PATRONES DE DISEÑO

Francisco Martí Polo – Gestión de inmobiliaria

Índice

1. Proyecto

1.1 Descripción

El proyecto realizado trata de la aplicación para la gestión de una inmobiliaria en la cual podemos controlar los aspectos fundamentales de gestión como pueden ser dar de alta a nuevos clientes, a nuevos empleados, inmuebles, ofertas, visitas, etc.

1.2 Descripción general

El proyecto inicial de la inmobiliaria es el realizado en la asignatura de ISW, en la cual desarrollamos una aplicación para la gestión de una inmobiliaria.

La aplicación permitirá controlar toda la información que sea útil para la inmobiliaria, como podrían ser los inmuebles, la bolsa de clientes, las visitas a los inmuebles, las ofertas realizadas por los clientes, etc. De esta manera los asesores de la inmobiliaria serán los que interactúen con la aplicación pudiendo dar de alta a nuevos asesores, clientes, inmuebles (pisos, naves industriales, casas de pueblo, garajes, etc.), visitas que realicen los asesores con los clientes a los inmuebles que estén disponibles, y las ofertas ofrecidas por los clientes a un inmueble concreto.

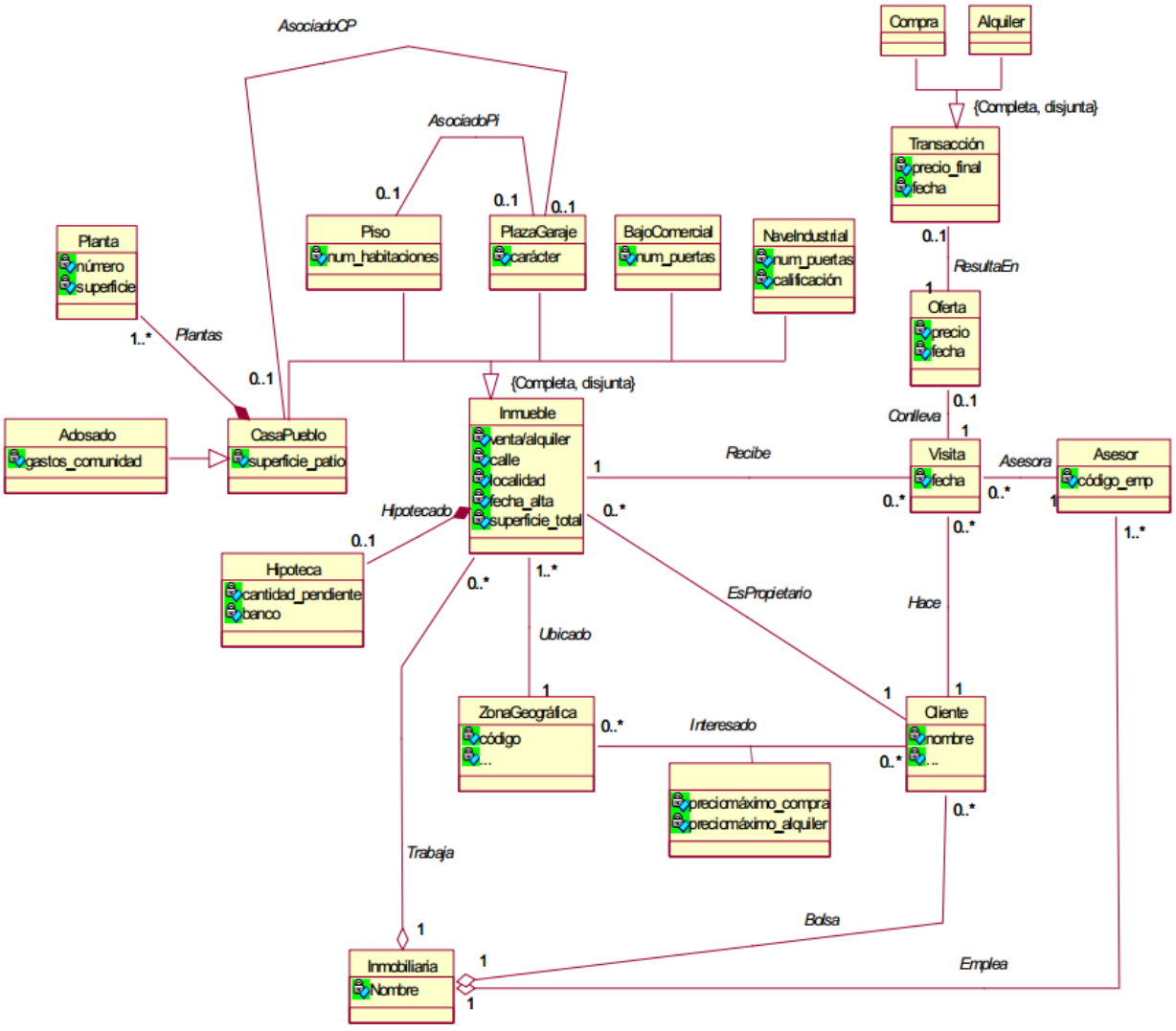
1.3 Implementación realizada

La aplicación entregada para el proyecto consta de parte de la funcionalidad de la aplicación completa, ya que solo se han llegado a desarrollar los casos de usos propuestos en la asignatura de ISW.

