UNIVERSIDAD POLITECNICA DE VALENCIA

Prácticas de Laboratorio

Práctica 4.
Persistencia. Acceso a Bases de Datos

(Guía de práctica)

Ingeniería del Software

ETS Ingeniería Informática
DSIC – UPV

Curso 2014-2015

0. Objetivo

El objetivo de esta práctica es entender los mecanismos que ofrece la tecnología JDBC para el acceso a bases de datos desde aplicaciones Java. Para ello, vamos a guiar a los equipos en el desarrollo de una pequeña aplicación cliente/servidor que permita acceder desde la capa de lógica a la de persistencia utilizando el patrón DAO (*Data Access Objects*).

La aplicación a desarrollar implementa las clases del caso de estudio "***Servicio de Llamadas de Emergencia***". Sin embargo, se muestra el modo de conseguir proveer de persistencia a la clase *Paciente* (ver Figura 1), con lo que la implementación de otras clases del caso de estudio debe realizarse de manera similar a la descrita en este documento.

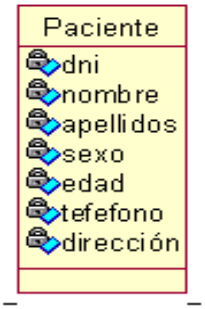

Figura 1. Diagrama de Clases (Clase Paciente)

Descripción del trabajo a realizar

Partimos de la clase *Paciente* que se muestra en la Figura 1. Vamos a implementar la persistencia de esta clase, con servicios de actualización y consulta, que serán invocados desde la capa de lógica. Para ello, vamos a seguir los pasos siguientes, estructurados en dos partes.

PARTE 1. Creación del entorno para trabajar con HSQLDB en Eclipse

1. Creación del proyecto en Eclipse
2. Diseño de la capa de lógica
3. Diseño y creación de la Base de Datos

PARTE 2. Implementación del patrón DAO y acceso a la base de datos

4. Implementación de los servicios de acceso a datos siguiendo el patrón DAO
5. Conexión de la capa de lógica y la de persistencia

Para todo ello, aplicaremos las técnicas explicadas en las sesiones de teoría de aula y seminarios, basadas en la transformación de modelos de clases en clases Java y en esquemas de BD (puntos 2 y 3), más la generación de las instancias del patrón DAO a las clases que queremos hacer persistentes (puntos 4 y 5).

PARTE 1. Creación del entorno para trabajar con HSQLDB en Eclipse

1. Creación del proyecto en Eclipse

El primer paso consiste en crear un nuevo proyecto en el entorno Eclipse, siguiendo los pasos que se vieron en las prácticas anteriores y asignarle el nombre "practica4". A continuación, crear dos paquetes llamados '*logica*' y '*persistencia*' que contendrán, respectivamente, las clases de la lógica del negocio y de acceso a datos. Nota: Crear también un paquete llamado '*excepciones*' que incluirá las clases para la gestión de excepciones en la aplicación.

2. Diseño e Implementación de la capa de Lógica

El paquete '*logica*' contendrá todas las clases de la capa de lógica de la aplicación, más algunas otras relacionadas con el patrón DAO. Las clases de la capa de lógica se derivan fundamentalmente del diagrama de clases siguiendo las reglas vistas en el tema 5 (Diseño de la Lógica de la Aplicación). De acuerdo con estas reglas, implementar en Java la clase `Paciente`. Se recomienda utilizar como tipo de los atributos `dni`, `nombre`, `apellidos` y `dirección` el tipo `String`, para el atributo `sexo` el tipo `char` y para los atributos `teléfono` y `edad` el tipo `int`.

```
// clase Paciente. Guardada en Paciente.java

package logica;

public class Paciente { . . . }
```

Además de estas clases, incluiremos en el paquete '*logica*' una clase `Aplicacion` que incluya el método `main` y las diferentes instrucciones para probar el acceso a datos. Posteriormente la completaremos cuando tengamos la implementación de los servicios de acceso a datos.

```
//clase Aplicacion. Guardada en Aplicacion.java

package logica;

public class Aplicacion {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
	}
}
```

3. Diseño y creación de la base de datos

El diseño lógico relacional de la base de datos se obtendrá también a partir del diagrama de clases de la Figura 1. En este caso quedará de la siguiente manera (* Las pautas para el diseño lógico-relacional se verán con detalle en la asignatura de BDA*):

- Se creará una tabla "PACIENTE", con columnas "DNI" (CP), "NOMBRE" , "APELLIDOS", "DIRECCION", "TELEFONO", "EDAD" y "SEXO"

La creación se compone de dos pasos diferenciados: definición del esquema y creación. Como base de datos trabajaremos con **HSQLDB**, un completo gestor de bases de datos relacionales 100% puro Java y de código abierto. En el anexo "Persistencia y Acceso a Datos. Herramientas" se resumen sus principales características y funcionalidad. Instalaros la base de datos en vuestro directorio de trabajo (simplemente descomprimir el .zip en el directorio escogido). Suponiendo que esté en "c:\hsqldb", hemos de incluir el hsqldb.jar como librería de nuestro proyecto en Eclipse (se encuentra en el directorio \lib).

1. Creamos una carpeta denominada *lib* en nuestro proyecto mediante New/Folder.
2. Copiar la librería hsqldb.jar a la carpeta *lib*.
3. Propiedades del proyecto/Java Build Path/Libraries/Add JARs (ver Figura 2).

