Producto3: Persistencia en base de datos

ICB0_P05: PROGRAMACIÓN ORIENTADA A OBJETOS CON ACCESO A BASES DE DATOS

Convocatoria febrero 2020

GRUPO ACADevelopers

Ana Iglesias

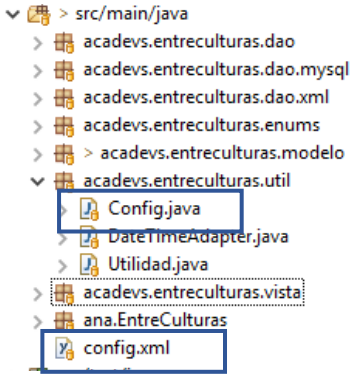
Antonio González

Consultor

Josep Vaño Chic

1. Introducción

1. En este punto del desarrollo, la aplicación se ejecuta únicamente en consola. Asimismo, hemos buscado que la interfaz sea lo más agradable y limpia posible.
   Por ello, nos hemos visto obligados a separar el código siguiendo el modelo MVC, por dos motivos:
   - Mantener un código más limpio y fácil de manejar.
   - Allanar el camino de cara al resultado final.



2. Para facilitar la configuración personal de cada uno de los miembros, teniendo en cuenta que trabajamos con bases de datos locales y por tanto independientes, hemos creado un archivo Config.xml que la aplicación lee al iniciar para cargar la configuración del equipo en el que se encuentra. (este archivo no lo compartimos por GitHub).
3. En todo momento hemos buscado aplicar un código bien indentado y de fácil lectura, nombrando variables y métodos de manera clara y explicativa.
4. Además, siempre que ha sido posible, hemos independizado las funciones de cada método, de manera que éste pueda ser reutilizado desde varios lugares de la aplicación y evitando así la duplicidad de código.
5. Las funcionalidades que en este momento del desarrollo están correctamente implementadas son Alta de socios (incluye alta de sede asociada si no existe), Actualización de socios completa, Actualización de las propiedades de la cuota de socio, Alta de Administración Física, Importación de Socios y Administraciones Físicas desde un único fichero, Listado de socios por pantalla y en formato XML.

