www.uniremington.edu.co

UNIREMINGTON®
CORPORACIÓN UNIVERSITARIA REMINGTON
RES. 2661 MEN JUNIO 21 DE 1996

LENGUAJE DE PROGRAMACIÓN II

TRANSVERSAL

FACULTAD DE CIENCIAS BÁSICAS E INGENIERÍA

Vicerrectoría de Educación a Distancia y virtual

2016

AUTOR

Cesar Augusto Jaramillo Henao
Ingeniero de Sistemas
Cesar.jaramillo@uniremington.edu.co

Nota: el autor certificó (de manera verbal o escrita) No haber incurrido en fraude científico, plagio o vicios de autoría; en caso contrario eximió de toda responsabilidad a la Corporación Universitaria Remington, y se declaró como el único responsable.

RESPONSABLES

Jorge Mauricio Sepúlveda Castaño
Decano de la Facultad de Ciencias Básicas e Ingeniería
jsepulveda@uniremington.edu.co

Eduardo Alfredo Castillo Builes
Vicerrector modalidad distancia y virtual
ecastillo@uniremington.edu.co

Francisco Javier Álvarez Gómez
Coordinador CUR-Virtual
falvarez@uniremington.edu.co

GRUPO DE APOYO

Personal de la Unidad CUR-Virtual
EDICIÓN Y MONTAJE

Primera versión. Febrero de 2011.
Segunda versión. Marzo de 2012
Tercera versión. noviembre de 2015

TABLA DE CONTENIDO

Pág.

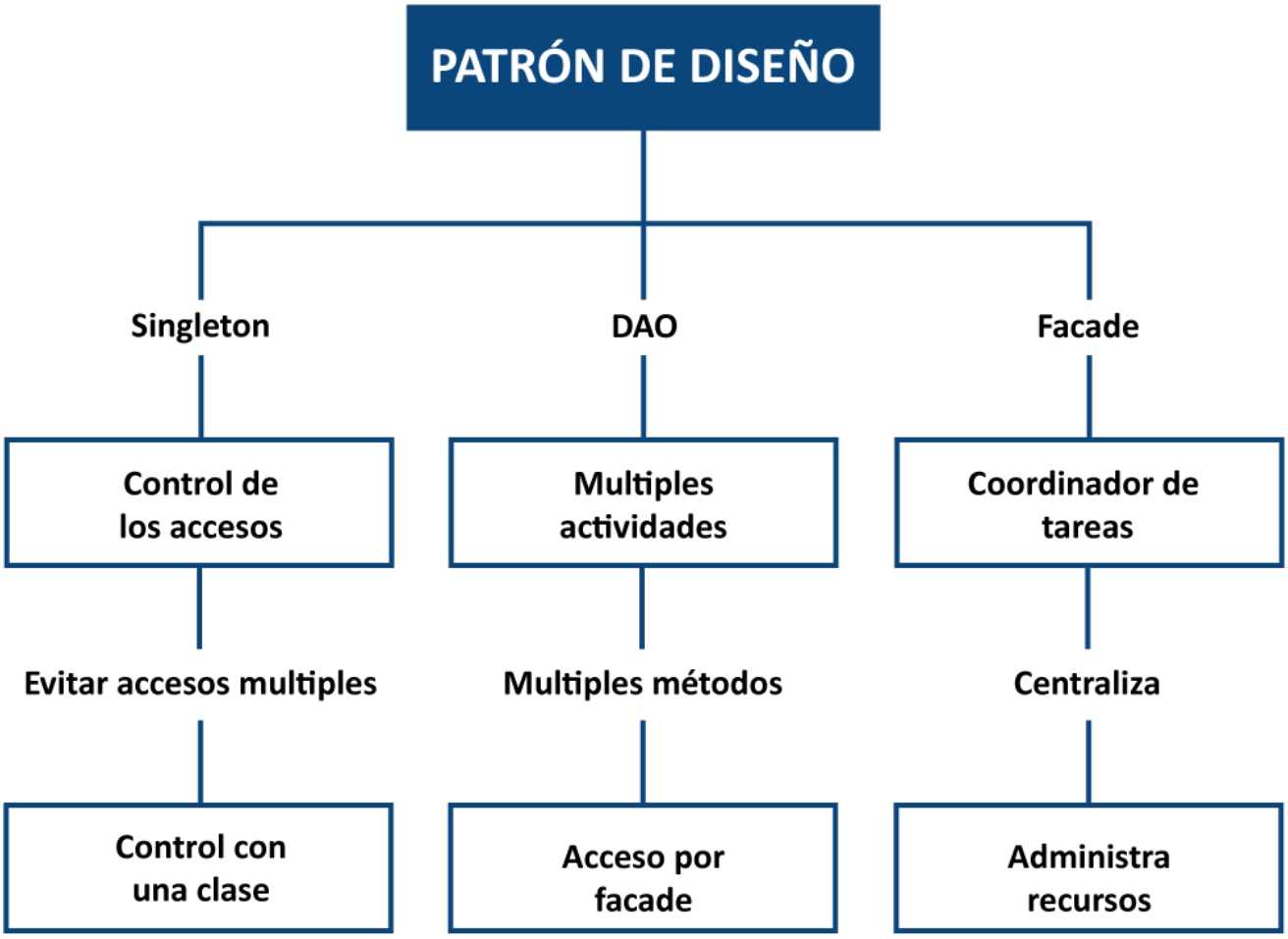
1 MAPA DE LA ASIGNATURA 5
2 UNIDAD 1 PATRONES DE DISENO 6
2.1.1 RELACIÓN DE CONCEPTOS 6
2.2 TEMA 1 SINGLETON 7
2.3 TEMA 2 DAO 9
3 TEMA 3 FACADE 10
3.1.1 EJERICICIO DE APRENDIZAJE 11
3.1.2 TALLER DE ENTRENAMIENTO 12
4 UNIDAD 2 PERSISTENCIA EN BASES DE DATOS 13
4.1 RELACIÓN DE CONCEPTOS 13
4.2 TEMA 1 APLICACIÓN CRUD 14
4.3 TEMA 2 POOL DE CONEXIONES 60
4.4 TEMA 3 REPORTES 62
4.5 TEMA 4 DOCUMENTACIÓN 81
4.5.1 EJERCICIO DE APRENDIZAJE 89
4.5.2 TALLER DE ENTRENAMIENTO 90
5 UNIDAD 3 HILOS 91
5.1 RELACIÓN DE CONCEPTOS 91
5.2 TEMA 1 DEFINICIÓN Y OBJETIVOS 92
5.3 TEMA 2 COMPONENTES 92
5.4 TEMA 3 IMPLEMENTACIÓN DE LA INTERFAZ RUNNABLE 93

5.5 TEMA 4 CICLO DE VIDA 95
5.6 TEMA 5 PRIORIDADES 96
5.7 TEMA 6 SINCRONIZACIÓN 98
5.7.1 EJERICICIO DE APRENDIZAJE 99
5.7.2 TALLER DE ENTRENAMIENTO 100
6 PISTAS DE APRENDIZAJE 101
7 GLOSARIO 102
8 BIBLIOGRAFÍA 103

1 MAPA DE LA ASIGNATURA

2 UNIDAD 1 PATRONES DE DISENO

2.1.1 RELACIÓN DE CONCEPTOS



Escriba la definición de todos los conceptos plateados en el mapa conceptual

Múltiples accesos: el singleton contrala las conexiones o los accesos

Múltiples métodos: existen múltiples métodos por tabla en la clase DAO

Centraliza: el facade centraliza y distribuye los recursos solicitados

Administra recursos: el facade administra todos los procesos que provea el DAO

Coordina tareas: Distribuye la información según la solicitud

OBJETIVO GENERAL

Introducir al alumno en el manejo de buenas prácticas, aplicando patrones de diseño como singleton, facade, DAO, pool de conexiones, entre otros.

OBJETIVOS ESPECÍFICOS

- Identificar los diferentes métodos de programación, los más útiles y los más escalables.
- identificar las metodologías de trabajo que se encuentran en el mercado para realizar proyecto con mejores prácticas.

2.2 TEMA 1 SINGLETON

Singleton o instancia única es un patrón de diseño que restringa la creación de objetos pertinentes, con esto se busca garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global con ella.

Con el patrón singleton se implementa creando en una clase un método que tiene una instancia al objeto si este no existe todavía, para evitar volver a instanciarla se regula con el constructor.

Dentro de este proceso encontramos algunas características como

- Ejemplo de Singleton

```
public class EjemploSingleton {

    protected EjemploSingleton() {

    }

   private static class SingletonHolder {

       private final static EjemploSingleton INSTANCE = new EjemploSingleton();
   }

   public static EjemploSingleton getInstance() {

       return SingletonHolder.INSTANCE;
   }
}
```

Autoria Propia

```
public class SingletonMain {

    public static void main(String arg[]){

        EjemploSingleton ejemploSingleton1 = EjemploSingleton.getInstance();
        EjemploSingleton ejemploSingleton2 = EjemploSingleton.getInstance();
        System.out.println(ejemploSingleton1 == ejemploSingleton2);
    }
}
```

Autoria Propia

PISTAS DE APRENDIZAJE

SINGLETON

Es una de las muchas herramientas en el universo de java, ideal para unas buenas prácticas de programación.

2.3 TEMA 2 DAO

DAO o el Data Access Object (Objeto de Acceso a Datos), los DAO son los proveedores de los accesos a la BD de serán necesarios durante la ejecución de la aplicación, cada DAO representa una tabla con los procesos específicos de trabajo, esto quiere decir que no tiene formato de presentación ante el usuario, solo será de procesamiento y tendrá tareas concretas como insertar, consultar, modificar, verificar, etc. Y serán invocados o usados por el formulario o proceso que lo requiera y cumplas las condiciones mínimas de trabajo.

- Ejemplo

```
public static void insertar (Cliente cliente){

    try {

        PreparedStatement registroCliente = cnn.prepareStatement("call insertarCliente (?, ?, ?, ?, ?, ?)");
        registroCliente.setString(1, cliente.getCedulaCliente());
        registroCliente.setString(2, cliente.getNombre());
        registroCliente.setString(3, cliente.getDireccion());
        registroCliente.setString(4, cliente.getTelefono());
        registroCliente.setString(5, cliente.getEmail());
        registroCliente.setString(6, "activo");
        registroCliente.executeUpdate ();
        JOptionPane.showMessageDialog(null, "Registro Almacenado");
    }
    catch (SQLException sqle) {

        sqle.printStackTrace ();
    }
}
```

Autoria Propia

Los metodos se manejarian todos de esta manera y seran invocados por el formulario, proceso o por una fachada que sirva como intermediario.

PISTAS DE APRENDIZAJE

DAO

Es una tecnología de gran utilidad, comprende la capa de lógica y no tiene ningún formato de presentación.

3 TEMA 3 FACADE

Facade o Fachada, es una clase muy simple que se encarga de administrar los recursos del proyecto, con este tipo de arquitectura se distribuye la información sin necesidad de ingresar directamente a la forma o formulario, todos los procesos se solicitan mediante la fachada y este distribuirá o solicitará la información en el DAO correspondiente.

En el siguiente ejemplo se encuentra que en un formulario Tipo Documento se esté consultando la fachada con el método consultar Empleado

```
sw=fachada.verificarTipoDocumentoUsuario(textCedula.getText());

switch(Opc){
case 1:
        if(sw==1){

                empleado=fachada.consultarEmpleado(textCedula.getText());
                textNombre.setText(empleado.getNombre());
                cboxTipo.setEnabled(true);
                cboxTipo.requestFocus();


        }else{
                JOptionPane.showMessageDialog(null,"El registro ya existe");
        }
        break;

}
```

Autoria Propia

En el siguiente proceso se encuentra que el formulario empleado también consultar la fachada consultar Empleado, ambos procesos llaman al mismo método de la misma fachada, en este caso se están aprovechando los recursos, que dentro de un archivo que coordina los llamados se invoque para distintas tareas el mismo método y con un resultado idéntico.

```
if(sw==1){
        empleado=fachada.consultarEmpleado(textCedula.getText());
        mostrar(empleado);
}else{
        JOptionPane.showMessageDialog(null,"No existe el registro");
}
break;
```

Autoria Propia

PISTAS DE APRENDIZAJE

FACADE

Se encarga de recibir solicitudes y enviar respuestas en distintos entes del proyecto