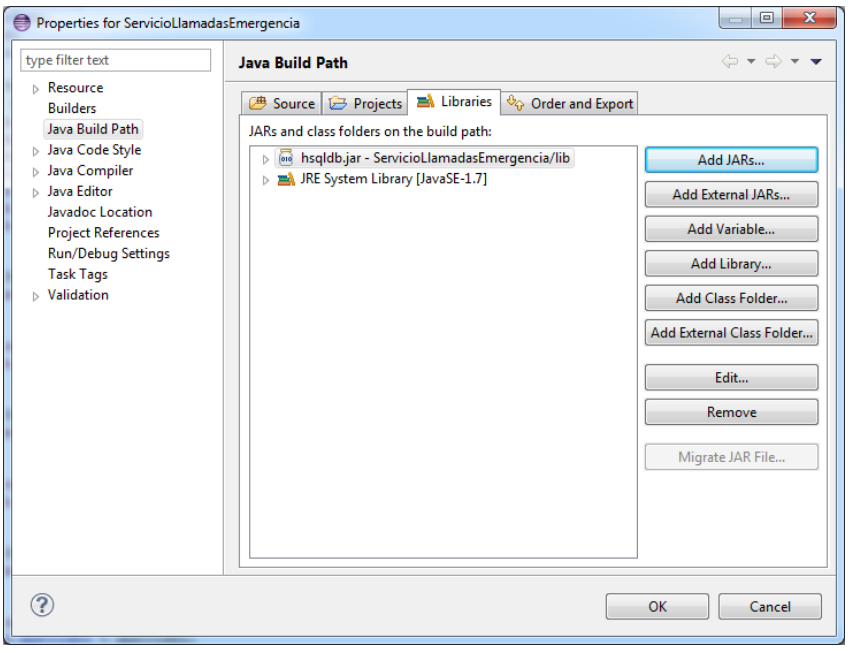

Figura 2. Propiedades del proyecto – Incluir hsqldb.jar como librería

3.1. Definición del esquema

Para definir el esquema de base de datos utilizaremos el plug-in para Eclipse "**Clay**", que permite representar gráficamente el modelo relacional de una base de datos y generar el script SQL para su creación.

Pasos para la definición del esquema:

1. Hacer click con el botón derecho sobre el paquete 'persistencia' y seleccionar "**New->Other...->Database Modelling->Clay Database Diagram**". Aparecerá la ventana para poder crear el diseño de la base de datos, a la que denominaremos "emergenciasBD" (ver Figura 3). Nota: comprobar que como *SQL Dialect* aparece HSQLDB 1.7.0/1.7.1

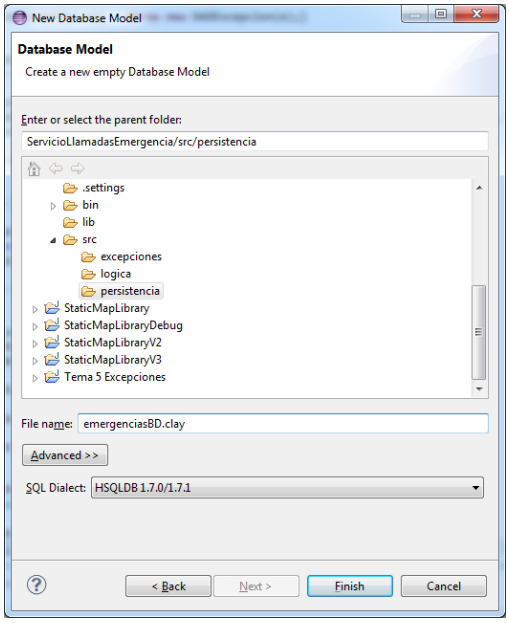

Figura 3. Creando un nuevo esquema

2. Para crear una nueva tabla, basta con hacer click en el icono correspondiente de la barra de herramientas Clay. Una vez seleccionada, al hacer click de nuevo en cualquier zona del área de diseño aparece una tabla de nombre TABLE_1. Para editar la tabla, hacer doble click sobre la misma, con lo que se abre un diálogo mediante el que se puede cambiar sus propiedades, añadir columnas, etc. La Figura 4 muestra la tabla "PACIENTE" una vez finalizada su descripción.

3. Finalmente, si hubiera que definir el enlace mediante clave ajena entre dos tablas, se haría haciendo click sobre el icono "**Foreign Key Reference**" de la barra de herramientas, y se uniría ambas tablas.

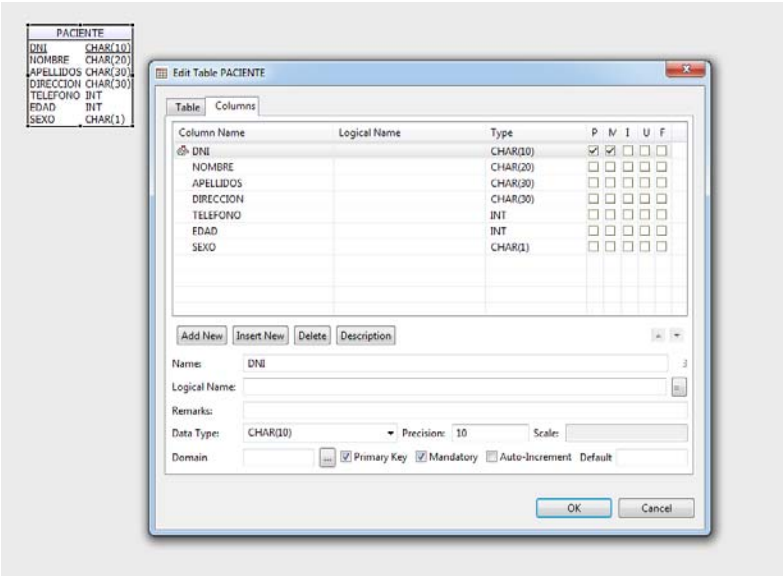

Figura 4. La tabla PACIENTE ha sido creada

4. **Una vez definido el modelo, hay que generar el script SQL** de creación de la base de datos que se utilizará posteriormente (descrito en sección 3.2). Para ello, y tras guardar el modelo, hemos de seleccionar la opción de menú principal "**Clay->Generate SQL (CREATE TABLE) Script...**" y almacenarlo en la carpeta "persistencia". La Figura 5 muestra los parámetros de la creación (// Atención, no olvidéis seleccionar la opción "Never Use Schema Names"). Al pulsar el botón "**Finish**" se abre una ventana con título "emergenciasBD.sql" que contiene el script generado. Nota: Puede avisarnos de que no hay ninguna conexión creada, será el siguiente paso.