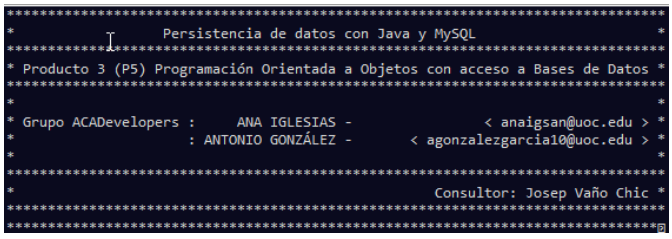


Ilustración 1 Créditos de inicio de la aplicación

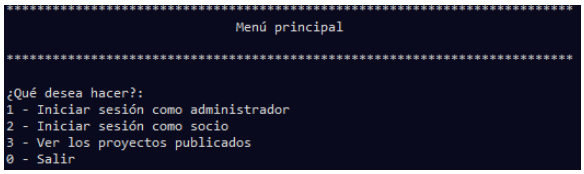


Ilustración 2 Menú principal

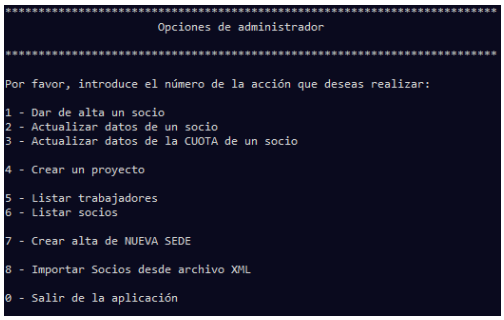


Ilustración 3 Menú de administrador

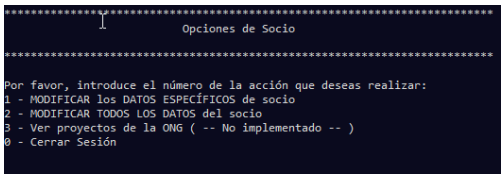


Ilustración 4 Menú de Socio

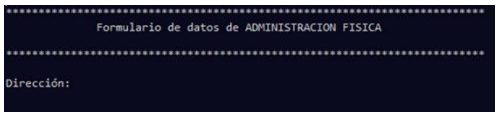


Ilustración 5 Formulario Administración

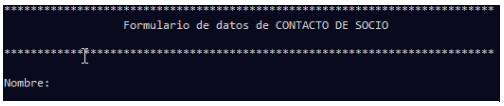
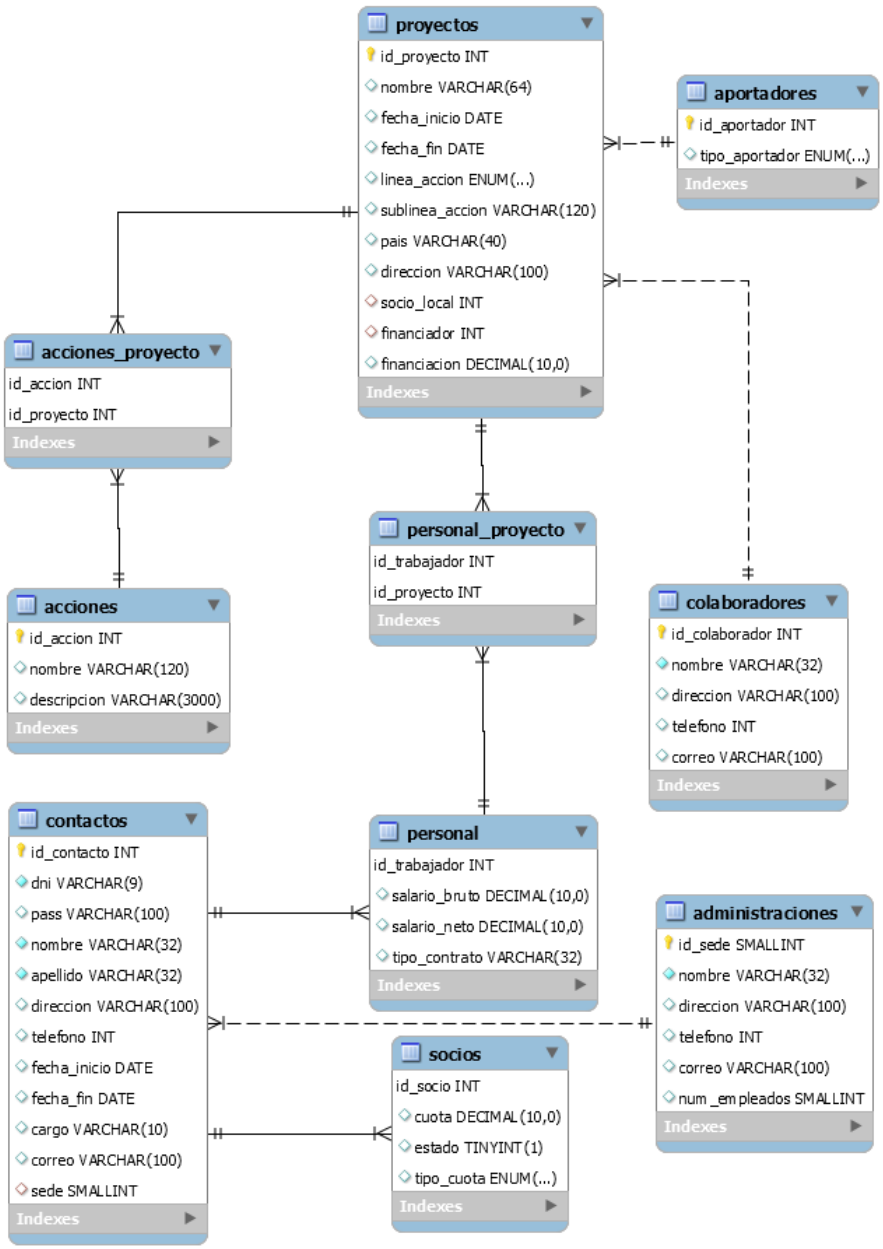