3.1.1 EJERICICIO DE APRENDIZAJE

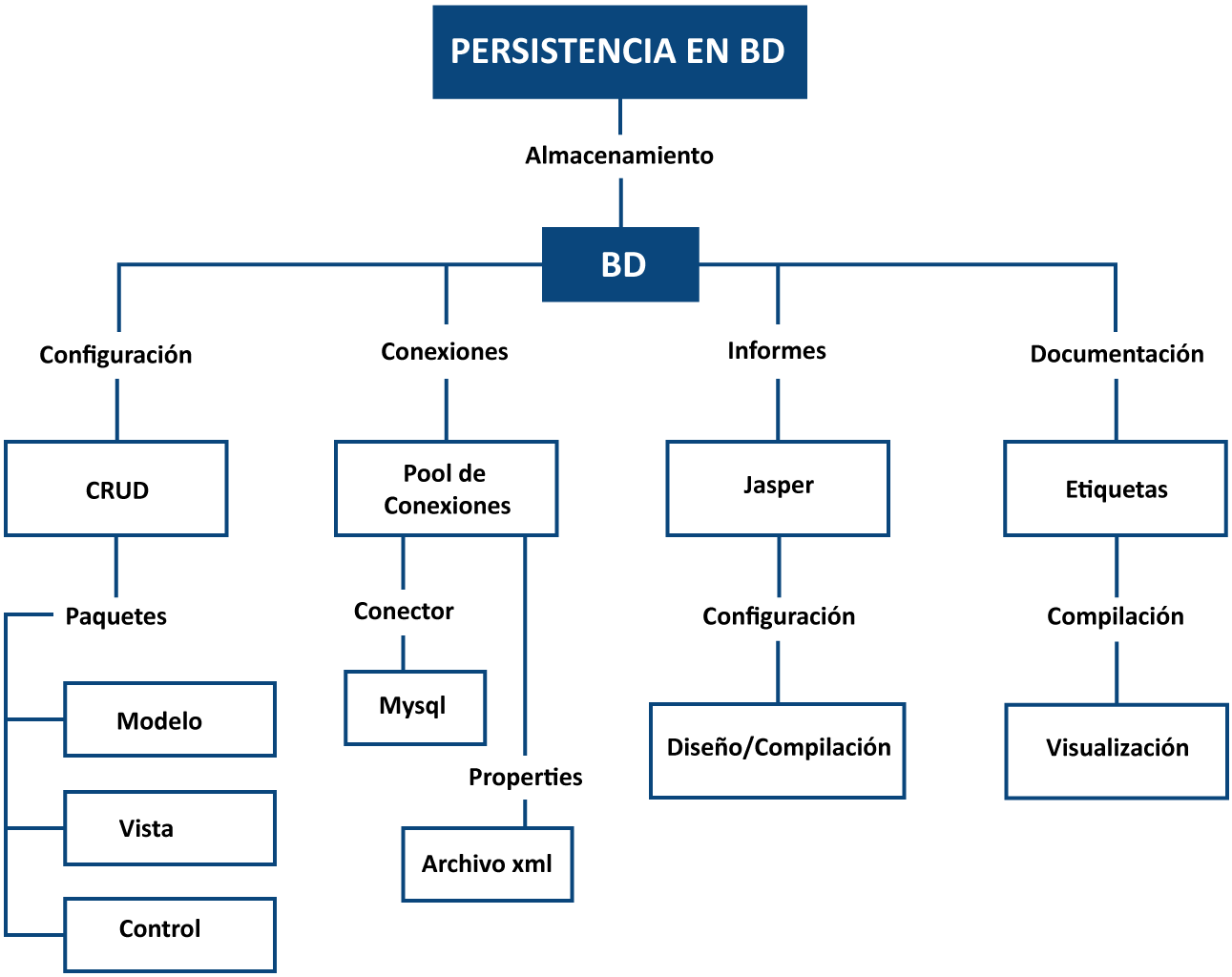
Nombre del taller de aprendizaje: Patrones	**Datos del autor del taller:** Cesar Augusto Jaramillo Henao
Escriba o plantee el caso, problema o pregunta: Que beneficios tiene trabajar con un patrón de diseño.	
Solución del taller: Los patrones hacen parte de las buenas prácticas de programación, si se aplican de la manera correcta los desarrollos serán mejor administrados y con unos recursos adecuados	

3.1.2 TALLER DE ENTRENAMIENTO

Nombre del taller: bancario	**Modalidad de trabajo:** individual
Actividad previa: Repase los conceptos del tema patrones de diseño y los de lenguaje I	
Describa la actividad: Cree un sistema bancario que permita calcular la tasa de interés de un préstamo en N cantidad de años, aplique patrones de diseño.	

4 UNIDAD 2 PERSISTENCIA EN BASES DE DATOS

4.1 RELACIÓN DE CONCEPTOS



BD: Estructura que permite el almacenamiento de información de forma organizada y relacionada

CRUD: Acrónimo de Create, Read, Update y Delete (Insertar, Consultar, Modificar y Eliminar)

Paquete: espacio creado para organizar la información y / o conjunto de clases del sistema

Modelo: paquete o capa donde se administran los datos (clase principal)

Vista: todo lo relacionado con las tareas que ve el usuario

Control: paquete que representa la lógica del negocio

Conexión: archivo o clase que estable las características que permiten la comunicación entre una forma y un espacio de almacenamiento

Pool de Conexión: herramienta de conexión con múltiples posibilidades, no limita los archivos ni depende de la compilación del proyecto

MySQL: Administrador de BD

XML: Lenguaje muy común con etiquetas "personalizadas" que permiten la comunicación o administración de datos

Jasper: aplicativo que permite la creación de reportes personalizados

Etiquetas: conjunto de elementos preestablecido para la creación de la documentación de un aplicativo

Compilación: creación de la documentación en un proyecto con formato HTML, ideal para el seguimiento paso a paso de los componentes lógicos de un aplicativo

OBJETIVO GENERAL

Crear habilidades de almacenamiento de la información en repositorios adicionales a los vistos en semestres previos, permitiendo un panorama de opciones de gran utilidad y de gran expansión.

OBJETIVOS ESPECÍFICOS

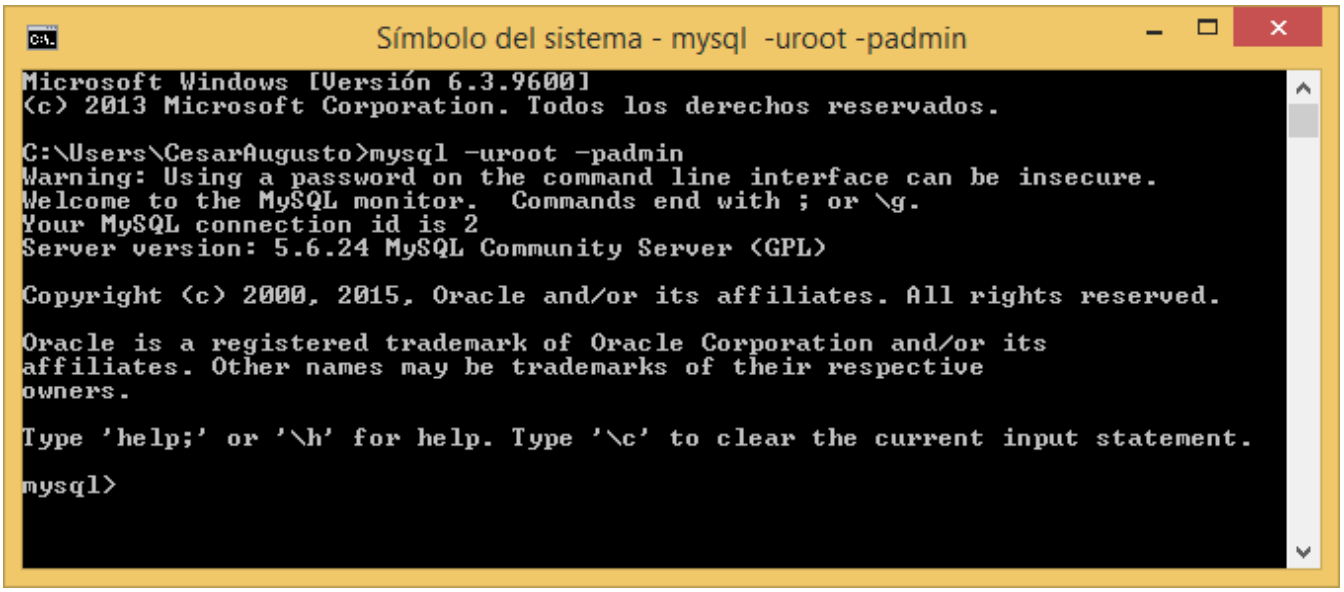
- Desarrollar adecuadamente los procesos del **CRUD** mediante **Bases de Datos**, con todos los requerimientos necesarios.
- Complementar el desarrollo de un aplicativo típico con herramientas como informes para el usuario final y documentación para el desarrollador.

4.2 TEMA 1 APLICACIÓN CRUD

Durante mucho tiempo se han realizado aplicaciones simples con manejo de la información en la memoria (**RAM**), el cual es un aspecto fundamental para el inicio de las primeras aplicaciones permitiendo hacer pruebas, pero todo de manera temporal. Dentro de la evolución de este tema se encontrará el manejo de archivos, opción muy útil cuando se desea almacenar la información de forma permanente, este proceso toma el nombre de **CRUD** por sus siglas en inglés (Create, Read, Update and Delete) Crear, Obtener, Actualizar y Borrar, pero para este nivel se tomará el camino de las **Bases de Datos (BD)**, que funciona de una forma similar a los archivos aunque mucho más estructurado.

Nuestra herramienta de trabajo para las **BD es MySQL**, aunque existe una gran variedad de herramientas que realizan tareas similares, **MySQL** es una herramienta muy sencilla de manejar, con gran alcance, esta utilidad se podrán descargar desde el sitio .

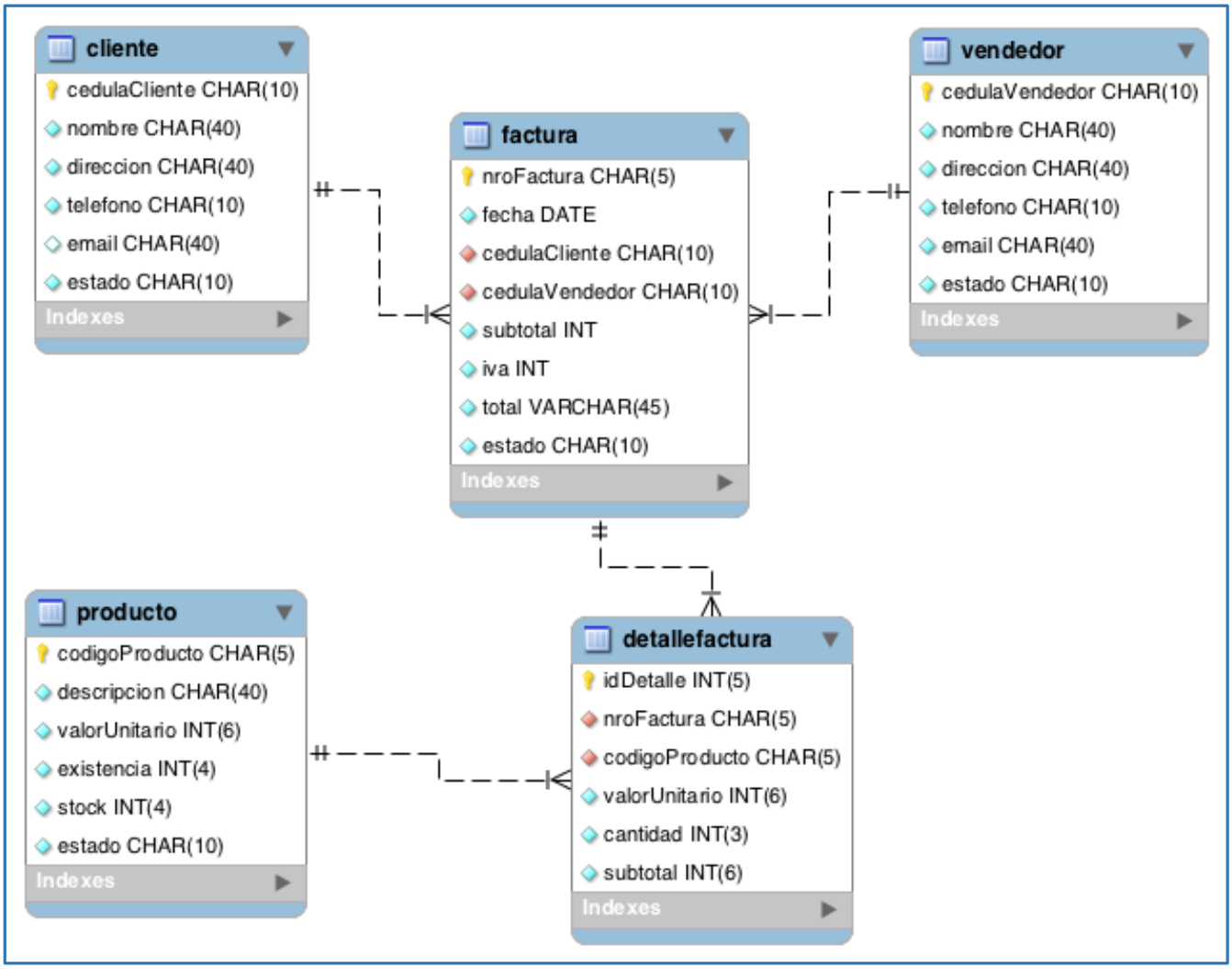
Después de la descarga e instalación y su posterior ingreso encontramos una **consola** como la siguiente.



Autoria Propia

Este es el ambiente típico de trabajo, podrían igualmente utilizar la herramienta **phpmyadmin** o herramientas comunes como **mysql-workbench** o **mysql query Browser** entre muchas otras, y nos permitirán realizar las **operaciones esenciales** desde un ambiente gráfico.

Lo primero que se creara para este propósito es un BD y sus respectivas tablas, generando un **MER (Modelo Entidad Relación)** como el siguiente.