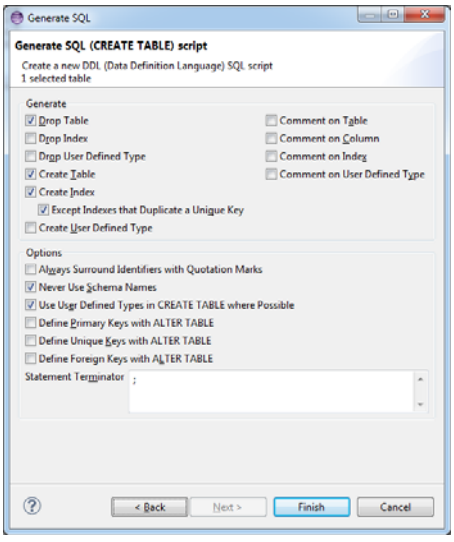

Figura 5. Generando el script de creación de la base de datos

3.2. Creación de la base de datos

Una vez generado el script, estamos en condiciones de crear nuestra base de datos y definir la conexión con nuestra aplicación mediante el drive JDBC. Para ello, haremos uso de SQL Explorer, seleccionando la perspectiva SQL Explorer. Los pasos a seguir son relacionados a continuación:

1. Seleccionar el driver para la conexión entre el SQL Explorer y la base de datos HSQLDB. Para ello, en **Window -> Preferences -> SQL Explorer -> JDBC drivers** marque 'HSQLDB Server'; haga click en el botón 'Set Default', y luego en 'Edit'. (ver Figura 6)

2. En la ventana de 'Change Driver'; en la pestaña 'Extra Class Path' haga click en 'Add Jars' y añada el *jar* de la base de datos (hsqldb.jar). Pulsar 'List Drivers' y después pulsar OK. (ver Figura 6)

3. Crear la conexión a la base de datos.

Importante: para que las conexiones funcionen adecuadamente, el servidor HSQLDB debe estar en funcionamiento. Para ello, la base de datos HSQLDB debe estar instalada. Suponiendo que esté en "c:\hsqldb" editar el archivo **c:\hsqldb\bin\runServer.bat** para que contenga lo siguiente:

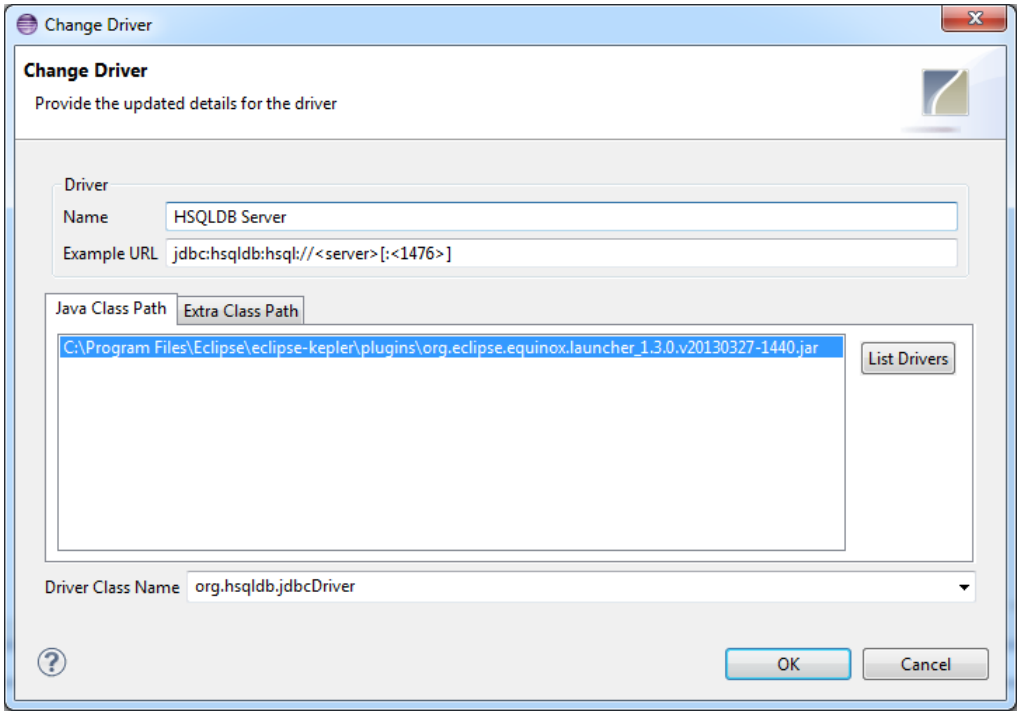

Figura 6. Selección del driver

cd ..\data
@java -classpath ../lib/hsqldb.jar org.hsqldb.Server -database.0 file:%1 -dbname.0 %2

Para arrancar el servidor, debe abrirse una ventana del intérprete de comandos, y en ella teclear lo siguiente:

```
C:\...>cd \hsqldb\bin
C:\hsqldb\bin>runserver emergenciasBD emergenciasBD
```