Los casos de usos implementados son los siguientes:

- Dar de alta nuevos inmuebles (Pisos y naves industriales)
- Dar de alta nuevos clientes
- Dar de alta nuevos asesores
- Crear visitas a inmuebles
- Crear ofertas a inmuebles por parte de los clientes
- Crear nuevas transacciones
- Consultar inmuebles
- Consultar las visitas realizadas con las ofertas correspondientes
- Consultar inmuebles disponibles
- Listar los clientes con los inmuebles asociados
- Listar los asesores con los inmuebles asociados

1.4 Diagrama de clases completo de la aplicación



2. Tecnología utilizada

Se han utilizado varias tecnologías para el desarrollo de la aplicación. En primer lugar la aplicación está desarrollada en java, y para ello se ha utilizado la plataforma de desarrollo Eclipse, la versión Kepler. Para el diseño de la base de datos se ha usado el plugin Clay, ya que te permite hacer el diseño de la base de datos mediante una interfaz bastante simple y clara. El plugin SQL-Explorer ha sido el elegido para crear las tablas de la base de datos mediante un script en SQL obtenido a partir del diseño con el Clay. Para crear las interfaces de usuario se ha utilizado el plugin Windows Builder. Finalmente como gestor de base de datos se ha utilizado el gestor hsqldb.

La instalación de estos plugins será descrita en el capítulo 7 Manual de instalación

3. Diseño Arquitectónico

3.1 Descripción

Para una mejor distribución de la aplicación y un mayor nivel de abstracción se ha dividido el proyecto en 3 capas diferentes: capa lógica, capa persistencia y capa presentación. De este modo tenemos un mayor control sobre la aplicación y sobre la gestión de los datos.

3.2 Capa Lógica o de negocio

Esta es la capa donde se sitúa toda la lógica de negocio de la aplicación. Dicho de otro modo, es la capa donde situamos todos los programas que se ejecutarán, y todas las reglas y condiciones que tendrá que cumplir nuestra aplicación. Esta capa será la encargada de comunicarse con la capa de presentación, a través de la cual recibirá solicitudes y presentará los resultados al usuario, y con la capa de persistencia, para establecer una comunicación con el gestor de la base de datos y así poder almacenar y consultar datos.

3.3 Capa Persistencia

En esta capa residirán los datos que sean manejados por toda la aplicación y desde aquí se realizará el acceso a los mismos. Se atenderán por tanto las peticiones que provengan de la capa de lógica o negocio, estando alojado en la capa persistencia el gestor de la base de datos.

3.4 Capa presentación

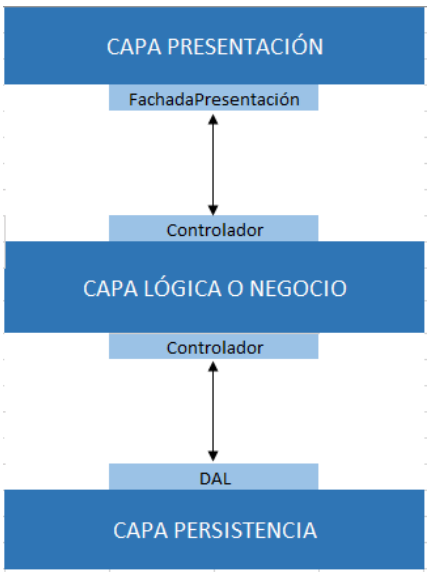
Esta capa será la encargada de mostrar información al usuario a través de la interfaz gráfica, y de recoger la información que proporcione el usuario y mandarla a la capa de lógica, para así realizar los cálculos necesarios o guardar/recuperar datos.

3.5 Aplicación a nuestro proyecto

En nuestro proyecto hemos creado la clase Controlador en la capa lógica y la clase DAL en la capa persistencia formando la siguiente estructura.

En la capa de persistencia se ha añadido también una clase llamada FachadaPresentación que forma parte de uno de los patrones aplicados en el proyecto. La clase FachadaPresentación es la encargada de recopilar los datos de las distintas interfaces y de hacerle llegar esos datos a la clase controlador.

3.6 Estructura del diseño arquitectónico



4. Patrones de diseño aplicados

4.1 Patrones utilizados para implementar la aplicación

Al haber escogido la aplicación de la gestión de la inmobiliaria que desarrollamos en la asignatura de ISW, hay patrones que implementamos cuando estábamos desarrollando la aplicación, y son los siguientes:

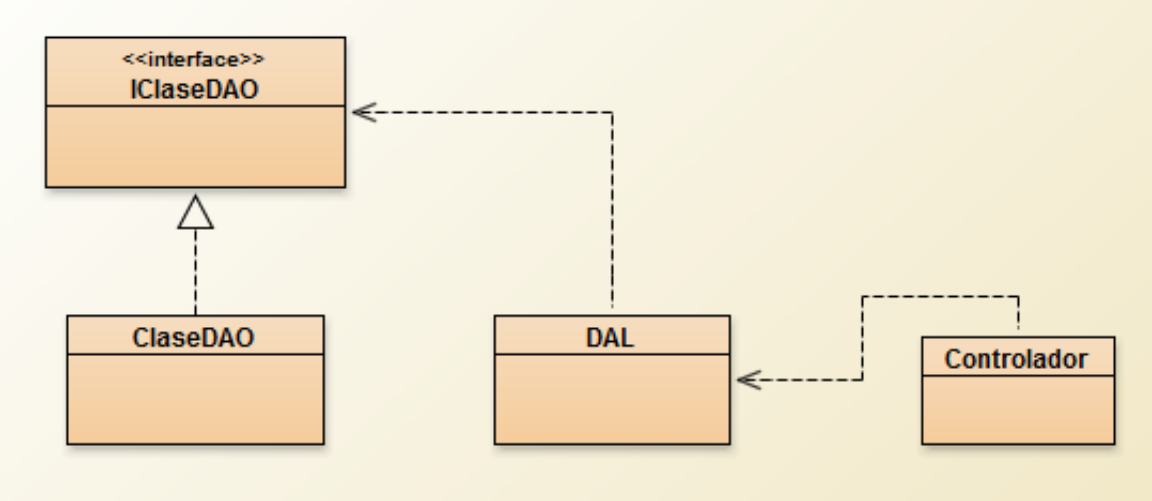
4.1.1 DAO (Data AccessObject) y Fachada

El patrón DAO se implementó en la capa de persistencia. Se implementó una clase DAO para cada una de las tablas de la base de datos con los métodos necesarios para consultarlas y modificarlas.

Implementando este patrón es más sencillo detectar los errores que se puedan producir a la hora de acceder a la base de datos. Además de que si se quisiera cambiar el gestor de la base de datos, tan solo tendríamos que cambiar las clases DAO, dejando las demás sin modificar.

Con el patrón DAO está muy relacionado el patrón Fachada. Se han creado dos clases: clase DAL (capa de persistencia) y clase Controlador (capa lógica), que son las encargadas de comunicarse entre sí para así comunicar las dos capas. Todas las clases DAO se comunican con el DAL, y este a su vez manda la información correspondiente al controlador de la clase lógica. De esta manera

La estructura de los dos patrones juntos es la siguiente:



4.1.2 Singleton

Otro de los patrones que se utilizó fue el patrón Singleton para la clase *Controlador*. De esta forma nos aseguramos de que solo se creará una instancia de la clase *Controlador* que es la que se encarga de lanzar las peticiones a la base de datos, por lo que creamos un sistema global desde toda la aplicación, pero con una única instancia.

El código necesario para aplicar el patrón es el siguiente:

```
private static Controlador control = null;
public static Controlador getInstance() throws LogicaExcepcion {
    if(control == null)
        control = new Controlador();
    return control;
}
```

4.1.3 Iterador

Este patrón nos ha permitido recorrer los elementos que recogemos de la base de datos de forma secuencial y sin revelar su representación interna. Como el recorrido que teníamos que hacer era completamente secuencial no hemos tenido que reescribir la clase Iterador que proporciona la librería java.util, por lo que su utilización en alguna de las clases es la siguiente:

```
public void cargaInmuebles(){
    try{
        ArrayList<Inmueble> listaInmuebles= control.encontrarInmuebles();
        Iterator<Inmueble> it = listaInmuebles.iterator();
        Inmueble in;
        InmuebleTableModel model = (InmuebleTableModel) tableInmuebles.getModel();
        model.clear();
        while (it.hasNext()){
            in=it.next();
            model.addRow(in);
        }
    }
    catch (Exception e){
        JOptionPane.showMessageDialog(this,e.getMessage(),"ERROR", JOptionPane.ERROR_MESSAGE);
    }
}
```

4.2 Patrones implementados en este proyecto

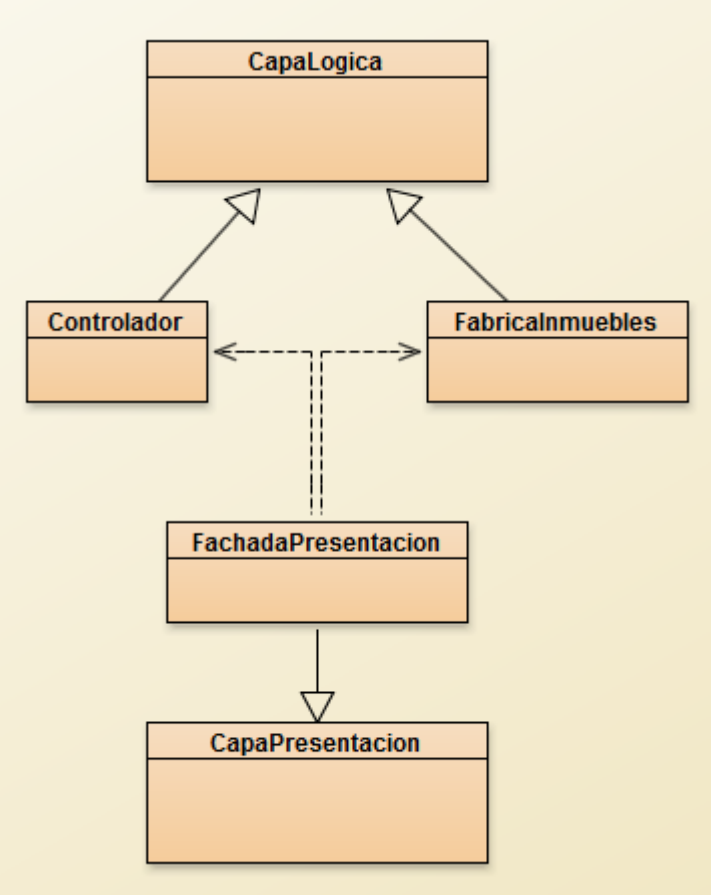
A continuación se describirían los nuevos patrones que se han añadido a la aplicación, de forma que mejoraremos alguno de los patrones ya implementados y añadiremos alguno más.