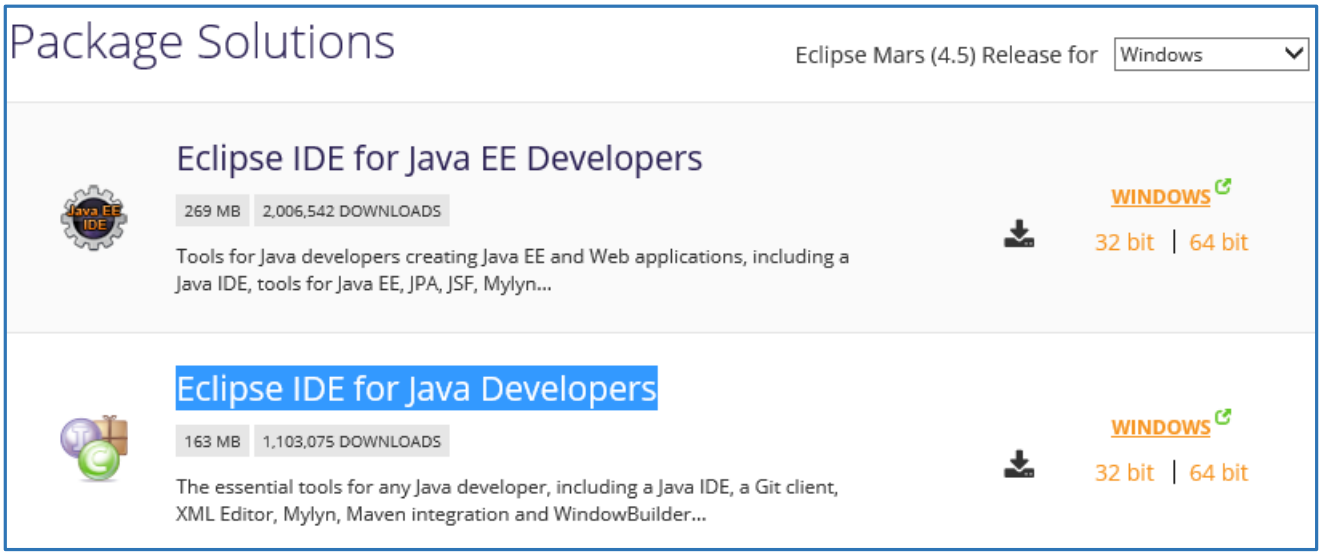


Autoría Propia

Con este **MER** podremos trabajar esta primera unidad relacionada con el CRUD, para tal efecto aplicaremos un concepto adiciona del BD como son los **procedimientos almacenados**, esta herramienta nos hará el proceso un poco más simplificado en la codificación a utilizar, los procedimientos que se van a utilizar son listar, consultar, modificar, eliminar e insertar

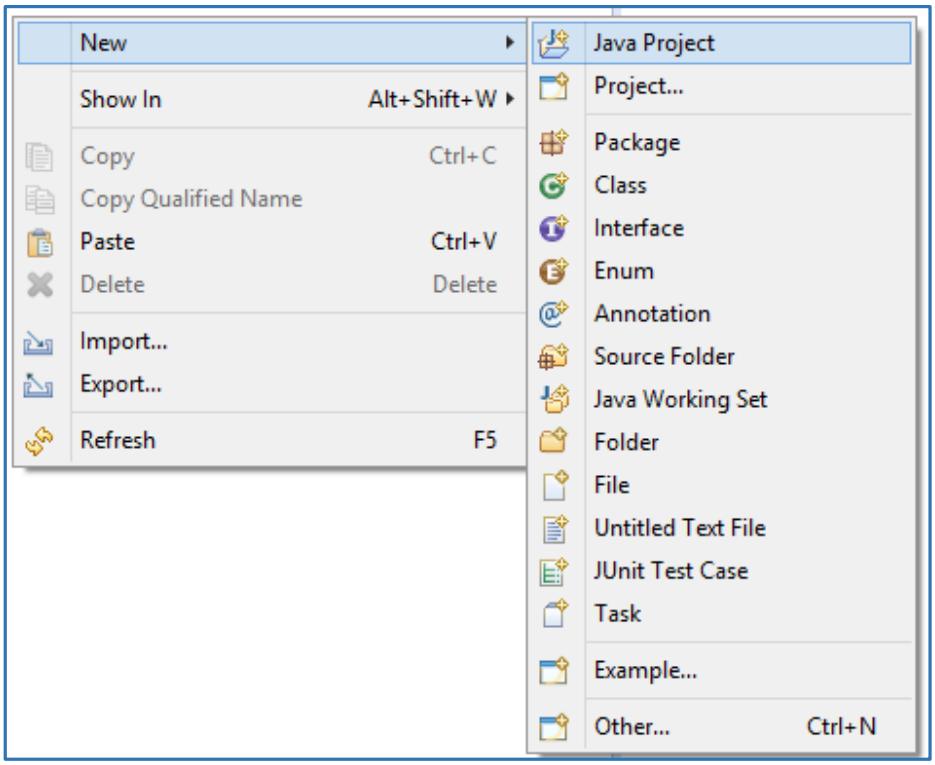
PROYECTO

Utilizando el **IDE (Ambiente de Desarrollo Integrado)** de su preferencia, sea este **Eclipse, NetBeans, JDevelopert** entre muchos otros, este proyecto particularmente se desarrollará mediante **Eclipse SE**, este se podrá descargar del sitio **www.eclipse.org**.



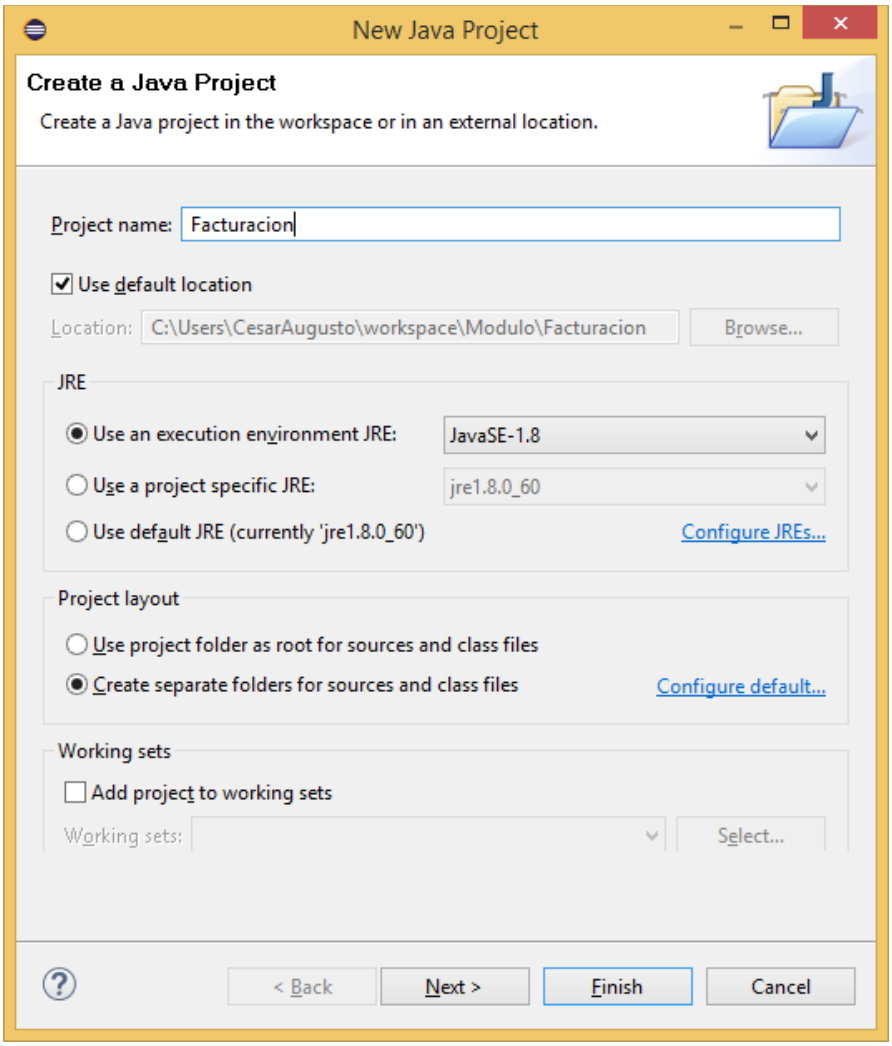
CREACIÓN DEL PROYECTO FACTURACIÓN

Luego de ingresar al **IDE**, con el botón emergente en el área izquierda de la pantalla (Package Explorer), se selecciona new / Java Project



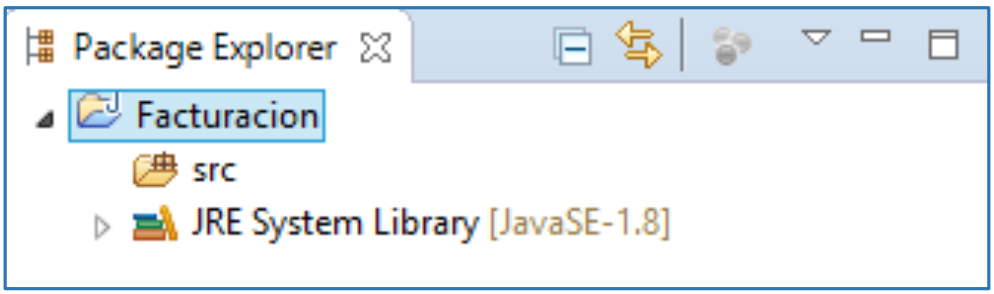
Autoría Propia

Nombre del Proyecto Facturación



Autoria Propia

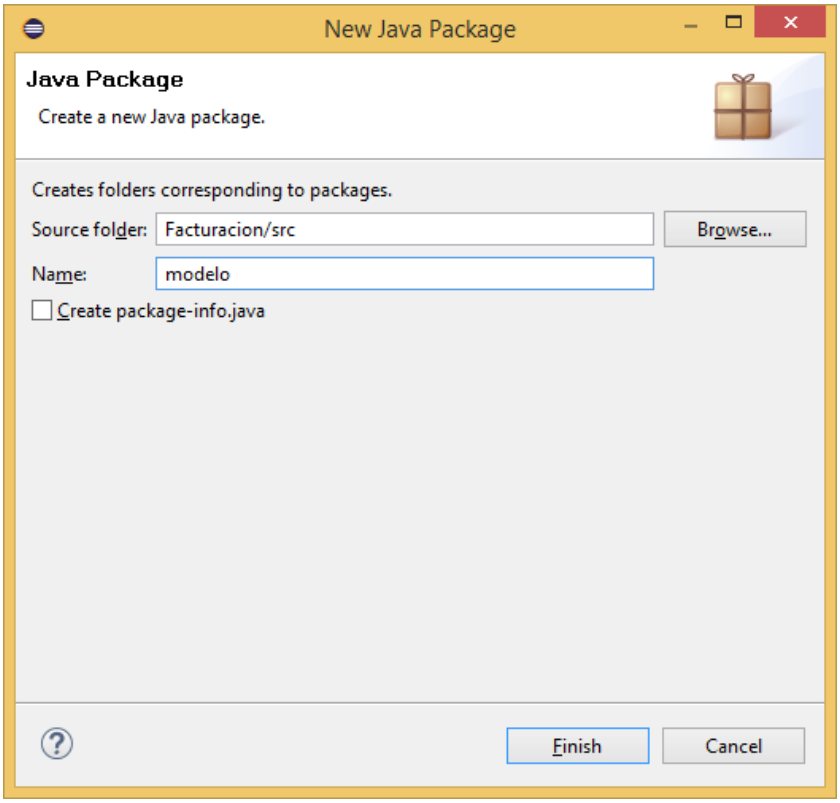
Se dejan las opciones de configuración por defecto y se selecciona el **botón Finish** y se obtiene el siguiente resultado



Autoria Propia

> En la opción src irán los archivos y / o paquetes del proyecto, para una mejor clasificación de los archivos, el propósito de estos es crearan 4 paquetes con los nombres modelo, vista, control y utilidades.

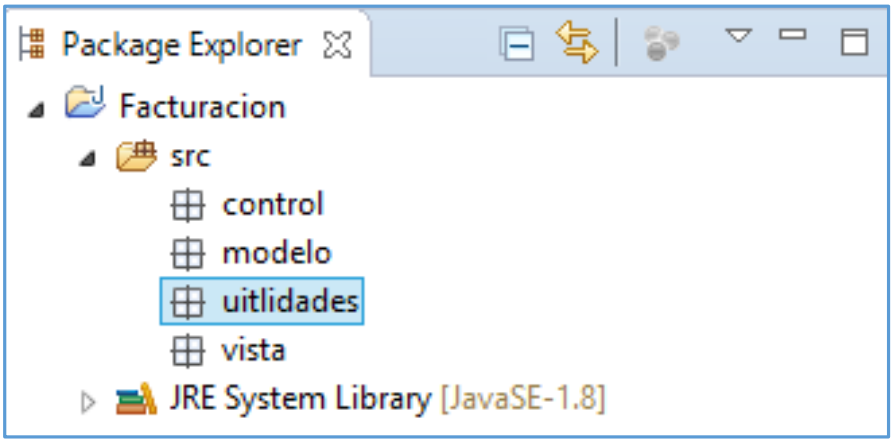
Con el botón emergente sobre el nombre del proyecto se selecciona **New / Package**



Autoria Propia

Luego el botón **Finish** para terminar la creación del paquete, este proceso se realiza para cada paquete especifico.