El servidor de base de datos arranca mostrando la siguiente información en la ventana. **No cierre la ventana durante el uso de la aplicación, para que el servidor esté en funcionamiento.**

```
C:\hsqldb\bin>cd ..\data
[Server@3e2f1b1a]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@3e2f1b1a]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@3e2f1b1a]: Startup sequence initiated from main() method
[Server@3e2f1b1a]: Loaded properties from
[C:\temp\hsqldb\data\server.properties]
[Server@3e2f1b1a]: Initiating startup sequence...
[Server@3e2f1b1a]: Server socket opened successfully in 14 ms.
[Server@3e2f1b1a]: Database [index=0, id=0, db=file:emergenciasBD,
alias=emergenciasbd] opened sucessfully in 115 ms.
[Server@3e2f1b1a]: Startup sequence completed in 130 ms.
[Server@3e2f1b1a]: 2014-10-30 12:35:08.522 HSQLDB server 1.8.0 is
online
[Server@3e2f1b1a]: To close normally, connect and execute SHUTDOWN SQL
[Server@3e2f1b1a]: From command line, use [Ctrl]+[C] to abort abruptly
```

Una vez arrancada la base de datos, se puede crear la conexión. Para ello, en la vista SQL Explorer hacer click en el botón 'New Connection Profile' de la pestaña 'Connections' (parte superior izquierda de la pantalla. Se abrirá el diálogo correspondiente, tal y como muestra la Figura 7. Los valores a introducir son los siguientes:

- Nombre de la conexión: el que deseen (en la figura, 'emergenciasBD)
- Driver: 'HSQLDB Server'
- URL: jdbc:hsqldb:hsql://localhost/emergenciasBD (nombre de la base de datos)
- Marcar la casilla "Username is not required for..."

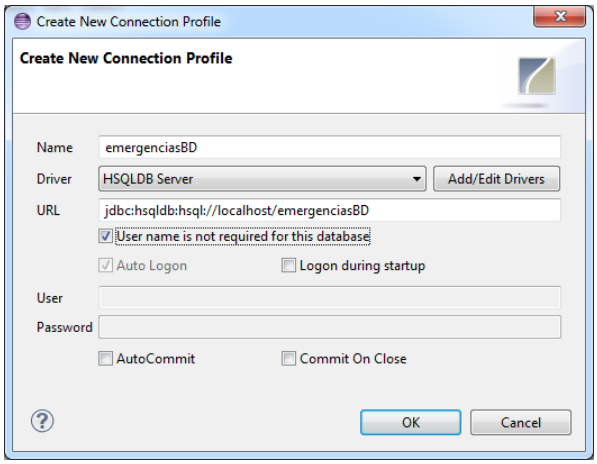

Figura 7. Creación de la conexión a la base de datos

Pulsar 'OK' y se creará la conexión. Tras ello, la vista SQL Explorer será la de la Figura 8. A la izquierda aparece la conexión creada, a la derecha la base de datos (todavía sin tablas), y en el centro las ventanas de los scripts SQL, con una ventana vacía de nombre SQL Editor (puede incluir algunos caracteres adicionales).

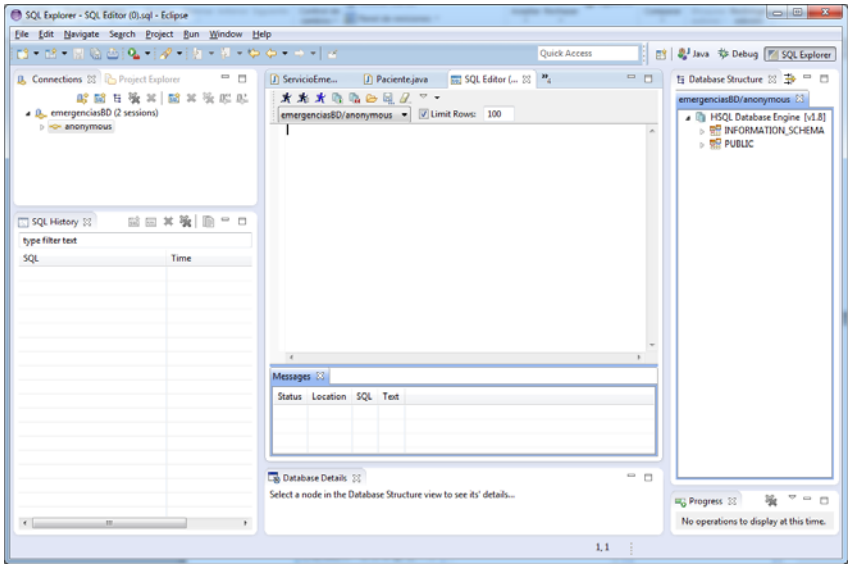

Figura 8. Base de datos creada (vacía)

Una vez el texto del script emergenciaBD.sql aparezca en la ventana correspondiente (bien refrescando la conexión establecida o copiándolo), a continuación haga click sobre el botón "**Execute SQL**". El script SQL se ejecutará, y podrá ver cómo en la **vista** "**Database Structure**", en la parte derecha, las tablas han sido creadas (puede necesitar un refresco con botón derecho sobre Tables->Refresh). La figura 9 muestra el resultado final.