En este proyecto se han utilizado 4 patrones, que son el patrón Singleton, el patrón Fachada, el patrón Comando y el método Fábrica. Más adelante se podrá comprobar que algunos de estos patrones están relacionados entre sí.

4.2.1 Fachada

Este patrón se ha aplicado en la capa de presentación, ya que se había observado que las otras dos capas tenían una clase que hacía de fachada para así comunicarse con las demás, por lo que se ha optado por realizar lo mismo en la capa presentación. Antes, cualquier clase de la capa de presentación se comunicaba directamente con la capa lógica, pero ahora se ha centralizado todos los accesos a esta capa creando una nueva clase llamada FachadaPresentación, la cual centraliza todas las peticiones que se hacen desde la esta capa a la capa lógica, ya sea a la clase Controlador o FabricaInmuebles.

La estructura del patrón al aplicarlo es la siguiente:



Un ejemplo de código sería el siguiente para la clase FachadaPresentación (solo se muestra una parte del código de la clase):

```
public class FachadaPresentacion {

    private static FachadaPresentacion fachada = null;
    private FabricaInmueble fabrica = null;
    private Controlador control = null;

    private FachadaPresentacion() throws LogicaExcepcion{
        fabrica = FabricaInmueble.getInstance();
        control = Controlador.getInstance();
    }

    public static FachadaPresentacion getInstance() throws LogicaExcepcion{
        if(fachada == null){
            fachada = new FachadaPresentacion();
        }
        return fachada;
    }


    @SuppressWarnings("rawtypes")
    public Piso crearPiso(ArrayList lista) throws LogicaExcepcion{
        return (Piso) fabrica.crearInmueble(lista);
    }
```

Ejemplo de llamada desde una clase de la capa de persistencia:

```
try {
    c = fachada.encontrarClientePorCod(cl[0]);
    a = fachada.encontrarAsesorPorCod(as[0]);

    ArrayList lista = new ArrayList();
    lista.add("Piso");
    lista.add(codigo.getText());
    lista.add(direccion.getText());
    lista.add(localidad.getText());
    lista.add(fecha.getText());
    lista.add(superficie.getText());
    lista.add(opcion);
    lista.add(habitaciones.getText());
    lista.add(c);
    lista.add(a);

    control.crearPiso(fachada.crearPiso(lista));
    dispose();

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

Con esta fachada conseguimos un mayor nivel de abstracción entre las capas.

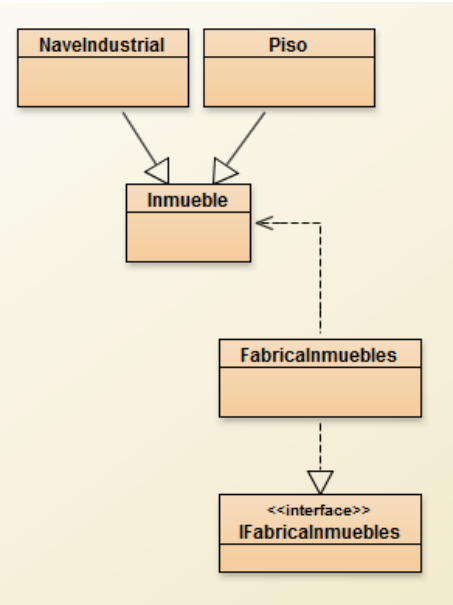
4.2.2 Método Fábrica

El método fábrica se ha aplicado en dos ocasiones, una en la capa de lógica creando la clase FabricaInmuebles, y otra en la capa Presentación creando la clase FabricaPresentacion.

De esta forma podemos crear los elementos del mismo tipo de una forma más organizada y quitando código de la interfaz principal.

En ambos casos se ha creado una interfaz de la cual implementan las correspondientes clases que actúan de fábrica, en ellas se ha creado un método para crear los diferentes tipos de objetos. Este método consta de un parámetro para elegir el tipo de objeto que se desea crear.

Un ejemplo (FabricaInmuebles) y su estructura simplificada se presentan a continuación:



```
private static FabricaInmueble fabrica = null;

public static FabricaInmueble getInstance() throws LogicaExcepcion {
    if(fabrica == null)
        fabrica = new FabricaInmueble();
    return fabrica;
}