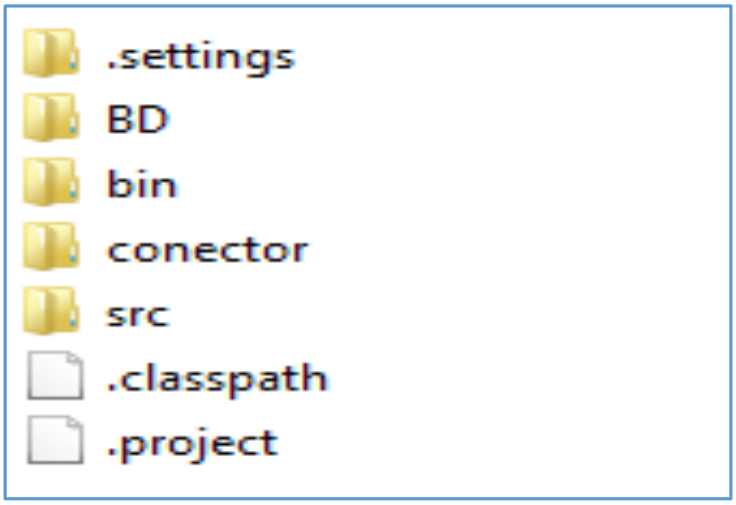
El resultado es el siguiente



Autoria Propia

CONECTOR DE LA BD

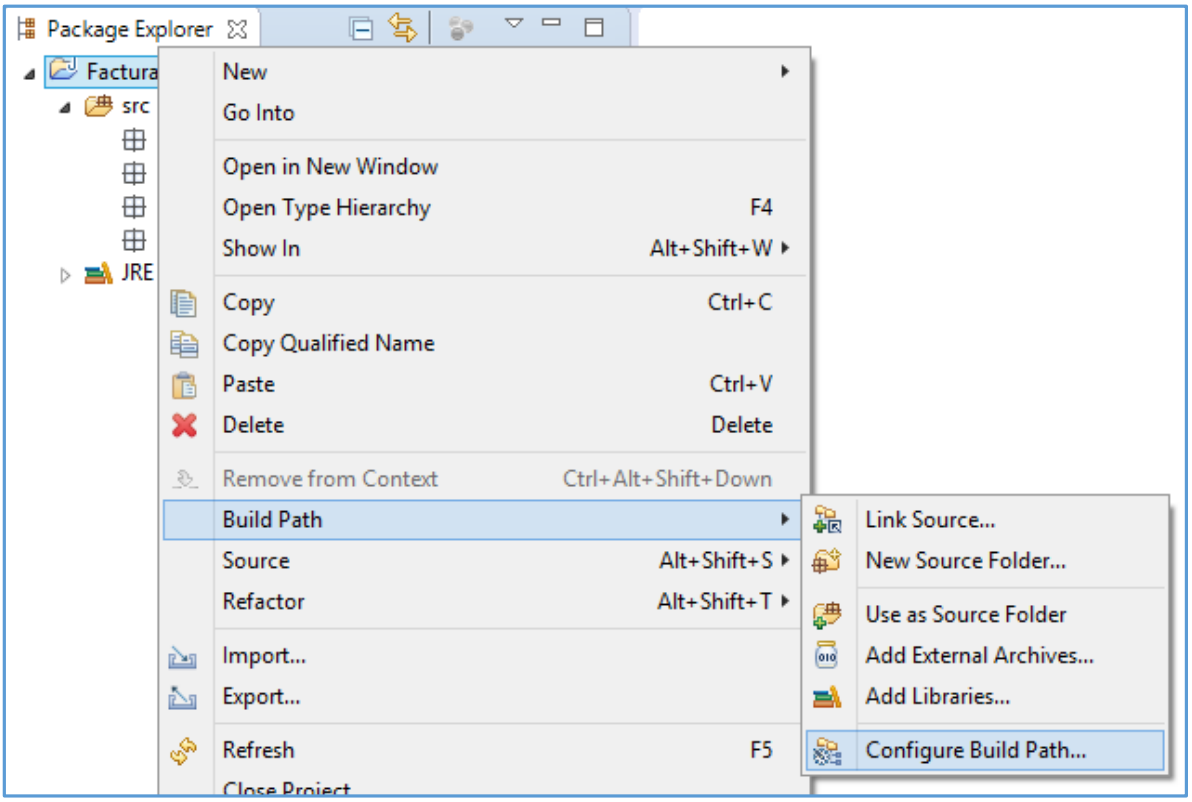
Para establecer un proyecto con **BD** requerimos de un archivo (**librería**) externo que podemos descargar del sitio , este archivo tiene por extensión .jar y su nombre es **mysql-connector-java-5.1.36** (cambia la ultima parte según la versión), este archivo se puede descargar compreso o de instalación, después de haberlo realizado, descomprima dicho archivo y copie el archivo mysql-connector-java-5.1.36-bin.jar en su proyecto para un trasporte mas simplificado (no tiene que ser siempre esta ruta), cree en el proyecto una carpeta con el nombre conector o librería y una carpeta con el nombre de BD para mayor control de los elementos:



Autoria Propia

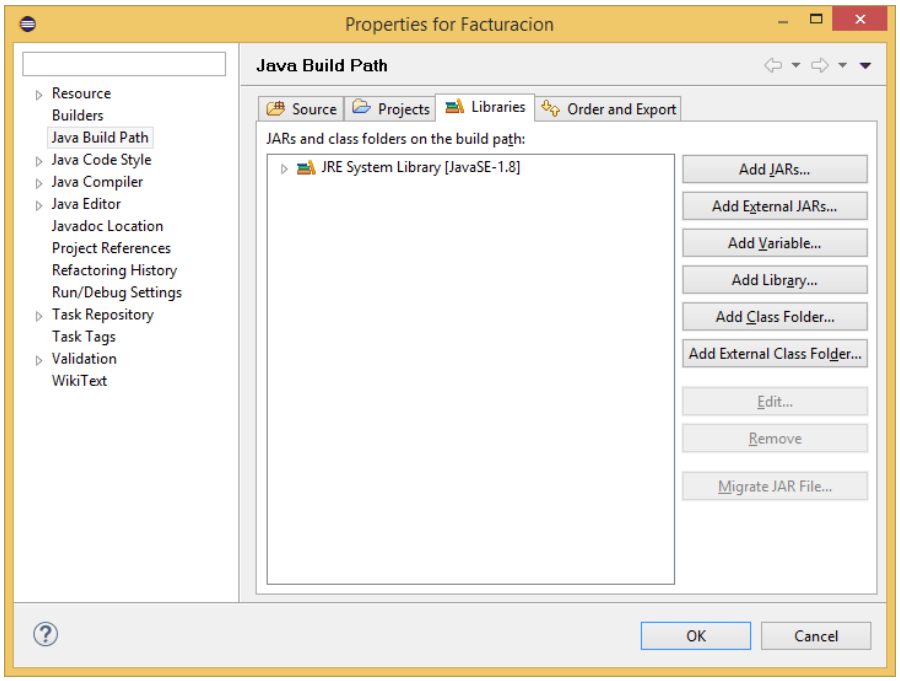
Regrese al proyecto, para la vinculación de este archivo **.jar** con el proyecto de facturación se aplica lo siguiente

Botón emergente sobre el proyecto

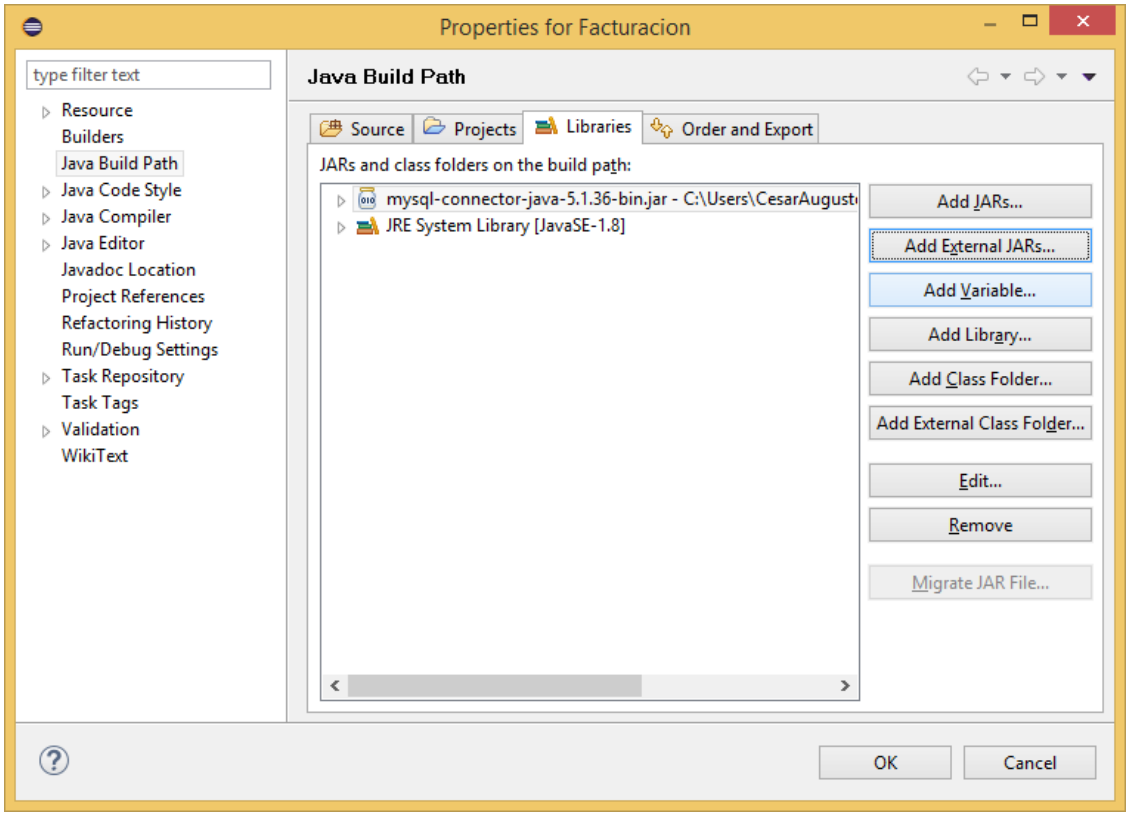


Autoría Propia

En la ventana resultante seleccione el botón **Add External JARs…** y busque el archivo **mysql-connector-java-5.1.36-bin.jar** dentro de la carpeta creada recientemente



Autoria Propia



Autoria Propia

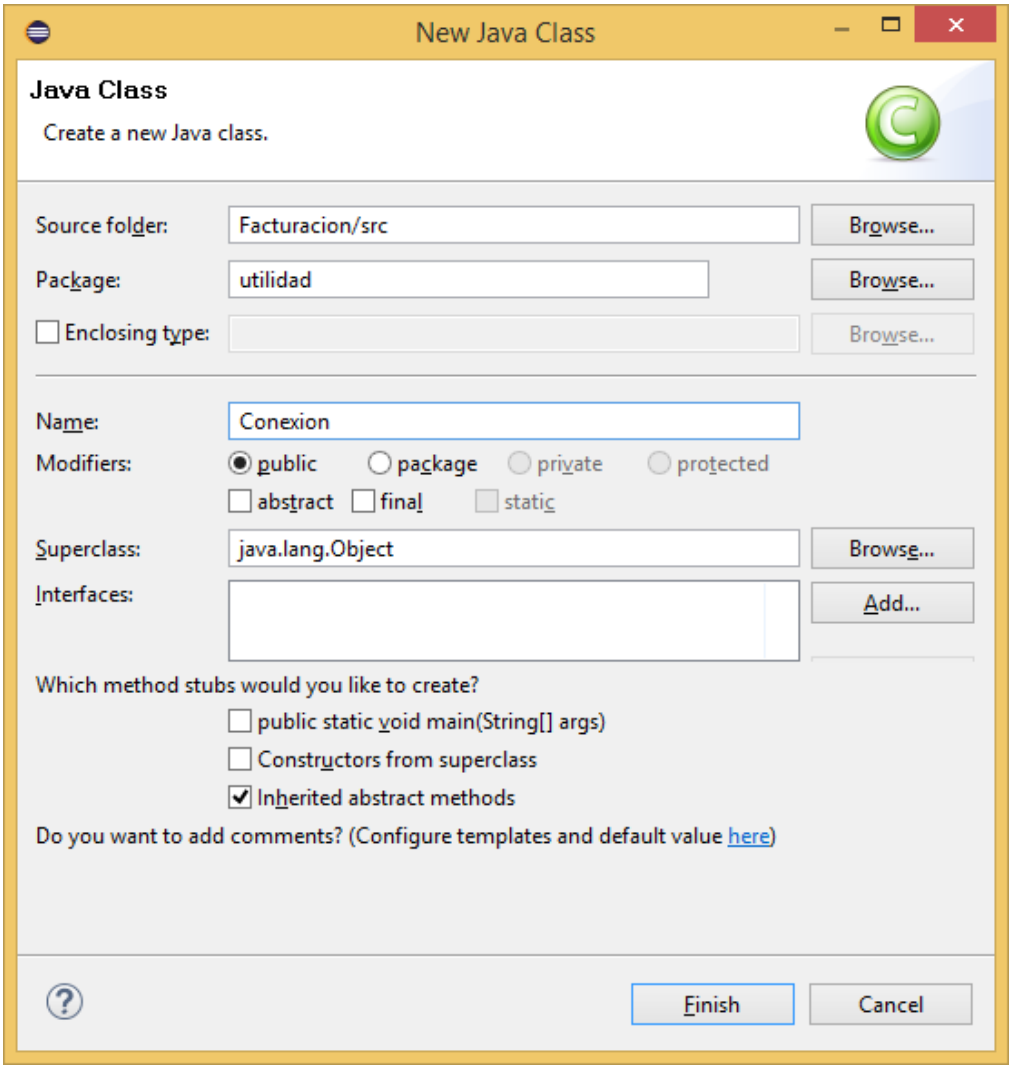
Seleccione el **botón OK**, con este proceso el sistema ya tiene el conector vinculado.