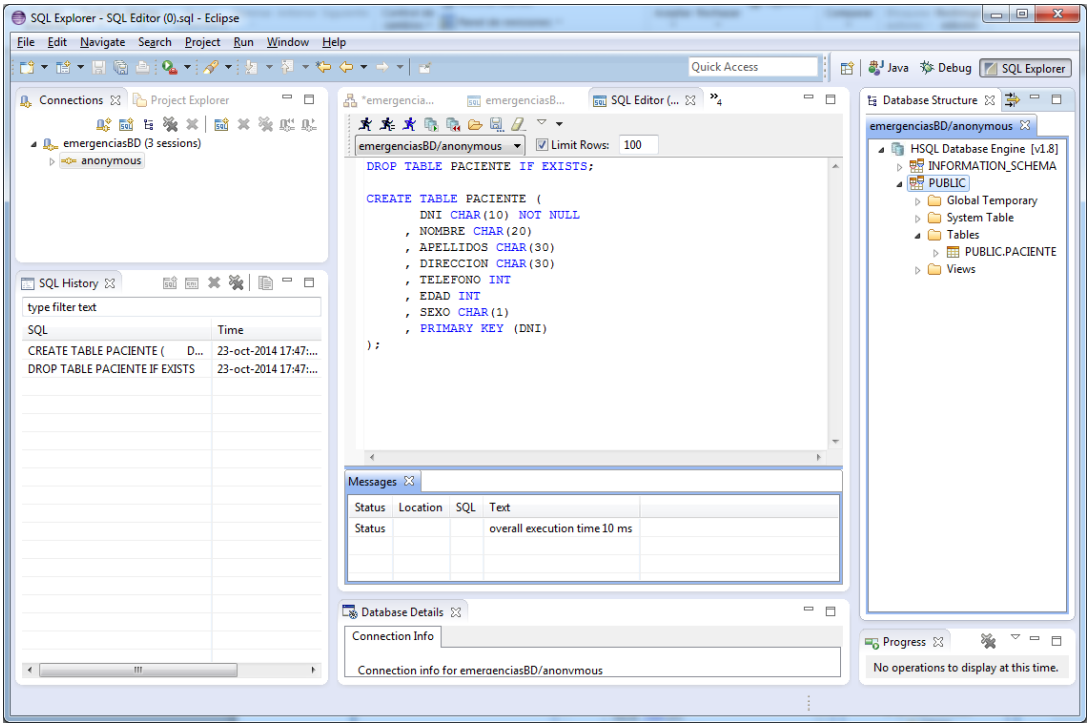

Figura 9. Base de datos con las tablas creadas

PARTE 2. Implementación del patrón DAO y acceso a la base de datos

4. Implementación de los servicios de acceso a datos siguiendo el patrón DAO

Vamos a emplear una versión simplificada del patrón DAO para implementar los servicios de la capa de persistencia. Como se explicó en clase (Tema 6. Diseño de la Persistencia), vamos a realizar una instanciación del mismo para cada una de las clases que queremos hacer persistentes (en este caso, **`Paciente`**). La aplicación debe permitir la creación de pacientes, y la localización de los mismos vía su identificador; además, deberá permitir obtener una lista de los pacientes que existen en la base de datos.

La instanciación se realiza como sigue:

a) `Paciente`

- Data Access Object: definiremos la interfaz `IPacienteDAO` y su implementación en la clase `PacienteDAOImp` (ambas en el paquete 'persistencia'):
- Data Source: un objeto de la clase `ConnectionManager`, que manejará las conexiones a la base de datos
- Un objeto de la clase `Paciente` lo tomaremos como argumento de los métodos de actualización y como resultado de los métodos de consulta.

La implementación de `IPacienteDAO` y `PacienteDAOImp` se encuentra en la unidad de disco *fileserver* del laboratorio (…/isw/practica4). Importar los ficheros a vuestro proyecto y explorar el código proporcionado.

```
//IPacienteDAO
package persistencia;
import java.util.List;

import excepciones.*;
import logica.Paciente;;

public interface IPacienteDAO {
	public Paciente buscarPaciente(String dni)throws DAOExcepcion;
	public void crearPaciente (Paciente p)throws DAOExcepcion;
	public List <Paciente> listarPacientes() throws DAOExcepcion;
}

//PacienteDAOImp
package persistencia;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
```

```
import logica.Paciente;
import excepciones.DAOExcepcion;

public class PacienteDAOImp implements IPacienteDAO {
	protected ConnectionManager connManager;

	public PacienteDAOImp() throws DAOExcepcion {
		super();
		try{
			connManager= new ConnectionManager("emergenciasBD");
		}
		catch (ClassNotFoundException e){
			throw new DAOExcepcion(e);}
	}

	public void crearPaciente(Paciente pa) throws DAOExcepcion {
		try{
			connManager.connect();
			connManager.updateDB("insert into PACIENTE (DNI,
			  NOMBRE, APELLIDOS, DIRECCION, TELEFONO, EDAD,
			  SEXO)
			  values('"+pa.getDni()+"','"+pa.getNombre()+"','"+
			  pa.getApellidos()+"', '"+pa.getDireccion()+
			  "', "+pa.getTelefono()+","+pa.getEdad()+",'"+
			  pa.getSexo()+"')");
			connManager.close();
		}
		catch (Exception e){    throw new DAOExcepcion(e);}
	}

	public List <Paciente> listarPacientes() throws DAOExcepcion{
		try{
			connManager.connect();
			ResultSet rs=connManager.queryDB("select * from
			  PACIENTE");
			connManager.close();

			List<Paciente> listaPacientes = new
			  ArrayList<Paciente>();

			try{
				while (rs.next()){
					Paciente pa =
					  buscarPaciente(rs.getString("DNI"));
					listaPacientes.add(pa);
				}
				return listaPacientes;
			}
			catch (Exception e){    throw new DAOExcepcion(e);}
		}
		catch (DAOExcepcion e){      throw e;}
	 }

	public Paciente buscarPaciente(String dni)throws DAOExcepcion{
		try{
			connManager.connect();
			ResultSet rs=connManager.queryDB("select * from
			  PACIENTE where DNI= '"+dni+"'");
			connManager.close();

			if (rs.next())
```

```
                        return new Paciente(dni,
                          rs.getString("NOMBRE"),
                          rs.getString("APELLIDOS"),
                          rs.getString("DIRECCION"),
                          rs.getInt("TELEFONO"),
                          rs.getInt("EDAD"),
                          rs.getString("SEXO").charAt(0));
                  else
                        return null;
            }
            catch (SQLException e){ throw new DAOExcepcion(e);}
      }
}
```