@SuppressWarnings("rawtypes")
public Inmueble crearInmueble(ArrayList lista){

    if(lista.get(0).toString().equals("Inmueble")){
        return new Inmueble(lista.get(1).toString(),lista.get(2).toString(),lista.get(3).toString(),lista.get(4).toString(),
                lista.get(5).toString(),lista.get(6).toString(),(Cliente)lista.get(7),(Asesor)lista.get(8));
    }

    if(lista.get(0).toString().equals("Piso")){
        return new Piso(lista.get(1).toString(),lista.get(2).toString(),lista.get(3).toString(),lista.get(4).toString(),
                lista.get(5).toString(),lista.get(6).toString(),lista.get(7).toString(),
                (Cliente)lista.get(8),(Asesor)lista.get(9));
    }

    if(lista.get(0).toString().equals("Nave Industrial")){
        return new NaveIndustrial(lista.get(1).toString(),lista.get(2).toString(),lista.get(3).toString(),lista.get(4).toString(),
                lista.get(5).toString(),lista.get(6).toString(),lista.get(7).toString(),lista.get(8).toString(),
                (Cliente)lista.get(9),(Asesor)lista.get(10));
    }
```

Cabe destacar que normalmente este patrón tiene un método en su fábrica al cual se le pasa un código indicándole que tipo de objeto queremos crear, pero en este caso he tenido el problema de que dependiendo del tipo de objeto necesitaba recibir unos parámetros u otros, así que opté por pasarle un ArrayList al método en el cual contenía en la primera posición el tipo de objeto que debía crear y en las demás posiciones los parámetros necesarios el objeto en concreto.

4.2.3 Comando

Se ha creado en la capa de persistencia una nueva interfaz llamada Comando y una nueva clase llamada SentenciaSQL, la cual implementa tres métodos de la interfaz Comando: Uno para establecer la conexión con la base de datos, otro para hacer actualizaciones y otro para hacer consultas sobre la base de datos.

De esta forma estamos consiguiendo unas clases DAO con menos implementación, ya que delegan la responsabilidad de la gestión de la base de datos a la clase SentenciaSQL, pasándole por parámetro únicamente la sentencia que se quiera ejecutar en la base de datos.

El código antes de aplicar el patrón es el siguiente (solo se mostrará un ejemplo concreto):

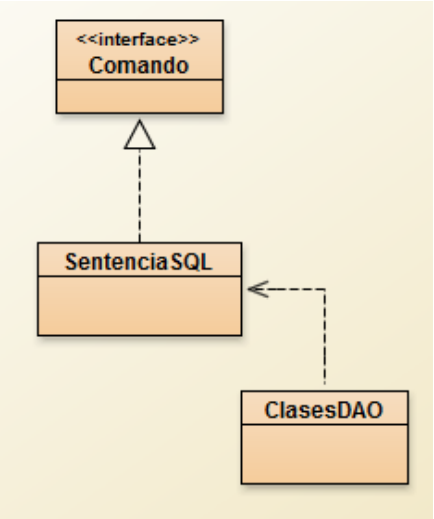
```
public void crearAsesor(Asesor as) throws DAOExcepcion {

    try{
        connManager.connect();
        connManager.updateDB("insert into ASESOR (CODIGO_EMPLEADO, NOMBRE, APELLIDOS) values ('"+
        as.getCodigo_emp()+"','"+as.getNombre()+"','"+as.getApellidos()+"')");
        connManager.close();
    }
    catch (Exception e){
        throw new DAOExcepcion(e);
    }
}
```

Al aplicar el patrón vemos como el código se reduce bastante:

```
public void crearAsesor(Asesor as) throws DAOExcepcion {
    manejador.updateSQL("insert into ASESOR (CODIGO_EMPLEADO, NOMBRE, APELLIDOS) values ('"+
                        as.getCodigo_emp()+"','"+as.getNombre()+"','"+as.getApellidos()+"')");
}
```

Finalmente la estructura del patrón comando es la siguiente:



4.2.4 Singleton

Este patrón ya se aplicó, como se ha explicado anteriormente, cuando la aplicación se estaba implementando, pero se ha vuelto a utilizar en las clases FabricaInmuebles, SentenciaSQL, FabricaPresentación y FachadaPresentación para así poder tener únicamente una instancia global de esas clases en toda la aplicación.

5. Refactorización

5.1 Rename Variable

Una de las refactorizaciones aplicadas es la de renombrar variables que pueden llevar a la confusión por el nombre que utilizan. Uno de los renombramientos es el siguiente:

Código anterior:

```
else{
    textId.setText("");
    txtPrecio.setText("");
    ofertasVisitas.addItem("Seleccione una oferta...");
    ArrayList<Visita> visitas = control.encontrarVisitasPorAsesor(asesor);
    Iterator<Visita> i = visitas.iterator();
    while(i.hasNext()){
        Visita v = i.next();
        Inmueble in = v.getInmueble();
        ArrayList<Oferta> of = control.encontrarOfertaPorVisita(v.getCod_Id());
        if(!of.isEmpty()){
            Oferta o = of.get(0);
            if(!control.comprobarOfertaTransaccion(o.getCod_Id())){
                ofertasVisitas.addItem("IN: "+in.toString()+" - OF: "+o.toString()+" - VIS: "+v.toString());
            }
        }
    }
```

Código resultante:

```
else{
    textId.setText("");
    txtPrecio.setText("");
    ofertasVisitas.addItem("Seleccione una oferta...");
    ArrayList<Visita> visitas = control.encontrarVisitasPorAsesor(asesor);
    Iterator<Visita> iterator = visitas.iterator();
    while(iterator.hasNext()){
        Visita visita = iterator.next();
        Inmueble inmueble = visita.getInmueble();
        ArrayList<Oferta> ofertaVisita = control.encontrarOfertaPorVisita(visita.getCod_Id());
        if(!ofertaVisita.isEmpty()){
            Oferta oferta = ofertaVisita.get(0);
            if(!control.comprobarOfertaTransaccion(oferta.getCod_Id())){
                ofertasVisitas.addItem("IN: "+inmueble.toString()+" - OF: "+oferta.toString()+" - VIS: "+
                        visita.toString());
            }
        }
    }
    ofertasVisitas.setEnabled(true);
    lblOfertasVisitas.setEnabled(true);
}
```

Como podemos comprobar, se ha hecho una refactorización de 4 variables diferentes.

5.2 Encapsulate Field

Esta refactorización se ha llevado a cabo en varias clases, pero como ejemplo sólo se pondrá una de ellas.

En código anterior a la refactorización era el siguiente:

```
public class ConsultarInmueblesNoVendidosNiAlquiladosJDialog extends JDialog {
    private JTable tableInmuebles;
    Controlador control;
```

Y el código resultante tras la refactorización queda de la siguiente manera:

```
public class ConsultarInmueblesNoVendidosNiAlquiladosJDialog extends JDialog {
    private JTable tableInmuebles;
    private Controlador control;
```

Algunas de las refactorizaciones se hicieron mientras se introducían los nuevos patrones de diseño, por lo que esta refactorización en concreto más tarde fue sustituida, ya que se aplicó el patrón fachada, y el controlador dejó de ser útil en esta clase.

5.3 Extract Method

En la clase persistencia se usaban repetidas veces varias instrucciones para conectarse a la base de datos, o para hacer consultas. Se ha creado varios métodos parametrizados en la clase SentenciaSQL (manejador) para acortar código, este es un ejemplo concreto.

Código anterior:

```
public void crearInmueble(Inmueble in) throws DAOExcepcion {
    try{
        connManager.connect();
        connManager.updateDB("insert into INMUEBLE (COD_ID, ASESOR, CALLE, CLIENTE, FECHA_ALTA, LOCALIDAD, SUPERFICIE_TOTAL, VENTA_ALQUILER) values ('"+
                in.getCod_Id()+"','"+in.getAsesor().getCodigo_emp()+"','"+in.getCalle()+"', '"+in.getCliente().getNifCliente()+"', '"+
                in.getFecha_Alta()+"', '" +in.getLocalidad()+"', '"+in.getSuperficie_Total()+"', '"+in.getVenta_Alquiler()+"')");
        connManager.close();
    }
    catch (Exception e){
        throw new DAOExcepcion(e);
    }
}
```

Código resultante:

Clase InmuebleDAOImp:

```
public void crearInmueble(Inmueble in) throws DAOExcepcion {
    manejador.updateSQL("insert into INMUEBLE (COD_ID, ASESOR, CALLE, CLIENTE, FECHA_ALTA, LOCALIDAD, SUPERFICIE_TOTAL, VENTA_ALQUILER) values ('"+
            in.getCod_Id()+"','"+in.getAsesor().getCodigo_emp()+"','"+in.getCalle()+"', '"+in.getCliente().getNifCliente()+"', '"+in.getFecha_Alta()+"', '"+
            in.getLocalidad()+"', '"+in.getSuperficie_Total()+"', '"+in.getVenta_Alquiler()+"')");

}
```

Clase SentenciaSQL (manejador):

```
public void updateSQL(String sentencia){
    try {
        connManager.connect();
        connManager.updateDB(sentencia);
        connManager.close();
    } catch (DAOExcepcion e) {e.printStackTrace();}
}
```

5.4 Inline Temp

Código anterior:

```
public void cargaInmuebles(){
    try{
        ArrayList<Inmueble> listaInmuebles= fachada.encontrarInmuebles();
        Iterator<Inmueble> iterator = listaInmuebles.iterator();
```

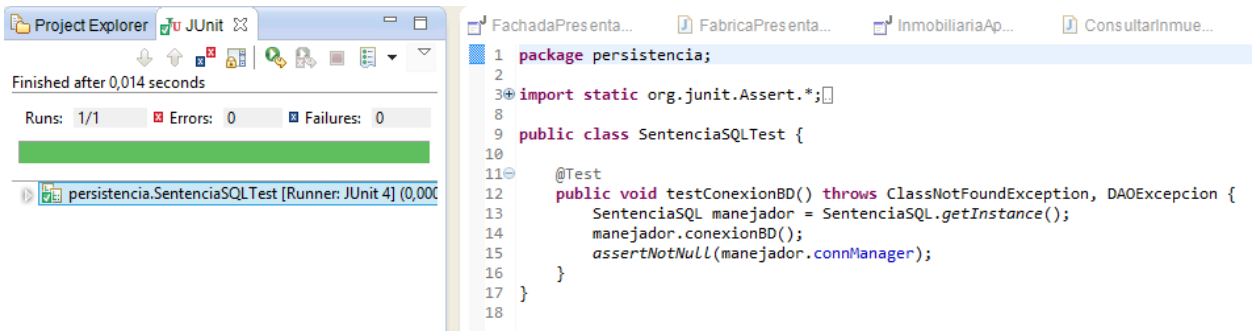
Código resultante:

```
public void cargaInmuebles(){
    try{
        Iterator<Inmueble> iterator =  fachada.encontrarInmuebles().iterator();
```

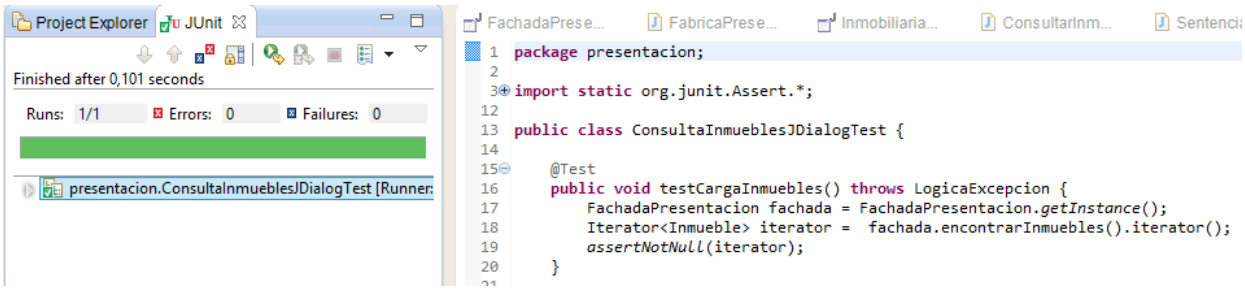
6. Pruebas con JUnit

Se han realizado algunas pruebas para comprobar que la aplicación sigue funcionando como toca después de la refactorización.

Pruebas del método de conexión a la base de datos:



Prueba de Inline Temp:



7. Manual de instalación

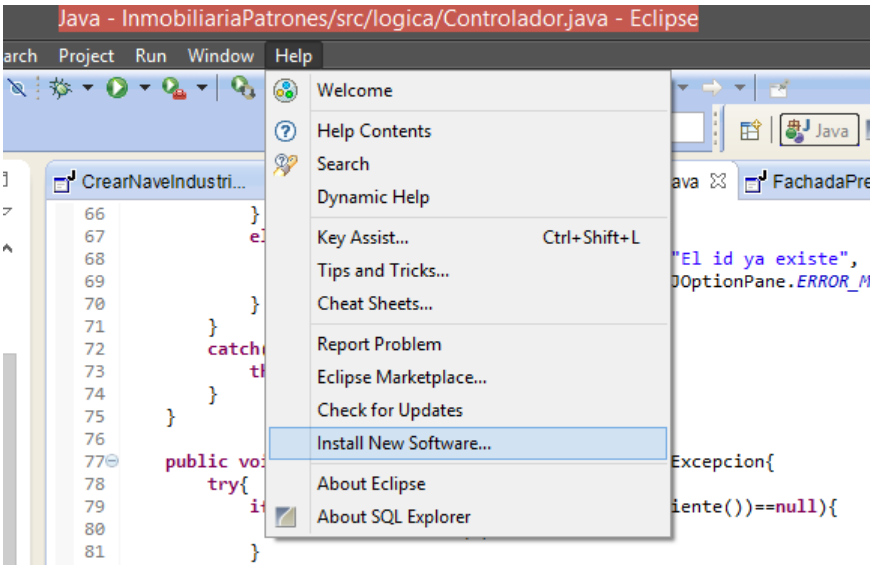
En el archivo comprimido que se adjunta junto a esta memoria se encuentra una **carpeta con el proyecto** llamada **"inmobiliariaPatrones"**, otra carpeta llamada **"Ejecutables"** que contiene los archivos para lanzar la base de datos tanto para Mac, Linux y Windows; y el jar ejecutable de la aplicación. En el proyecto también se adjunta un script de la base de datos. Aunque se proporciona la base de datos ya creada.

Como se explicó al principio de este documento, este proyecto ha sido desarrollado con la plataforma de desarrollo Eclipse, y para su correcto funcionamiento se recomienda la versión estándar (Kepler 4.3). Descargar Eclipse Kepler.

Una vez instalado Eclipse, el proyecto está preparado para ser importado y ejecutado. En primer lugar lance la base de datos mediante su acceso directo en la carpeta "Ejecutables", a continuación copie la carpeta "inmobiliariaPatrones" al workspace, diríjase a Eclipse y dele a **File -> Import… -> Existing Projects into Workspace**, y seleccione el proyecto InmobiliariaPatrones. Después de importar el proyecto, podrá ejecutar la aplicación lanzando la clase InmobiliariaApp incluida en el paquete presentación que está dentro de src.

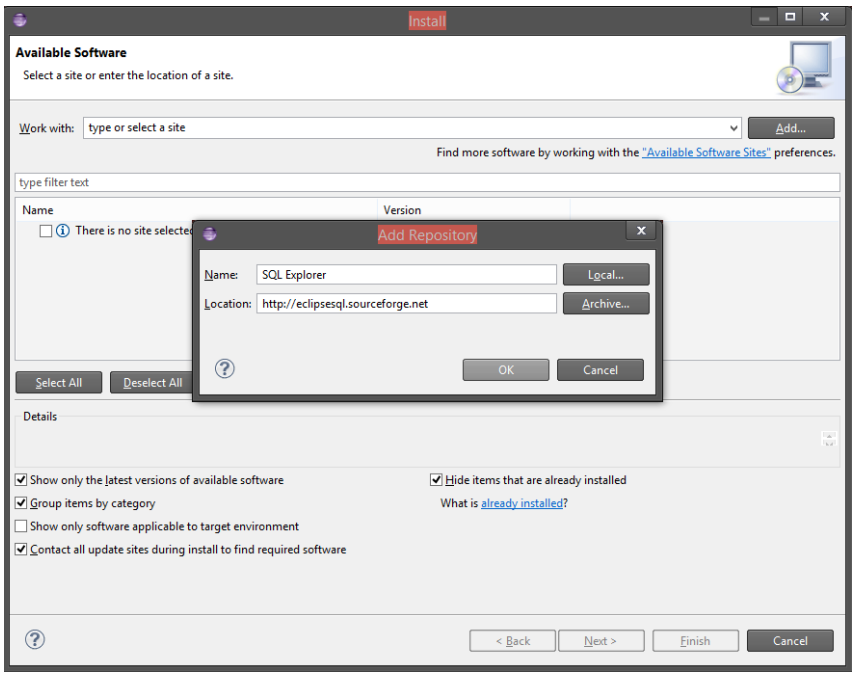
Se detallará ahora la instalación de los plugins necesarios y la instalación del driver de la base de datos**, proceso que no es necesario para revisar el proyecto o ejecutarlo**

Para instalar los plugins iremos a la **barra de herramientas** de Eclipse y seleccionaremos **Help -> Install New Software…**



A continuación nos saldrá una ventana, clicamos sobre el botón **Add…,** y rellenaremos los dos campos que nos aparecerán con la siguiente información, tal y como se muestra en la imagen, después seleccionaremos el plugin y continuaremos con la instalación aceptando los términos.

(El nombre y la dirección de cada plugin están después de la imagen)



Repetir el proceso indicado arriba para estos casos:

- Descargar SQL explorer

Name: SQL Explorer

Location: http://eclipsesql.sourceforge.net

Después de reiniciar Eclipse podremos abrir la perspectiva SQL Explorer dándole a **Window -> Open Perspective -> SQL Explorer**