Ilustración 6 Formulario Socio Completo

– Todos los formularios comprueban la existencia de datos clave --

2. Estructura de tablas de base de datos MySQL

Para la creación de la base de datos, utilizamos los conocimientos adquiridos en el proyecto de diseño de bases de datos. Hacemos las entradas en código SQL, dando como resultado este diagrama:



Somos conscientes de que podríamos haber normalizado las tablas, mejorando la calidad de la integridad de los datos y que existen más entidades a las que podríamos aplicar persistencia, pero debido a las dificultades para realizar todos los puntos en el tiempo exigido y a que no es el objeto de este proyecto, hemos preferido evitar problemas de más.

Asimismo, consideramos que con el trabajo mostrado se puede comprobar que hemos adquirido las habilidades necesarias para realizar persistencia en MySQL de forma eficiente.

3. Campos códigos autonuméricos

A diferencia de la entrega del producto 2, modificamos la aplicación para que todos los modelos a los que les aplicamos persistencia en base de datos, cuenten con un identificador autonumérico gestionado por el SGBD MySQL.

```
// ATRIBUTOS DEL MODELO SOCIO

    private Integer id = null;
    private String pass;
    private float cuotaAportacion;
    boolean estadoAportacion = false;
    private TipoCuota tipoCuota;
```


```
//ATRIBUTOS DEL MODELO ADMINISTRACION FISICA

private Integer idAdmin = null;
private String nombre;
private String direccion;
private short numEmpleados;
private String correo;
private int telefono;
```

Nótese que el id, al ser gestionado por MySQL, no es un valor que el usuario en ningún momento pueda manipular. Por tanto, las consultas y ataques contra la base de datos y los archivos XML se realizan en base a otros identificadores candidatos a *Primary key* en la base de datos (véase El DNI de Socio, perteneciente a la clase Persona a la que especializa, o el nombre de la Administración Física)

Encontrará más información sobre la gestión y manipulación de identificadores autonuméricos en la sección de transiciones.

4. Carga de datos a partir de ficheros XML

Se accede a la carga de socios y sedes desde la opción 8 en el menú administrador.

```
public void importarSociosXML() throws DAOException {

    System.out.println("El directorio del archivo xml establecido es "+Config.rutaXML+". \n ¿Es correcto?");
    try {

        if (br.readLine().equalsIgnoreCase("n")) {
            System.out.print("Ruta del archivo de socios a importar: ");
                Config config = new Config();
                config.setRutaXML(br.readLine());
                Utilidad.grabaConfiguracion(config);
        }
    } catch (IOException e) {
        System.out.println("Error :"+e);
        e.printStackTrace();
        }

    XMLSocioDAO sociosXML = xmlF.getSocioDAO();
    AdministracionFisica sedeXml = null, sedeMySQL = null;

    List <Socio> listaSocios = sociosXML.obtenerTodos();
    int leidos = 0;
    int importados = 0;

    MySQLSocioDAO sociosMySQL = mysqlF.getSocioDAO();
    MySQLAdministracionFisicaDAO administraciones = mysqlF.getAdministracionFisicaDAO();

    for (Socio socioXml: listaSocios) {
        sedeXml = socioXml.getSedeAsignada();
        sedeMySQL = administraciones.obtener(sedeXml.getNombre());
        if (sedeMySQL == null) {
            administraciones.crearNuevo(sedeXml);
            sedeMySQL = administraciones.obtener(sedeXml.getNombre());
        }
            socioXml.setSedeAsignada(sedeMySQL);

        if (subirSocio(socioXml, sociosMySQL)) {
            importados++;
        };
        leidos++;
    }

    System.out.println("Socios leídos: "+leidos+"; Socios importados: "+importados);
}
```