ARCHIVO DE CONEXIÓN A BD

Uno de los procesos fundamentales dentro del manejo de un **CRUD** con **BD** es el archivo o proceso de **conexión**, este nos establece un vínculo directo entre la BD y el proyecto, el proceso inicial ira cambiando con el paso del tiempo, será un proceso muy elemental, básico y con muchas limitantes, pero es funcional, a medida que se avance en el tema se trabajara en la solución de estos posibles problemas mediante el **Pool de Conexiones**.

CREACIÓN DE UN ARCHIVO JAVA PARA CONEXIÓN

Ubicados en el paquete Utilidades y mediante el botón emergente, selecciona New / Class



Autoria Propia

Este primer archivo no es definitivo, servirá como prueba de trabajo y verificación de funcionamiento

```
package utilidad;

import java.sql.Connection;

public class Conexion {

    static Connection con = null;

    public static Connection getConnection () {

        try {

            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection("jdbc:mysql://localhost/facturacion", "root", "admin");
        }
        catch (SQLException se) {

            se.printStackTrace();
            throw new RuntimeException ("Error al crear la Conexion");
        }
        catch (ClassNotFoundException ce) {

            ce.printStackTrace();
            throw new RuntimeException ("Error al crear la Conexion");
        }
        return con;
    }
}
```

El archivo inicial para pruebas se conformaría como el caso anterior, se tiene una clase típica Conexión, existe una clase Connection, esta clase contiene los estándares y las formas apropiadas para establecer una conexión a la BD, un DriverManager, contiene el protocolo que especifica el servidor local, se conforma por los siguientes parámetros

jdbc: conectividad de BD de Java

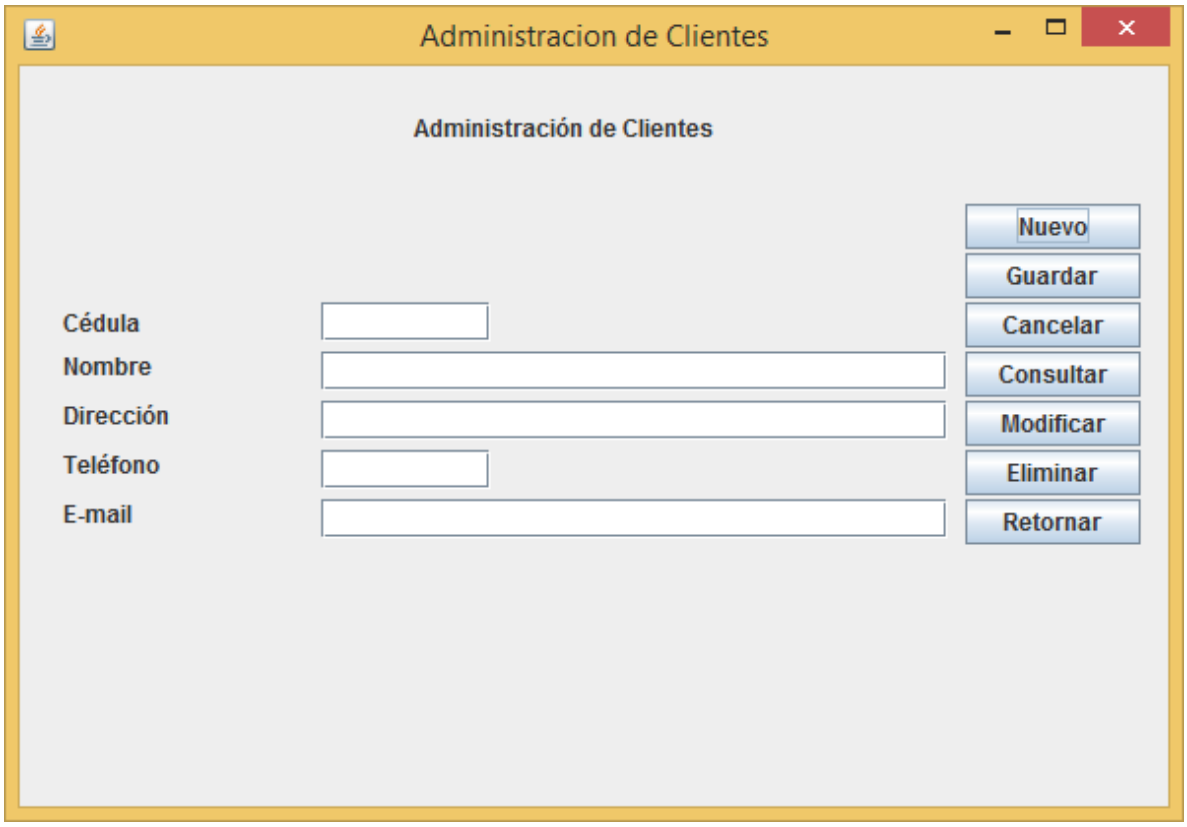
mysql://localhost: especifica el servidor local sobre el que se va a realizar la operación de la BD

Facturación: representa la BD de trabajo

root: usuario por defecto de mysql, este puede ser cambiado o creado

admin: contraseña del sistema de BD de MySQL, puede ser cambiada o creada

Con este proceso terminado, se procede a la construcción de la primera interfaz gráfica, que contendrá las opciones necesarias para representar la tabla de cliente según el MER, este proceso se creara dentro del paquete vista y tendrá el nombre FrmCliente, se trabajara con un JFrame.

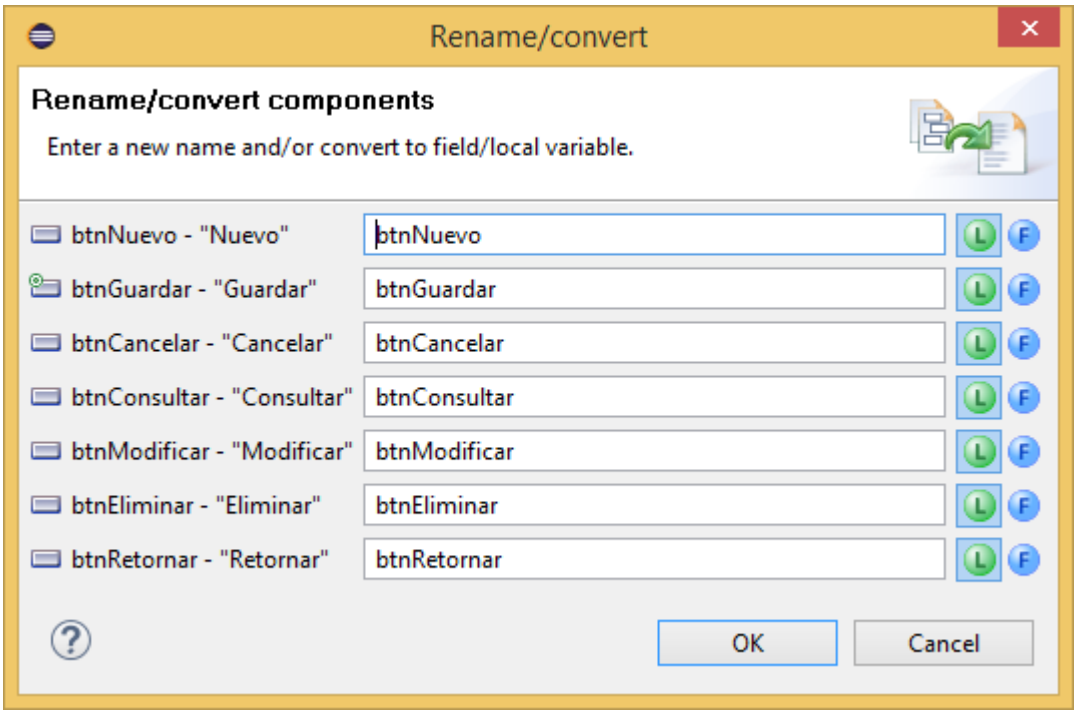


Autoria Propia

Este formulario inicial de trabajo deberá estar completamente validado en sus cajas de texto permitiendo el control de los datos, cedula debe de ser un valor numérico, de máximo 10 cifras, no deberá estar vacío, de cumplirse estas condiciones y presionar la tecla **enter** llevará el foco al campo de nombre, cada campo debe cumplir condiciones similares según los requerimientos que se establezcan al inicio del aplicativo, el ultimo campo llevará el foco al botón de Guardar.

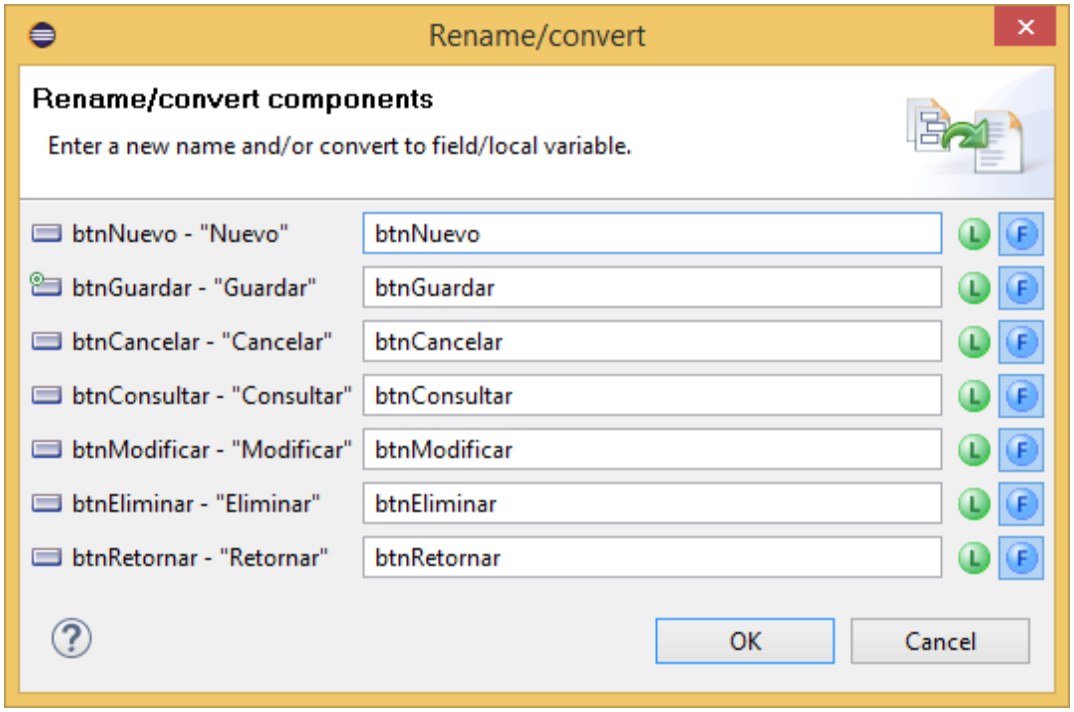
CAMBIOS EN LOS BOTONES DE COMANDO

Los botones de comando están definidos como variables y para el propósito de mayor alcance se requiere que estén como campos, el cambio de este alcance es muy simple, solo seleccionamos los botones, con el menú emergente selecciona **Rename**.



Autoria propia

Cambie L (local) por F (fields)



Autoria Propia

CREACIÓN DE MÉTODOS FUNCIONALES

Existe un sin número de opciones para el buen funcionamiento de los procesos, uno de ellos es la construcción de los métodos que permitan la adecuada administración de las tareas, dentro de ellos tendremos un método que **limpie las cajas de texto**, de **habilite/deshabilite** los botones de comando, etc., en el **Source** o código de este formulario, ubicados al final del archivo se crearan los siguientes.

MÉTODO LIMPIAR

Diseñado para las cajas de texto

```
private static void limpiar () {

    txtCedula.setText("");
    txtNombre.setText("");
    txtDireccion.setText("");
    txtTelefono.setText("");
    txtEmail.setText("");
}
```

Autoria Propia

MÉTODO DESHABILITAR

Bloquea las cajas de texto al inicio del aplicativo

```
private static void deshabilitar () {

    txtCedula.setEditable(false);
    txtNombre.setEditable(false);
    txtDireccion.setEditable(false);
    txtTelefono.setEditable(false);
    txtEmail.setEditable(false);
}
```

Autoria Propia

MÉTODOS DE ACTIVAR Y DESACTIVAR

Controla los botones según la operación que se este realizando

```
private static void activar () {

    btnNuevo.setEnabled(true);
    btnGuardar.setEnabled(false);
    btnCancelar.setEnabled(false);
    btnConsultar.setEnabled(true);
    btnModificar.setEnabled(true);
    btnEliminar.setEnabled(true);
    btnRetornar.setEnabled(true);
}

private static void desactivar () {

    btnNuevo.setEnabled(false);
    btnGuardar.setEnabled(false);
    btnCancelar.setEnabled(true);
    btnConsultar.setEnabled(false);
    btnModificar.setEnabled(false);
    btnEliminar.setEnabled(false);
    btnRetornar.setEnabled(false);
}
```