- Instalar el plugin WindowsBuilder

Name: Windows Builder

Location: http://download.eclipse.org/windowbuilder/WB/release/R201309271200/4.3/

- Instalar Clay

Name: Clay

Location: http://www.azzurri.co.jp/eclipse/plugins/

NOTA: Este plugin ya nos está disponible de forma gratuita, así que se puede omitir este paso ya que no es necesario para ver el proyecto en funcionamiento. Este plugin solo lo utilizamos para el modelado de la base de datos.

Aunque la base de datos va incluida en el proyecto, tendremos que preparar a eclipse para trabajar con ella, importando el driver de la base de datos. Para ello iremos a **Window -> Preferences -> SQL Explorer -> JDBC drivers** y marque "HSQLDB Server"; haga clic en el botón "Set Default", y luego en "Edit".

En la ventana "Change Driver", en la pestaña "Extra Class Path" haga clic en "Add Jars" y añada el jar de la base de datos (hsqldb.jar) que se encuentra **inmobiliariaPatrones -> hsqldb -> lib -> hsqldb.jar**. Pulsar "List Drivers" y después pulsar OK.

Una vez hecho esto tenemos que crear la conexión de la base de datos. No se olvide de ejecutar primero la base de datos desde la carpeta "Ejecutables".

Abrir la **vista SQL Explorer** en Eclipse, hacer clic en el botón "New Connection Profile" de la pestaña "Connections". Los valores a introducir en el dialogo son los siguientes:

- **Nombre de la conexión:** practica4
- **Driver:** 'HSQLDB Server'
- **URL:** jdbc:hsqldb:hsql://localhost/practica4
- **Marcar** la casilla "Username is not required for…"

Ahora Eclipse ya está preparado para el desarrollo de la aplicación.

8. Manual de usuario

La aplicación es muy sencilla de utilizar. La única observación que se debe hacer es que como podrá comprobar, primero tiene que dar de alta Clientes y Asesores para poder crear nuevos inmuebles. Y una vez hecho esto, podrá crear visitas, luego ofertas relacionadas con las visitas a cada inmueble, y finalmente transacciones sobre los inmuebles.

9. Observaciones

En el archivo comprimido que se adjunta con la memoria se encuentra el proyecto inicial y el proyecto final una vez aplicados los patrones. Este proyecto final cuenta con algunas mejoras respecto al proyecto final, ya que aunque no eran relevantes para el objetivo de la asignatura, he creído conveniente realizar esos cambios y mejoras para mejorar el funcionamiento de la aplicación. En esos cambios se incluye el control de errores de algunas funcionalidades, así como más mejoras internas triviales en relación con la asignatura.