Para la carga de elementos, realizamos un escaneo del archivo utilizando la API de JAXB, y creando un DOM que lee los datos de los métodos *get*…() del modelo objeto. Esta lógica está situada en la capa de persistencia DAO.XML

Además, como comentábamos en la introducción de este documento, se realiza una actualización de los datos de conexión al inicio de la aplicación.

```
public class Application {

    public MySQLDAOFactory accesoMySQL;

    public static void main(String [] args) throws ViewException {

        Utilidad.cargaConfiguracion();
        Utilidad.conectarMySQL("MySQL");
```

```
public static void cargaConfiguracion() {

    File archivoConfig = new File("config.xml");

    Config config = null;

    if (archivoLegible(archivoConfig)) {
        try {
            JAXBContext jaxbContext = JAXBContext.newInstance(Config.class);
            Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
             config = (Config) unmarshaller.unmarshal(archivoConfig);
        } catch (JAXBException e){
            System.out.println("Error al cargar los datos de config.txt . Se aplicarán los valores por defecto \n"+e.toString());
        }
    } else {
        grabaConfiguracion(config);
    }
}

public static void grabaConfiguracion(Config config) {

    File archivoConfig = new File("config.xml");
    if (config == null) {
        config = new Config();
    } else {
        if (archivoLegible(archivoConfig)) {
            try {
                JAXBContext jaxbContext = JAXBContext.newInstance(Config.class);
                Marshaller marshaller = jaxbContext.createMarshaller();
                marshaller.marshal(config, archivoConfig);
            } catch (JAXBException e) {
                System.out.println("Error al crear el archivo config.xml"+e.toString());
                }
        }
    }
}
```

Nótese que hemos separado el trabajo de carga y guardado de datos de configuración para permitir realizar estos trabajos independientemente y reutilizar ese código.

Una vez cargados los datos en modelos de java, realizamos el guardado en la base de datos mediante el Driver JDBC en la capa de persistencia *DAO.MySQL*.

5. Persistencia mediante Java y JDBC, y gestión de procedimientos almacenados.

Creamos constantes en el DAO para Socios de MySQL, para no cargar el código de las ejecuciones.

```
//SENTENCIAS MYSQL
    final String INSERT_CONTACTO = "insert into contactos "
            + "(dni,"
            + " nombre,"
            + " apellido,"
            + " direccion,"
            + " telefono,"
            + " fecha_inicio,"
            + " fecha_fin,"
            + " cargo,"
            + " correo,"
            + " pass,"
            + " sede)"
            + "values (?, ?, ?, ?, ?, ? ,? ,? ,?, ?, ?)";

    final String INSERT_SOCIO = "insert into socios "
            + "(cuota,"
            + " estado,"
            + " tipo_cuota, "
            + "id_socio)"
            + " values (?, ?, ?, ?)";
```

Realizamos las sentencias de inserción y actualización de datos de socio en dos métodos diferentes, pero en un mismo proceso.

Para ello desactivamos la propiedad de *AutoCommit* de nuestra conexión para tener control total de las ejecuciones de sentencias. Volveremos a activarla al finalizar el proceso para evitar tener que preocuparnos de manejar los *commits* en el resto de las sentencias sencillas.

```
@Override
public void crearNuevo(Socio t) throws DAOException {

    //se crea un PreparedStatement en cada método porque la intención es que se cierre al finalizarlo.
    PreparedStatement statContacto = null;
    ResultSet rs = null;

    Utilidad.activaAutoCommit(conexion, false);

    try {
        statContacto = conexion.prepareStatement(INSERT_CONTACTO, Statement.RETURN_GENERATED_KEYS);
        statContacto.setString(1,  t.getDni());
        statContacto.setString(2, t.getNombre());
        statContacto.setString(3, t.getApellidos());
        statContacto.setString(4, t.getDomicilio());
        statContacto.setInt(5, t.getTelefono());
        statContacto.setDate(6, Utilidad.conversorFecha(t.getFechaInicio()));
        statContacto.setDate(7, Utilidad.conversorFecha(t.getFechaFin()));
        statContacto.setString(8,  t.getCargo());
        statContacto.setString(9, t.getCorreo());
        statContacto.setString(10, t.getPass()); // la contraseña debería subirse encriptada.
        statContacto.setInt(11, t.getSedeAsignada().getIdAdmin());

        if (statContacto.executeUpdate() == 0) {
            throw new DAOException("Hubo algún problema al intentar la llamada insert a la tabla Contactos");
        }

        rs = statContacto.getGeneratedKeys();

        if (rs.next()) {
            t.setId(rs.getInt(1));
        } else {
            throw new DAOException("Hubo un problema al recuperar el ID del contacto en la inserción de socios");
            }

        datosEspecificosSocio(t, INSERT_SOCIO);

        conexion.commit();
    } catch (SQLException e) {
        throw new DAOException("Error al intentar guardar datos en las tablas Contactos y/o Socios", e);
    } finally {
        Utilidad.cierraRs(rs);
        Utilidad.cierraStat(statContacto);
    }

    Utilidad.activaAutoCommit(conexion, true);
}
```