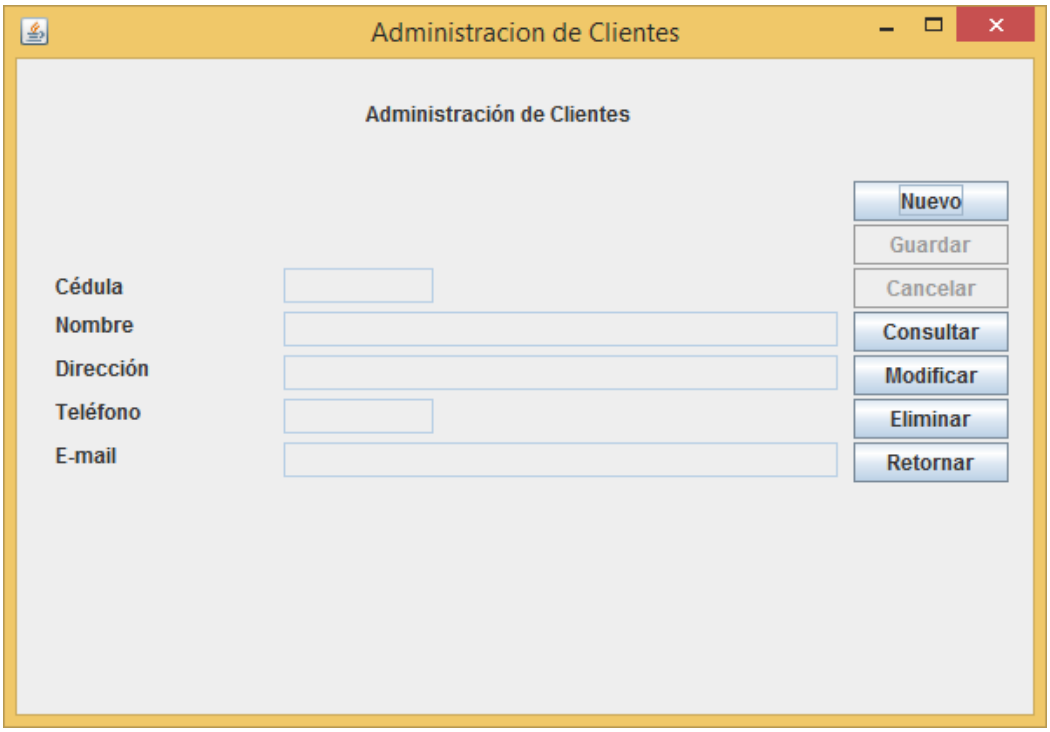
Autoria Propia

LLAMADO DE LOS MÉTODOS

```
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                FrmCliente frame = new FrmCliente();
                frame.setVisible(true);
                limpiar ();
                activar ();
                deshabilitar ();

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

Autoria Propia

Así se verá el proyecto con estos cambios



Autoria Propia

CREACIÓN DE VARIABLES ADICIONALES

```
public class FrmCliente extends JFrame {

    private JPanel contentPane;
    private static JTextField txtCedula;
    private static JTextField txtNombre;
    private static JTextField txtDireccion;
    private static JTextField txtTelefono;
    private static JTextField txtEmail;
    private static JButton btnNuevo;
    private static JButton btnGuardar;
    private static JButton btnCancelar;
    private static JButton btnConsultar;
    private static JButton btnModificar;
    private static JButton btnEliminar;
    private static JButton btnRetornar;
    private int sw = 0;
    private int opc = 0;
```

Autoria Propia

En el inicio del archivo, se encontrarán las definiciones de las cajas de texto, botones y demás elementos que comprendan el formulario de trabajo, al final se declararan 2 variables complementarias, sw será una bandera de encendido y apagado, mediante esta se determinara si existe o no un elemento consultado, la variable opc se utilizara como clasificación según la operación a realizar, sea esta una inserción, consulta, modificación o eliminación.

ASIGNACIÓN DE LOS BOTONES DE COMANDO

La siguiente asignación de código a cada botón permitirá el buen desempeño de las operaciones, para realizar esta tarea basta con presionar **doble clic** sobre el botón mencionado.

BOTÓN NUEVO

```
limpiar ();
desactivar ();
deshabilitar ();
opc = 1;
txtCedula.setEditable(true);
txtCedula.requestFocus();
```

Autoria Propia

BOTÓN CANCELAR

```
limpiar ();
activar();
deshabilitar ();
```

Autoria Propia

BOTÓN GUARDAR

```
deshabilitar ();
activar ();
```

Autoria Propia

- BOTÓN CONSULTAR

```
limpiar ();
deshabilitar ();
opc = 2;
txtCedula.setEditable(true);
txtCedula.requestFocus();
```

Autoria Propia

- BOTÓN MODIFICAR

```
limpiar ();
desactivar ();
deshabilitar ();
opc = 3;
txtCedula.setEditable(true);
txtCedula.requestFocus();
```

Autoria Propia

- BOTÓN ELIMINAR

```
limpiar ();
deshabilitar ();
opc = 4;
txtCedula.setEditable(true);
txtCedula.requestFocus();
```

Autoria Propia

- BOTÓN SALIR

```
dispose ();
```

Autoria Propia

> En cada botón se aplican una serie de métodos que permiten desde limpiar las cajas de texto, hasta bloquearlas, esto con el fin de evitar posibles ingresos involuntario de información que pueden afectar las tareas principales del formulario, estos procesos acompañados de las validaciones se vuelven fundamentales para unas buenas practicas del control de los datos.

La variable **opc** es de vital importancia porque con ella se determinar la función a realizar, 1 para registros nuevos, 2 para consultar información, 3 para modificar y 4 para la eliminación, esta eliminación se realizará de manera lógica mas no física, lógica indica que el registro permanecerá en la tabla cliente, pero no será visible a el usuario, permitirá el control de históricos de la BD.

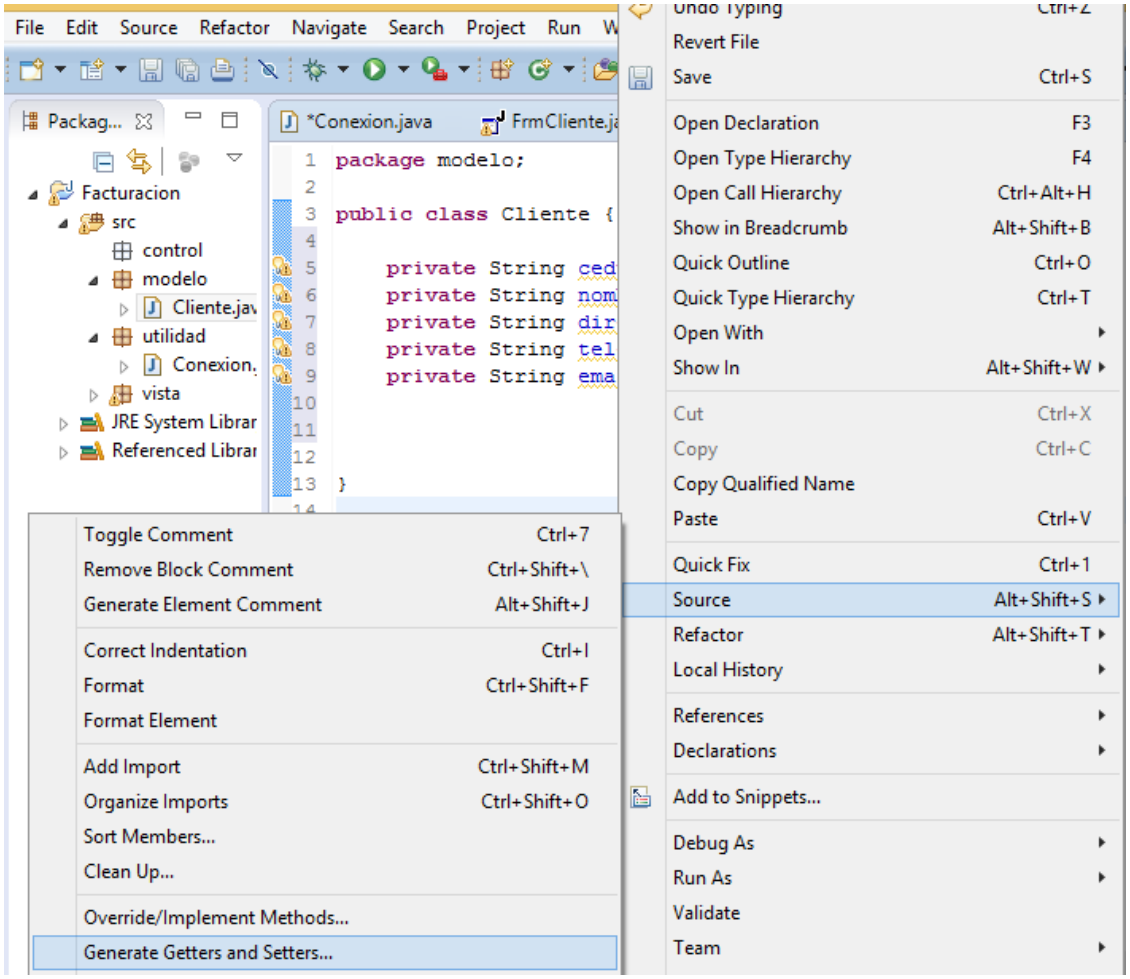
CREACIÓN DE LA CLASE PRINCIPAL (BEANS)

La clase principal del formulario inicial es muy simple pero de gran importancia, se definen las **variables** (campos) a trabajar y se generaran los **getters** y **setters**, estos serán los contenedores de la información en distintas etapas, los datos serán enviados y / o tomados de ellos, evitando así acceder a una clase de formulario o a otro proceso que pueda vulnerar la seguridad del aplicativo, para este caso se crea una clase en el paquete modelo, con el nombre Cliente, quedando de la siguiente manera

```
package modelo;

public class Cliente {

    private String cedula;
    private String nombre;
    private String direccion;
    private String telefono;
    private String email;



}
```

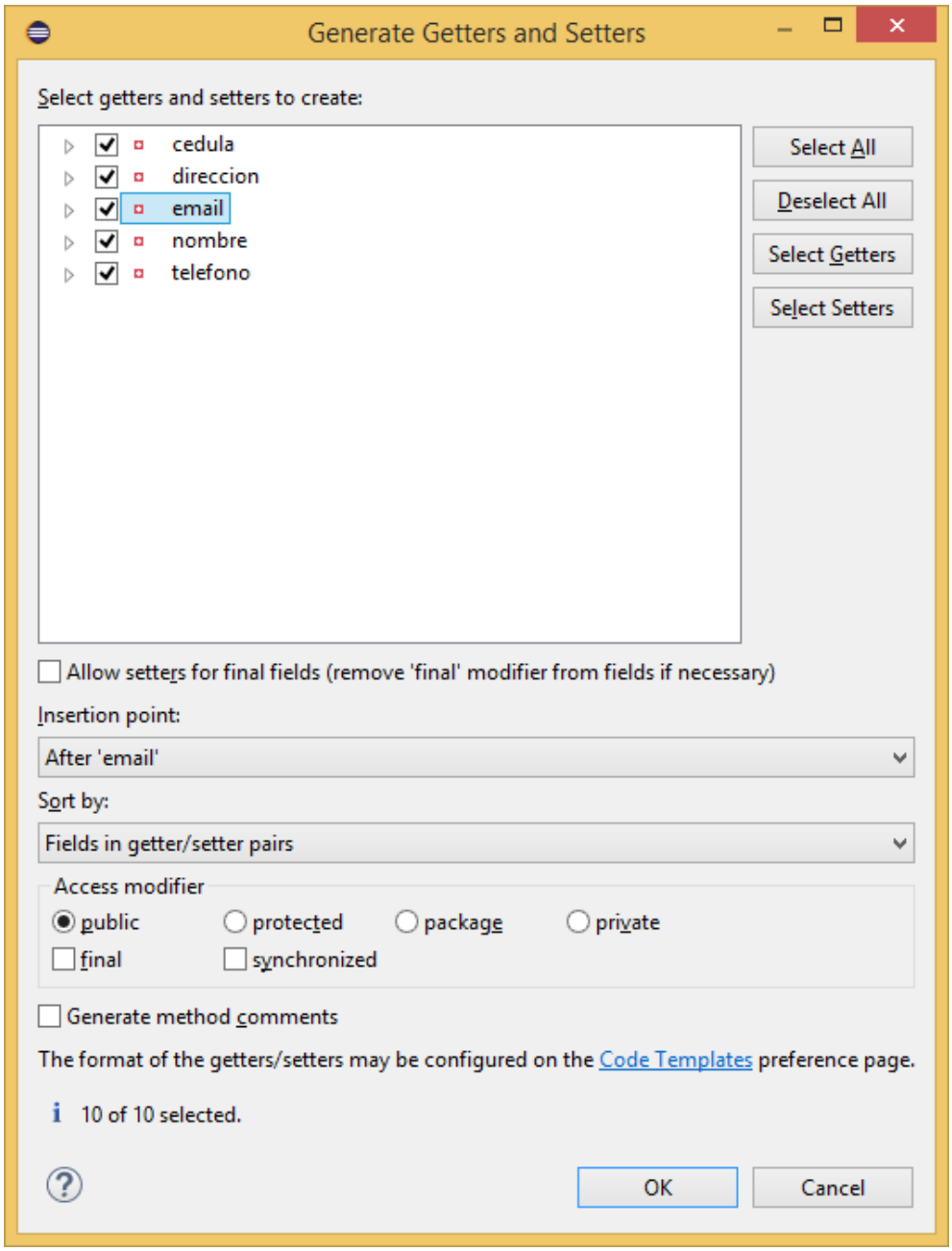
Autoria Propia

Posterior a este proceso se generan los **getters** y **setters**, para este proceso se elije el botón emergente sobre cualquiera de los campos creados



Autoria Propia

Se seleccionan los campos con los que se trabajara, no es obligatorio seleccionarlos todos, solo los que tengan a bien ser utilizados o actualizados permanentemente.



Autoria Propia

Y luego el botón OK

```
package modelo;


public class Cliente {
```

```
private String cedula;

private String nombre;

private String direccion;

private String telefono;

private String email;


public String getCedula() {

	return cedula;

}

public void setCedula(String cedula) {

	this.cedula = cedula;

