }
}
```

Hasta ahora hemos trabajado con un modelo de 3 capas.

Modelo de N-CAPAS

El paso del modelo de arquitectura 3 capas al de n-capas consiste en introducir fachadas entre las capas, con el objeto de desacoplar las clases de una capa y otra. De esta forma, cuando una clase de la capa n necesite algo de la capa n-1, simplemente deberá pedírselo a la fachada, nunca a la clase de la otra capa directamente. ¿Qué logramos con esto? El impedir que futuros cambios en una capa afecten a clases de otras capas distintas.

A partir de aquí dispondremos de dos raíces de paquete diferentes: **com.tew.* y impl.tew.***. La primera raíz contendrás las fachadas/interfaces de las diferentes capas y el segundo las implementaciones de dichas fachadas/interfaces

Para adaptar nuestra mini-aplicación al modelo de n-capas debemos:

1. Crear el interfaz **com.tew.business.AlumnosService**. Esta clase simplemente tendrá un método getAlumnos() que instancie e invoque a **AlumnosListado** para devolver el mismo conjunto que reciba. Debemos modificar la jsp de nuevo para que invoque a la fachada de la capa de negocio y no a la clase directamente, es decir, al **AlumnosService**

2. Hacer exactamente lo mismo con la capa de persistencia, creando la clase **com.tew.persistence.AlumnoDAO**

Nótese que a partir de ahora, ninguna clase deberá invocar directamente a otra clase que pertenezca a otra capa, sino que sólo conocerán a la clase que haga de fachada.

Desacoplar las implementaciones con Interfaces

Para poder cambiar implementaciones de las capas sin que estos cambios afecten a las capas superiores los más correcto es sofisticar la idea de emplear fachadas definiendo interfaces entre

las capas. De esta forma las clases que ahora son fachada de cada capa implementarán esa interfaz. Para ello:

1. Definimos la interfaz **com.tew.busines.AlumnosService** y trasladamos la actual implementación a un paquete de implementación haciendo los retoques necesarios: ahora se llamará **SimpleAlumnosService** e implementará el interfaz.
2. Aplicamos la misma separación a la capa de persistencia.

Obtener instancias con factorías

Aún hay un acoplamiento delicado, cuando una capa superior necesita una instancia de una clase fachada de la capa inferior, ésta se obtiene usando un new.

Si se necesita cambiar la implementación de la fachada de la capa (por múltiples razones), las clases de la capa superior que instancian la inferior se verán afectadas. Este acoplamiento por construcción es frecuente y la solución habitual para evitarlo es aplicar el patrón Factory. El siguiente refinamiento consiste en añadir una clase Factoría que devuelva instancias de la fachada.

1. Crearemos la interfaz **com.tew.busines.ServicesFactory** con un método createAlumnosService() cuya implementación nos devuelva un objeto instancia de SimpleAlumnosService.
2. Sustituiremos todas las llamadas a new SimpleAlumnosService() por llamadas a la factoría.
3. Aplicaremos esto a la capa de persistencia.

Desacoplar las factorías

Cada factoría tiene un acoplamiento con las clases de las que crea instancias. Por lo tanto cada implementación de la capa debe tener una factoría que la cree. Tal y como está ahora cada capa superior tiene entonces dependencia de la factoría concreta. Para hacerlo totalmente independiente deberíamos evitar ese acoplamiento. ¿Cómo? Usando interfaces de nuevo.

Cada capa ofrece sus servicios en forma de interface y además otro con los servicios de factoría que se necesita para obtener implementaciones. Así, cada implementación de capa deberá aportar la clase fachada (y todas las demás que se ocultan tras ella) y una implementación de factoría.

Queda entonces en el aire la cuestión ¿Cómo se obtiene la instancia de la factoría sin que haya acoplamiento? Sería entonces el único punto de acoplamiento entre capas, y es lo que en sistemas más sofisticados se extrae al sistema de configuración (xml, properties, etc). Esta funcionalidad por lo tanto aparecerá en la capa de infraestructura. En nuestra mini aplicación no llegaremos tan lejos y dejaremos la configuración en forma de código java con una clase como esta:

```
public class Factories {

      public static ServicesFactory services =
                                    new SimpleServicesFactory();
      public static PersistenceFactory persistence =
                                    new SimplePersistenceFactory();
}
```

Para todo ello seguiremos esta secuencia de pasos:

1. Crearemos interfaces para las factorías de cada capa: **com.tew.business.ServicesFactory** y **com.tew.persistence.PersistenceFactory**.
2. Modificaremos las implementaciones actuales de las factorías para que cada una implemente el interface adecuado.
3. Crearemos el paquete para la capa de infraestructura **com.tew.infraestrucuture** y añadiremos en él la clase Factories presentada antes.
4. Modificaremos las llamadas anteriores a las factorías para que usen la clase Factories. Por ejemplo el JavaBean de negocio GestioneitorBean:

```
public class GestioneitorBean implements java.io.Serializable{

   /**
    *
    */
   private static final long serialVersionUID = 1L;
   private AlumnosService service;


   public GestioneitorBean() {

         // Acceso a la implementacion de la capa de negocio
         // a través de la factoría
         service = Factories.services.createAlumnosService();
   };

   public List<Alumno> getListado() {
       List<Alumno> listado = null;
         try {

               listado = service.getAlumnos();

         } catch (Exception e) {
               e.printStackTrace();

         }

         return listado;
   }
}
```

Trabajo a realizar

Los objetivos de este ejercicio son:

1. Que el alumno analice, pruebe y comprenda los tres modelos que se suministran. En la clase del Martes 29 de Octubre se hará una puesta en común en clase.
2. Se deberá realizar un diagrama de clases empleando la notación que cada alumno considere más adecuada para los modelos de 3 capas y N-capas. No se exige una corrección formal del diagrama sino tener una guía para poder comprender mejor los modelos.