Prevenimos de ataques de inyección de código SQL ocultando las variables en la sentencia *String* para insertarlos de manera segura mediante *PrepareStatement* en lugar de simples *Statements.*

Obtenemos el valor del id Autoincrementado de MySQL, mediante un *ResultSet* del propio *Statement* que nos devuelve la clave generada.

Cerramos todas las peticiones realizadas contra la base de datos.

Nótese que hemos separado las funciones de cierre de sentencias y modificación de la propiedad *Autocommit* de la conexión para disponer de un código más limpio y reutilizar ese código en el resto de los procesos.

6. Clase utilidad para controlar las conexiones

Realizamos una primera conexión al iniciar la aplicación y la cerramos al salir de ella. Para ello, los métodos que realizan estas funciones y todos aquellos que serán utilizados desde Utilidad, tienen la propiedad *Static* convirtiéndose en métodos de clase y siendo así accesibles desde cualquier parte de la aplicación.

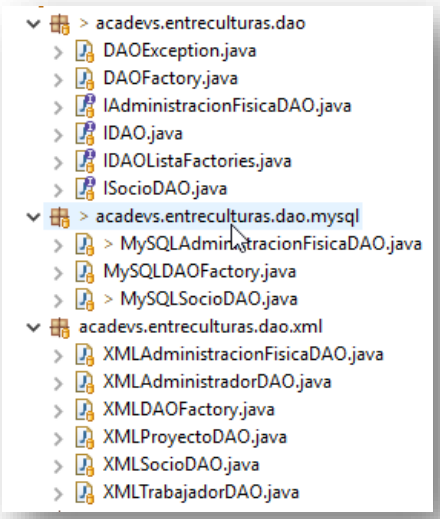
Aprovechando esta propiedad, hemos situado en la clase Utilidad todos aquellos métodos útiles y que son llamados en más de una ocasión.

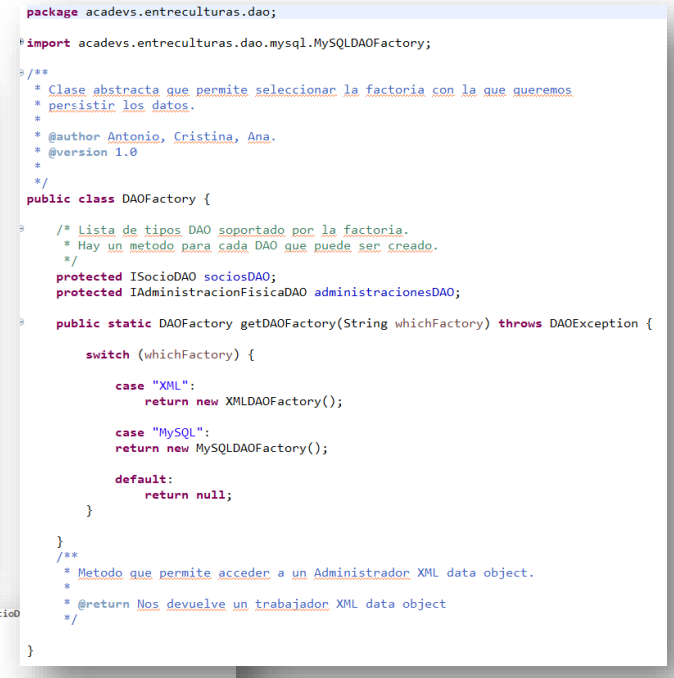
Valoramos la posibilidad de dividir esta clase en varias, dedicadas cada una a una temática concreta. (ConfigUtil, DBUtil, ScreenUtil, etc.)