Nota!: Para facilitar la conexión a la base de datos a través del driver JDBC, se ha implementado la **clase `ConnectionManager`**, cuyo código se adjunta (también disponible en la unidad de disco *fileserver* del laboratorio. Proporciona los métodos `connect()`, `close()`, `updateDB(String sql)` y `queryDB(String sql)`.

```
package persistencia;
import java.sql.*;


public class ConnectionManager {
      private String sourceURL;
      private Connection dbcon=null;


public ConnectionManager(String dbname) throws ClassNotFoundException{
       Class.forName("org.hsqldb.jdbcDriver");
       sourceURL = "jdbc:hsqldb:hsql://localhost/"+dbname;
}
public void connect() throws SQLException{
      if (dbcon==null)
            dbcon = DriverManager.getConnection(sourceURL);
}
public void close() throws SQLException{
      if (dbcon!=null){
            dbcon.close();
            dbcon=null;
      }
}
public void updateDB(String sql) throws SQLException{
      if (dbcon!=null){
            Statement sentencia = dbcon.createStatement();
            sentencia.executeUpdate(sql);
      }
      }
public ResultSet queryDB(String sql) throws SQLException{
      if (dbcon!=null){
            Statement sentencia = dbcon.createStatement();
            return sentencia.executeQuery(sql);
      }
      return null;
}
}
```

Nota 2: También se puede ver en el código proporcionado la gestión de excepciones `SQLException` y `DAOExcepcion`. La implementación de `DAOExcepcion` se encuentra en la unidad de disco *fileserver* del laboratorio (…/isw/practica4). Importar los ficheros para la gestión de excepciones a vuestro proyecto y explorar el código proporcionado.

5. Conexión de la capa Lógica con la capa de Persistencia

Para ilustrar la conexión entre ambas capas, vamos a crear objetos, almacenándolos en la base de datos, y luego recuperar algunos de ello. El código que se muestra a continuación ilustra el uso de los objetos DAO con algunos ejemplos. Por simplicidad, se asume que la clase *SistemaEmergencia* tiene una colección de Pacientes (en lugar de una colección de *RegistroEmergencia*), y disponemos de una clase *Aplicación* que utiliza la clase *SistemaEmergencia*, como puede verse en la Figura 10.

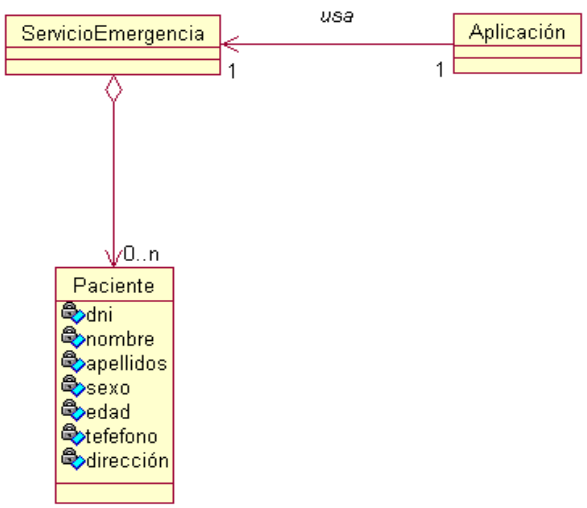

Figura 10. Ejemplo para conectar la base de datos (tabla PACIENTE)

Paso 1. Se crea la clase *ServicioEmergencia* en el paquete '*logica*'. Esta clase tiene una colección de pacientes (implementado con una *HashMap*) y tres métodos: `añadirPaciente()`, `buscarPaciente()` y `listarPacientes()`. El primer método añade, si es necesario, un nuevo paciente a la colección de pacientes y a la base de datos. El segundo busca un paciente: primero se busca en la colección de pacientes y, si no se encuentra, lo busca en la base de datos. El tercero recupera todos los pacientes de la base de datos, porque no existe ninguna garantía de tener todos los pacientes almacenados en memoria, y actualiza la colección. El código de estos métodos se muestra a continuación.

```java
package logica;

import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;

import excepciones.DAOExcepcion;
import persistencia.IPacienteDAO;
import persistencia.PacienteDAOImp;

public class ServicioEmergencia {
   private HashMap<String, Paciente> listaPacientes;

   //Objeto para la comunicación con persistencia
   private IPacienteDAO pacienteDAO;


   public ServicioEmergencia() throws DAOExcepcion {
       this.listaPacientes = new HashMap<String, Paciente>();
       this.pacienteDAO= new PacienteDAOImp();
   }


   public void añadirPaciente(Paciente paciente) throws DAOExcepcion
   {
       //Buscamos el paciente
       if (buscarPaciente(paciente.getDni())==null)
       {
          //Si no lo tenemos lo añadimos
          this.listaPacientes.put(paciente.getDni(), paciente);
          pacienteDAO.crearPaciente(paciente);
       }
   }



   public Paciente buscarPaciente(String dni) throws DAOExcepcion
   {
       //Busco el paciente en memoria
       Paciente paciente = this.listaPacientes.get(dni);

       //El paciente no esta en memoria, se busca en la BD
       if (paciente==null)
       {
          paciente = pacienteDAO.buscarPaciente(dni);

          //Si lo he encontrado en la BD, lo añado a memoria porque no lo
          //tenía
          if (paciente != null) this.listaPacientes.put(dni, paciente);
       }

       //Devuelvo el paciente que he encontrado o null en caso contrario
       return paciente;
   }
```

```
   public List<Paciente> ListarPacientes() throws DAOExcepcion
   {
       //Se obtienen los pacientes de la BD, porque no hay ninguna garantía
       //de tenerlos todos en memoria
       List<Paciente> lista = pacienteDAO.listarPacientes();

       //Para cada paciente de la BD, si no esta en memoria lo añadimos
       for(Paciente p:lista)
          if (this.listaPacientes.containsKey(p.getDni()))
              this.listaPacientes.put(p.getDni(), p);
       return new ArrayList<Paciente>(listaPacientes.values());
   }

}
```