}

public String getNombre() {

	return nombre;

}

public void setNombre(String nombre) {

	this.nombre = nombre;

}

public String getDireccion() {

	return direccion;

}

public void setDireccion(String direccion) {

	this.direccion = direccion;

}

public String getTelefono() {

	return telefono;
```

```
	}
	public void setTelefono(String telefono) {
		this.telefono = telefono;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
}
```

Quedando de este modo, los sets (**setters**) actualizaran la información, los get (**getters**) la retornaran a quien la solicite.

Creación del patrón de diseño DAO (Data Access Object), los patrones de diseño, un patrón de diseño sugiere una solución a un problema, con un formato generalmente aceptado como un estándar, en él se ubicarán la lógica del negocio, los métodos del proceso.

PROCEDIMIENTOS ALMACENADOS

Los procedimientos son herramientas de las BD que permiten realizar tareas de forma más flexible, sin exponer tanto el código y con la seguridad que pueda brindar la BD, permite las instrucciones más comunes del SQL como son las inserciones, consultas, modificaciones y eliminaciones, a continuación los 5 procedimientos a utilizar en este primer formulario.

VERIFICACIÓN

Se utiliza para comprobar que el registro se encuentre o no dentro de la tabla

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `verificarCliente` $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `verificarCliente`(in _cedulaCliente char(10))
BEGIN
  select cedulaCliente from cliente where cedulaCliente = _cedulaCliente and estado = 'activo';
END $$

DELIMITER ;
```

Autoria Propia

INSERCIÓN

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `insertarCliente` $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `insertarCliente`(in _cedulaCliente char (10), in _nombre char (40),
                                    in _direccion char (40), in _telefono char (10), in _email char(40), in _estado char(10))
BEGIN
  insert into cliente (cedulaCliente, nombre, direccion, telefono, email, estado) values
                      (_cedulaCliente, _nombre, _direccion, _telefono, _email, _estado);
END $$

DELIMITER ;
```

Autoria Propia

CONSULTA

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `consultarCliente` $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `consultarCliente`(in _cedulaCliente char (10))
BEGIN
  select cedulaCliente, nombre, direccion, telefono, email from cliente where cedulaCliente = _cedulaCliente and estado ='activo';
END $$

DELIMITER ;
```

Autoria Propia

MODIFICACIÓN

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `modificarCliente` $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `modificarCliente`(in _cedulaCliente char (10), in _nombre char (40),
                                                               in _direccion char (40), in _telefono char (10), in _email char(40))
BEGIN
  update cliente set nombre = _nombre, direccion =_direccion, telefono = _telefono, email = _email
        where cedulaCliente = _cedulaCliente;
END $$

DELIMITER ;
```

Autoria Propia

ELIMINACIÓN

La eliminación no se aplica con la sentencia **delete** como es tradicional, sino con **update**, se está aplicando una eliminación lógica, que permite conservar los datos para consultas históricas.

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `eliminarCliente` $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `eliminarCliente`(in _cedulaCliente char(10))
BEGIN
  update cliente set estado = 'inactivo' where cedulaCliente = _cedulaCliente;
END $$

DELIMITER ;
```

Autoria Propia

Existe un archivo adicional en este modelo llamado **ClienteDAO (Data Access Object),** el modelo (patrón de diseño) **DAO** brinda la posibilidad de almacenar toda la lógica del negocio, para este caso particular los métodos de insertar, consultar, modificar, eliminar y verificar, este archivo se ubica en el paquete control.

Se ubicarán en el mismo orden de los procedimientos almacenados para mayor claridad.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.swing.JOptionPane;

import modelo.Cliente;
import utilidad.Conexion;

public class ClienteDAO {

    static int rta = 0;
    static Connection cnn = Conexion.getConnection();
    static Cliente cliente = new Cliente ();
```

Autoria Propia

en la parte inicial de la clase se encuentra una variable **rta** iniciada en cero (0), esta variable será una **sw** que retornara si existe o no un valor consultado, la clase **Connection** permite la asignación a una variable **cnn** de del método **getConnection** de la clase Conexión, por ultimo se instancia la clase Cliente que contiene los getters y setters.

VERIFICAR

Este método verificar contiene la sentencia **PreparedStatement**, esta instrucción permite tener una sentencia SQL de fácil lectura, al final de la sentencia podríamos colocar una instrucción tradicional o como en este caso el llamado a un procedimiento almacenado, el signo de interrogación (?) indica el parámetro con el que va trabajar, en la siguiente línea se “traduce” ese signo a que valor corresponde.

En la tercera línea se encuentra **ResultSet**, esta instrucción permite el almacenamiento de una consulta SQL en memoria.

En la misma línea se encuentra la instrucción **executeQuery**, acá se este ejecutando la instrucción **SQL select** con los parámetros que se hayan especificado, otra instrucción muy común es **executeUpdate** que aplicara para las demás sentencias.

La sentencia **rta = rs.next() ? 1 : 0;** nos permitirá determinar si el registro consultado existe (1) o no en la tabla (0), y al final retornara este resultado al formulario donde se precederá con las siguientes tareas.

La referencia **registroCliente.setStrin**g determina el tipo de campo que esta llegando (carácter), si fuera un tipo entero se representaría con **setInt**, contiene 2 parámetros, el primero es la posición o el orden de los parámetros y el segundo el valor o referencia.

```
public static int verificar (String cedula){

    try {

        PreparedStatement registroCliente = cnn.prepareStatement("call verificarCliente (?)");
        registroCliente.setString(1,  cedula);
        ResultSet rs = registroCliente.executeQuery();
        rta = rs.next() ? 1 : 0;
    }
    catch (SQLException sqle) {

        sqle.printStackTrace ();
    }

    return rta;

}
```

Autoria propia

INSERTAR

La inserción no retorna ningún valor, pero recibe la referencia de la clase cliente donde se podrán actualizar y enviar los valores solicitados.

En la instrucción **setString** se encuentran los dos parámetros mencionados anteriormente, el orden de los parámetros y en **cliente. getCedulaCliente (),** esta opción **getCedulaCliente ()** está tomando el valor de la clase Cliente.

Al final del try aparece el **executeUpdate** que procede a ejecutar la sentencia **SQL**.

```
public static void insertar (Cliente cliente){

    try {

        PreparedStatement registroCliente = cnn.prepareStatement("call insertarCliente (?, ?, ?, ?, ?, ?)");
        registroCliente.setString(1, cliente.getCedulaCliente());
        registroCliente.setString(2, cliente.getNombre());
        registroCliente.setString(3, cliente.getDireccion());
        registroCliente.setString(4, cliente.getTelefono());
        registroCliente.setString(5, cliente.getEmail());
        registroCliente.setString(6, "activo");
        registroCliente.executeUpdate ();
        JOptionPane.showMessageDialog(null, "Registro Almacenado");
    }
    catch (SQLException sqle) {

        sqle.printStackTrace ();
    }
}
```

Autoría Propia

CONSULTAR

Se aplica la instrucción **setCedulaCliente**, esta instrucción actualiza la información en la clase Cliente.

rs.next (), si el sistema ingresa en esta instrucción indica que existen datos para ser consultados.

```
public static Cliente consultar (String cedula) {

    try {

        PreparedStatement registroCliente = cnn.prepareStatement("call consultarCliente (?)");
        registroCliente.setString(1, cedula);
        ResultSet rs = registroCliente.executeQuery();
        if (rs.next()) {

            cliente.setCedulaCliente(rs.getString("cedulaCliente"));
            cliente.setNombre(rs.getString("nombre"));
            cliente.setDireccion(rs.getString("direccion"));
            cliente.setTelefono(rs.getString("telefono"));
            cliente.setEmail(rs.getString("email"));
        }
    }
    catch (SQLException sqle) {

        sqle.printStackTrace ();
    }

    return cliente;

}
```

Autoría Propia

MODIFICAR

```
public static void modificar (Cliente cliente) {

    try {

        PreparedStatement registroCliente = cnn.prepareStatement("call modificarCliente (?, ?, ?, ?, ?)");
        registroCliente.setString(1, cliente.getCedulaCliente());
        registroCliente.setString(2, cliente.getNombre());
        registroCliente.setString(3, cliente.getDireccion());
        registroCliente.setString(4, cliente.getTelefono());
        registroCliente.setString(5, cliente.getEmail());
        registroCliente.executeUpdate ();
        JOptionPane.showMessageDialog(null, "Registro Actualizado");
    }
    catch (SQLException sqle) {

        sqle.printStackTrace ();
    }
}
```

Autoría Propia

ELIMINAR

```
public static void eliminar (String cedula) {

    try {

        PreparedStatement registroCliente = cnn.prepareStatement("call eliminarCliente (?)");
        registroCliente.setString(1, cedula);
        registroCliente.executeUpdate ();
        JOptionPane.showMessageDialog(null, "Registro Eliminado");
    }
    catch (SQLException sqle) {

        sqle.printStackTrace ();
    }
}
```

Autoría Propia

> **Archivo Facade o Fachada, el patrón de diseño, permite la coordinación, el control, la adecuada distribución de los procesos, además de ser un puente de solicitudes y no la solicitud directa a un formulario o a un DAO, este archivo contendrá todos los llamados que sean necesarios para el aplicativo en general, no para cada formulario.**

En el paquete utilidad se ubicará este archivo Facade

```
import control.ClienteDAO;
import modelo.Cliente;

public class Facade {

    static Cliente cliente;
    static ClienteDAO clienteDAO;

    public Facade (){

        cliente = new Cliente ();
        clienteDAO = new ClienteDAO ();
    }
```

Autoria Propia

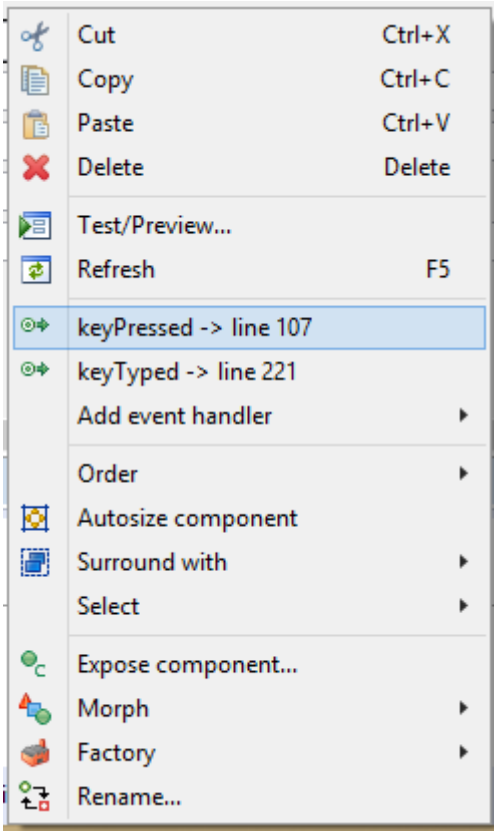
El facade se ubicarán procesos muy cortos asociados a llamados de los métodos del DAO

```
public static int verificarCliente (String cedula) {

    return clienteDAO.verificar(cedula);
}

public static void insertarCliente (Cliente cliente) {

    clienteDAO.insertar (cliente);
}

public static Cliente consultarCliente (String cedula) {

    return clienteDAO.consultar(cedula);
}

public static void modificarCliente (Cliente cliente) {

    clienteDAO.modificar(cliente);
}

public static void eliminarCliente (String cedula) {

    clienteDAO.eliminar(cedula);
}
```

Autoria Propia

Acá se encuentran los 5 procesos según los 5 métodos creados en el DAO, tienen la misma estructura y se invoca en todos los casos el DAO, todas las solicitudes provienen del formulario de trabajo.

Para finalizar este procedo de CRUD queda pendiente la invocación de las tareas desde el formulario de trabajo, para este caso FrmCliente, presionando el botón emergente sobre la cedula (Clave primaria).



Autoria Propia

Se desplegará el código de validación de este campo (todas las cajas de texto deben de cumplir con un estándar de validación mínima.

```
public void keyPressed(KeyEvent arg0) {

    if (arg0.getKeyCode() == KeyEvent.VK_ENTER) {

        if (txtCedula.getText().equals("")) {

            JOptionPane.showMessageDialog(null, "Debe Digitar la Cedula");
            txtCedula.requestFocus();
        }
        else if (txtCedula.getText().length() < 6 || txtCedula.getText().length() > 10) {

            JOptionPane.showMessageDialog(null , "La Cedula es Incorrecta, el rango comprende entre 6 y 10 cifras");
            txtCedula.requestFocus();
        }
        else {
```

Autoria Propia

En este código se esté validando que el campo no este vacío y que cumpla una longitud mínima de 6 cifras y una máxima de 10, en el **else** que queda abierto ira la codificación operacional

```
sw = facade.verificarCliente (txtCedula.getText());
switch (opc) {

    case 1:

        if (sw == 1) {

            JOptionPane.showMessageDialog (null, "la Cedula " + txtCedula.getText() + " ya Existe");
            activar ();
            limpiar ();
            deshabilitar ();
        }
        else {

            txtNombre.setEditable(true);
            txtNombre.requestFocus();
        }
        break;
```

Autoria Propia

La variable **sw** recibe un valor (0 o 1) que determinar si existe o no un registro, este proceso se realiza mediante el archivo **facade** en el método **verificar Cliente**, este proceso a su vez invoca el método **verificar** de la clase **ClienteDAO** que realiza el recorrido dentro del BD y arroja una respuesta.

En el **switch** se evalúa que operación se aplicara (1 nuevo, 2 consultar, 3 modificar y 4 eliminar), dentro de el se evalúa **sw**, para este ejemplo si el resultado es uno (1) indica que la cedula ya existe y no se podrá ingresar nuevamente si es cero (0) indica que el registro no existe y se podrá ingresar un nuevo registro, es por esto que se lleva el foco a la caja de texto de nombre.