7. Patrones de diseño DAO y *Factory*

Mantenemos los patrones de diseño DAO y *Factory* que implementamos en el producto 2, pero hemos separado toda la lógica de negocio de las clases DAO. De esta manera, ahora sí, las clases DAO únicamente se encargan de realizar los procesos relacionados con el acceso a datos, ya sea mediante lectura de XML o mediante acceso a BD. Los DTO o modelos, sirven como guía de los datos a los que se les va a aplicar persistencia y finalmente las clases alojadas en el paquete vista, son las encargadas de interactuar con el usuario y solicitar los procesos a las DAO.

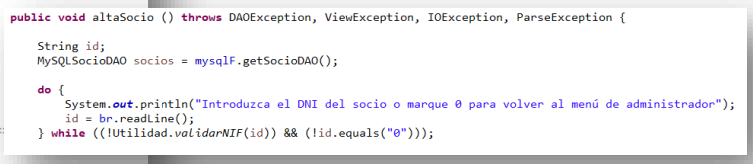
Algunas capturas que justifican el patrón *Factory:*





```
package acadevs.entreculturas.dao;

import acadevs.entreculturas.dao.mysql.MySQLDAOFactory;

/**
 * Clase abstracta que permite seleccionar la factoria con la que queremos
 * persistir los datos.
 *
 * @author Antonio, Cristina, Ana.
 * @version 1.0
 *
 */
public class DAOFactory {

    /* Lista de tipos DAO soportado por la factoria.
     * Hay un metodo para cada DAO que puede ser creado.
     */
    protected ISocioDAO sociosDAO;
    protected IAdministracionFisicaDAO administracionesDAO;

    public static DAOFactory getDAOFactory(String whichFactory) throws DAOException {

        switch (whichFactory) {

            case "XML":
                return new XMLDAOFactory();

            case "MySQL":
                return new MySQLDAOFactory();

            default:
                return null;
        }

    }
    /**
     * Metodo que permite acceder a un Administrador XML data object.
     *
     * @return Nos devuelve un trabajador XML data object
     */
}
```

```
public class MySQLDAOFactory extends DAOFactory implements IDAOListaFactories<MySQLSocioD

    public  MySQLDAOFactory () {

        this.sociosDAO = null;
        this.administracionesDAO = null;
        }

    @Override
    public MySQLSocioDAO getSocioDAO() {

if (sociosDAO == null) {
            this.sociosDAO = (ISocioDAO) new MySQLSocioDAO(Utilidad.conexion);
        }
        return (MySQLSocioDAO) sociosDAO;
    }

    @Override
    public MySQLAdministracionFisicaDAO getAdministracionFisicaDAO() {

        if (administracionesDAO == null) {
            this.administracionesDAO = (IAdministracionFisicaDAO) new MySQLAdministracionFisicaDAO(Ut
        }
        return (MySQLAdministracionFisicaDAO) administracionesDAO;
    }
}
```



```
public void altaSocio () throws DAOException, ViewException, IOException, ParseException {

    String id;
    MySQLSocioDAO socios = mysqlF.getSocioDAO();

    do {
        System.out.println("Introduzca el DNI del socio o marque 0 para volver al menú de administrador");
        id = br.readLine();
    } while ((!Utilidad.validarNIF(id)) && (!id.equals("0")));
```