Paso 2. Ahora se crea la clase Aplicación en el paquete "logica". Esta clase contiene el método `main` del proyecto que nos permite implementar el código para comprobar la capa de Acceso a Datos y trabajar con distintos pacientes.

```
public static void main(String[] args) {
    try{
        //Se crea el Servicio de Emergencias
        ServicioEmergencia emergencias = new ServicioEmergencia();

        // AQUÍ INCLUIR CADA UNA DE LAS EJECUCIONES

    }catch (DAOExcepcion e){

        System.out.print("DAOExcepcion: "+e);
    }

 }
```

Se presentan tres trozos de código que representan tres ejecuciones distintas de la aplicación, se puede comprobar la persistencia entre las distintas ejecuciones para saber si se ha realizado correctamente todo el proceso.

```
// Primera ejecución
//Se busca un paciente y se crea si no existe
if (emergencias.buscarPaciente("10123456A")==null)
   emergencias.crearPaciente(new Paciente("10123456A", "Juan",
      "Martinez Gandia", "Calle Santiago, 4 Valencia", 123453250,
      50, 'H'));

//Se busca un paciente y si lo encuentra se muestran sus datos
Paciente p = emergencias.buscarPaciente("10123456A");
if  (p!=null)
   System.out.println(" DNI: "+p.getDni()+" Nombre: "+p.getNombre()+
      " Apellidos: "+p.getApellidos()+
      " Dirección: "+p.getDireccion()+" Teléf.: "+p.getTelefono()+
      " Edad: "+p.getEdad()+" Sexo: "+p.getSexo());
```

```java
//Segunda ejecución
//Se busca un paciente y si lo encuentra se muestran sus datos
Paciente p1 = emergencias.buscarPaciente("10123456A");
if  (p1!=null)
   System.out.println(" DNI: "+p1.getDni()+
      " Nombre: "+p1.getNombre()+" Apellidos: "+p1.getApellidos()+
      " Dirección: "+p1.getDireccion()+" Teléf.: "+p1.getTelefono()+
      " Edad: "+p1.getEdad()+" Sexo: "+p1.getSexo());

//Se crea un paciente
emergencias.crearPaciente(new Paciente("10123457A", "Pedro",
   "Suecaz Santos", "Calle San Vicente, 4 Valencia", 123453251, 25,
    'H'));

//Se listan todos los pacientes
List<Paciente> listaPacientes = emergencias.ListarPacientes();
for(Paciente pac:listaPacientes)
   System.out.println(" DNI: "+pac.getDni()+
      " Nombre: "+pac.getNombre()+" Apellidos: "+pac.getApellidos()+
      " Dirección: "+pac.getDireccion()+" Teléf.: "+pac.getTelefono()+
      " Edad: "+pac.getEdad()+" Sexo: "+pac.getSexo());
```

```java
//Tercera ejecución
//Se crea un paciente
emergencias.crearPaciente(new Paciente("10123458A", "Ana",
   "Bezo Tosa", "Calle Francia, 4 Valencia", 923453251, 34, 'M'));

//Se listan todos los pacientes
List<Paciente> listaPac = emergencias.ListarPacientes();
for(Paciente pac:listaPac)
   System.out.println(" DNI: "+pac.getDni()+
      " Nombre: "+pac.getNombre()+" Apellidos: "+pac.getApellidos()+
      " Dirección: "+pac.getDireccion()+" Teléf.: "+pac.getTelefono()+
      " Edad: "+pac.getEdad()+" Sexo: "+pac.getSexo());
```

Extensión del proyecto 'Práctica4'

Una vez finalizado el proyecto 'Práctica4' tal como se explica en los puntos anteriores, se pide una extensión del mismo para incorporar toda la persistencia de la aplicación. En concreto, se extiende el diagrama de clases de partida, con un atributo "idRegistro" en "RegistroEmergencia" (ver figura 11). A partir de este diagrama y siguiendo de nuevo los pasos:

1. Se debe trabajar sobre el proyecto en Eclipse que se realizó para la práctica 3
2. Rediseño de la capa de lógica de dicho proyecto
3. Diseño de la Base de Datos de acuerdo con la figura 11, teniendo en cuenta que la clase "SistemaEmergencia" no es persistente
4. Implementación de los servicios de acceso a datos siguiendo el patrón DAO. La capa de persistencia no debe ofrecer ningún servicio para añadir a la base de datos objetos de las clases "Ambulancia", "Hospital" y "Especialidad"
5. Conexión de la capa de lógica y la de persistencia para poder trabajar con esta versión

Obtener la aplicación extendida que dé soporte a esta nueva versión.

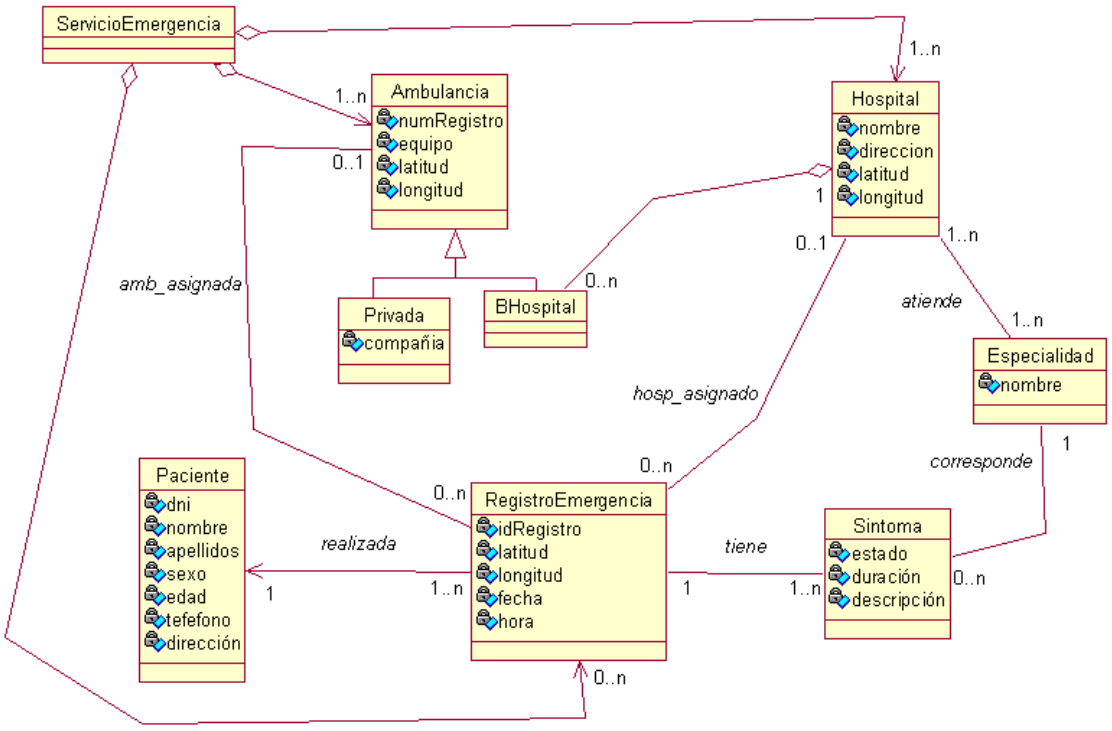

Figura 11. Diagrama de Clases adaptado para Persistencia

(*) El diseño lógico - relacional quedaría:

1. Tabla "AMBULANCIA"

AMBULANCIA (NUMREGISTRO, EQUIPO, LATITUD, LONGITUD).

CP { NUMREGISTRO }

VNN { EQUIPO, LATITUD, LONGITUD }

2. Tabla "HOSPITAL"

HOSPITAL (NOMBRE, DIRECCION, LATITUD, LONGITUD).

CP { NOMBRE }

VNN { DIRECCION, LATITUD, LONGITUD }

3. Tabla "ESPECIALIDAD"

ESPECIALIDAD (NOMBRE).

CP { NOMBRE }

4. Tabla "PACIENTE"

PACIENTE (DNI, NOMBRE, APELLIDOS, DIRECCIÓN, TELÉFONO, EDAD, SEXO).

CP { DNI }

VNN { NOMBRE, APELLIDOS, DIRECCIÓN, TELÉFONO, EDAD, SEXO }

5. Tabla "REGISTROEMERGENCIA"

REGISTROEMERGENCIA (IDREGISTRO, LATITUD, LONGITUD, FECHA, HORA, DNIPACIENTE, IDAMBULANCIA, IDHOSPITAL).

CP { IDREGISTRO }

VNN { LATITUD, LONGITUD, DNIPACIENTE }

CAJ { DNIPACIENTE } -> PACIENTE (DNI)

CAJ { IDAMBULANCIA } -> AMBULANCIA (NUMREGISTRO)

CAJ { IDHOSPITAL } -> HOSPITAL (NOMBRE)

6. Tabla “BHOSPITAL”

BHOSPITAL (NUMREGISTRO, IDHOSPITAL).

CP { NUMREGISTRO }

VNN { IDHOSPITAL }

CAJ { NUMREGISTRO } -> AMBULANCIA

CAJ { IDHOSPITAL } -> HOSPITAL (NOMBRE)

7. Tabla “ATIENDE”

ATIENDE (IDESPECIALIDAD, IDHOSPITAL).

CP { IDESPECIALIDAD, IDHOSPITAL }

CAJ { IDESPECIALIDAD } -> ESPECIALIDAD (NOMBRE)

CAJ { IDHOSPITAL } -> HOSPITAL (NOMBRE)

8. Tabla “SINTOMA”

SINTOMA (IDESPECIALIDAD, IDREGISTRO, ESTADO, DURACIÓN, DESCRIPCION).

CP { IDESPECIALIDAD, IDREGISTRO }

VNN { ESTADO, DESCRIPCION }

CAJ { IDESPECIALIDAD } -> ESPECIALIDAD (NOMBRE)

CAJ { IDREGISTRO } -> REGISTROEMERGENCIA

9. Tabla “PRIVADA”

PRIVADA (NUMREGISTRO, COMPAÑIA).

CP { NUMREGISTRO }

VNN { COMPAÑIA }

CAJ { NUMREGISTRO } -> AMBULANCIA