```
case 2:

    if (sw == 1) {

        cliente = facade.consultarCliente(txtCedula.getText());
        mostrar (cliente);
        deshabilitar ();
    }
    else {

        JOptionPane.showMessageDialog(null, "NO Existe la Cedula Consultada");
        deshabilitar ();
        limpiar ();
    }
    break;
```

Autoria Propia

```
case 3:
    if (sw == 1) {
        cliente = facade.consultarCliente(txtCedula.getText());
        mostrar (cliente);
        deshabilitar ();
        int respuesta = JOptionPane.showConfirmDialog(null, "Seguro de Modificar?", null, JOptionPane.YES_NO_OPTION);
        if (respuesta == 0) {

            txtCedula.setEditable(false);
            txtNombre.setEditable(true);
            txtNombre.requestFocus();
        }
        else {

            deshabilitar ();
            activar ();
        }
    }
    else {

        JOptionPane.showMessageDialog(null, "No Existe la Cedula");
        activar ();
        limpiar ();
        deshabilitar ();
    }
    break;
```

Autoria Propia

```
    case 4:
        if (sw == 1) {
            cliente = facade.consultarCliente(txtCedula.getText());
            mostrar (cliente);
            deshabilitar ();
            int respuesta = JOptionPane.showConfirmDialog(null, "Seguro de Eliminar?", null, JOptionPane.YES_NO_OPTION);
            if (respuesta == 0) {

                facade.eliminarCliente(txtCedula.getText());
                limpiar ();
                deshabilitar ();
            }
            else {

                deshabilitar ();
                activar ();
            }
        }
        else {
            JOptionPane.showMessageDialog(null, "No Existe la Cedula");
            activar ();
            limpiar ();
            deshabilitar ();
        }
        break;
}
sw = 0;
```

Autoria Propia

BOTÓN GUARDAR

El botón guardar ya tenia una codificación previa, esta codificación será ampliada con el fin de realizar el paso de los datos a modificar y a insertar.

```
cliente.setCedulaCliente(txtCedula.getText());
cliente.setNombre(txtNombre.getText());
cliente.setDireccion(txtDireccion.getText());
cliente.setTelefono(txtTelefono.getText());
cliente.setEmail(txtEmail.getText());

if (opc == 1)

    facade.insertarCliente (cliente);
else

    facade.modificarCliente(cliente);

deshabilitar ();
activar ();
```

Autoria Propia

En el FrmCliente, al final de la codificación falta la creación del método mostrar, este método recibe la información consultada y la lleva a las cajas de texto.

```
private void mostrar (Cliente cliente) {

    txtCedula.setText(cliente.getCedulaCliente());
    txtNombre.setText(cliente.getNombre());
    txtDireccion.setText(cliente.getDireccion());
    txtTelefono.setText(cliente.getTelefono());
    txtEmail.setText(cliente.getEmail());
}
```

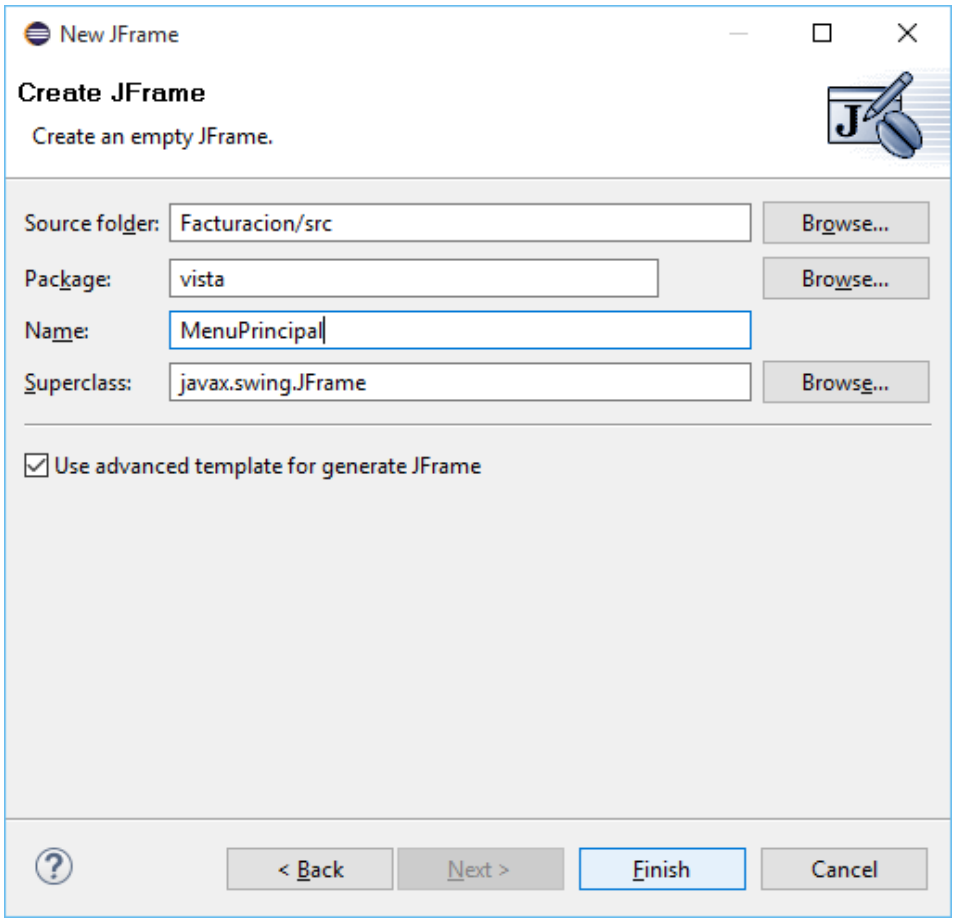
Autoria Propia

Este es el proceso final, el aplicativo esta 100% terminado y se podría poner a prueba.

CREACIÓN DE UN MENÚ

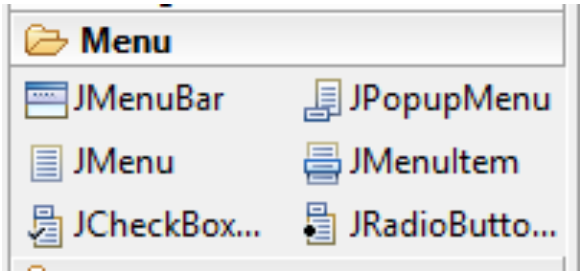
Después de tener el primer CRUD completo de una tabla se presentan nuevas necesidades, dentro de ellas encontramos una de ¿qué hacemos cuando tengamos varios CRUDs y necesitemos llamarlo de forma dinámica?, la respuesta a esto se puede resolver de varias maneras, pero existe una que nos brinda Java, los menús!.

En el paquete vista se creará un nuevo formulario (JFrame) con el nombre de MenuPrincipal



Autoria Propia

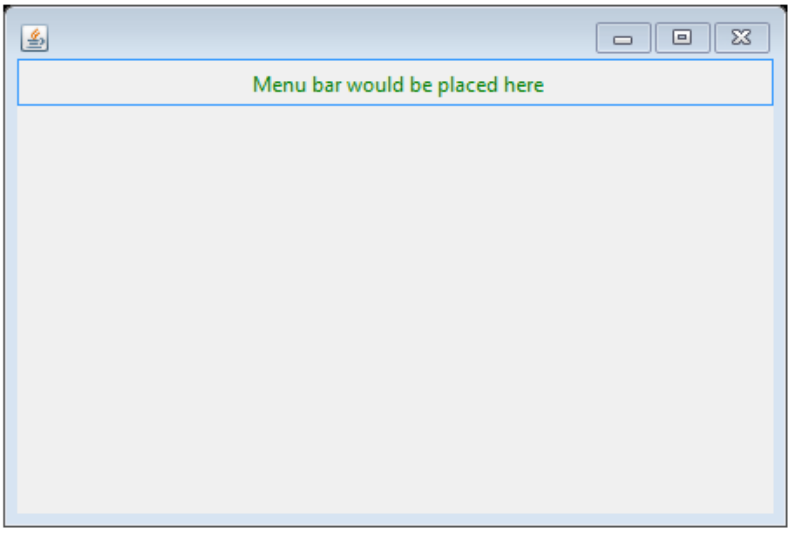
Se agrega un **Layout** como en el caso del primer formulario creado, para este caso se toma un **Absolute Layout**, en la paleta de controles se encuentra una pestaña llamada Menú



Autoria Propia

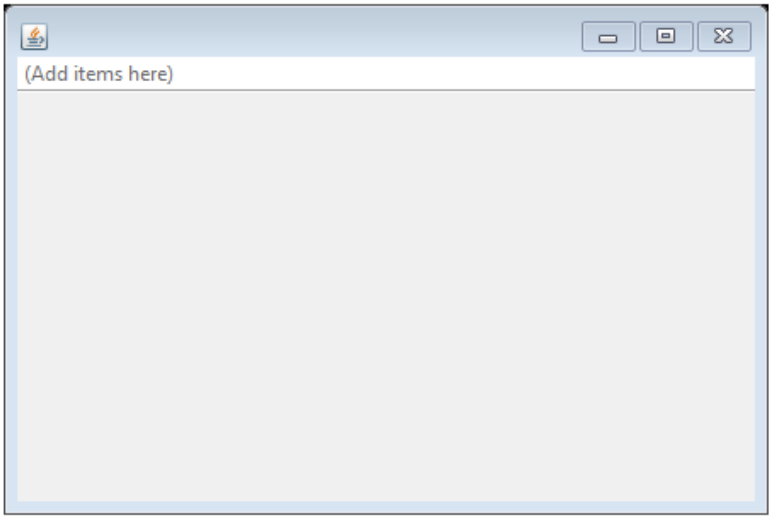
De estas opciones se requieren tres de ellas, JMenuBar (crea una barra de menú en la parte superior), JMenu (especifica cada opción principal del Menú horizontal), JMenuItem (especifica cada una de las opciones a ser invocadas).

CREACIÓN DE UNA JMENUBAR



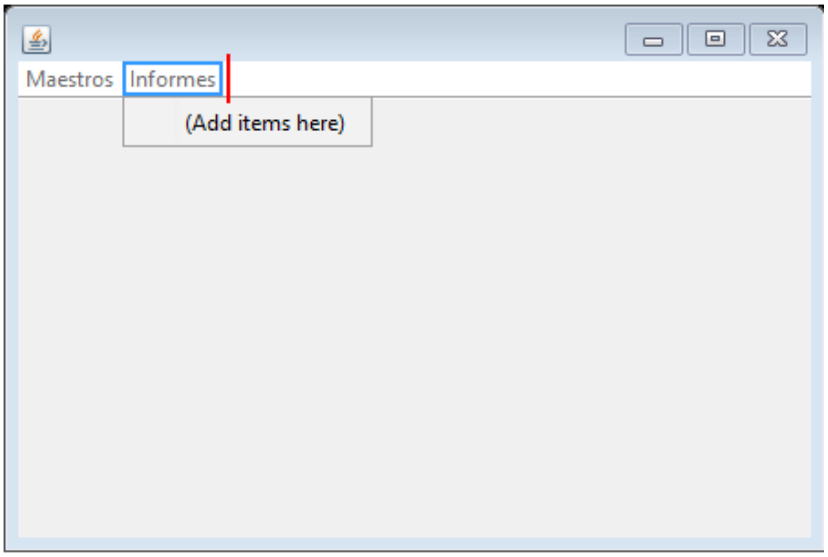
Autoria Propia

Seleccionando el control JMenuBar y ubicando el cursor levemente arriba el inicio de la Barra Menú aparecerá este contenedor quedando así:



Autoria Propia

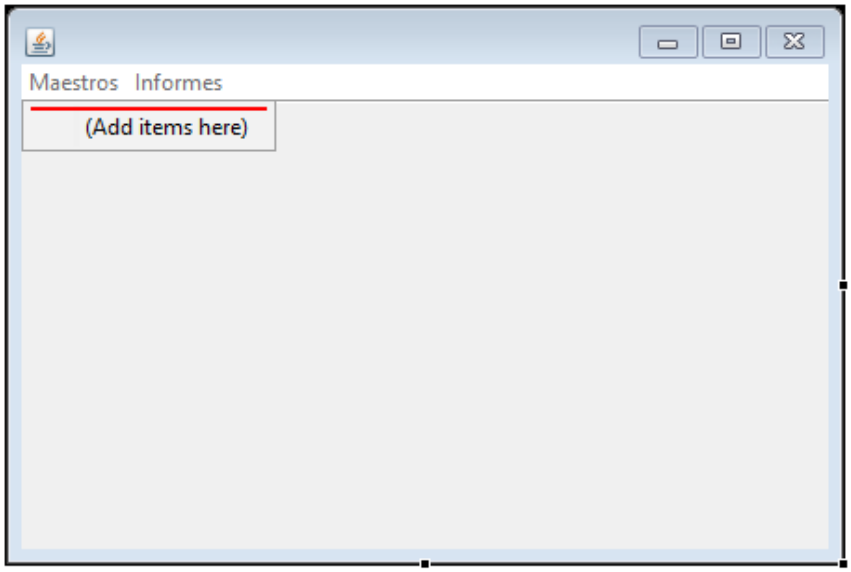
Sobre este contenedor se colocarán las opciones del menú principal, esta se realizará con la opción JMenu.



Autoria Propia

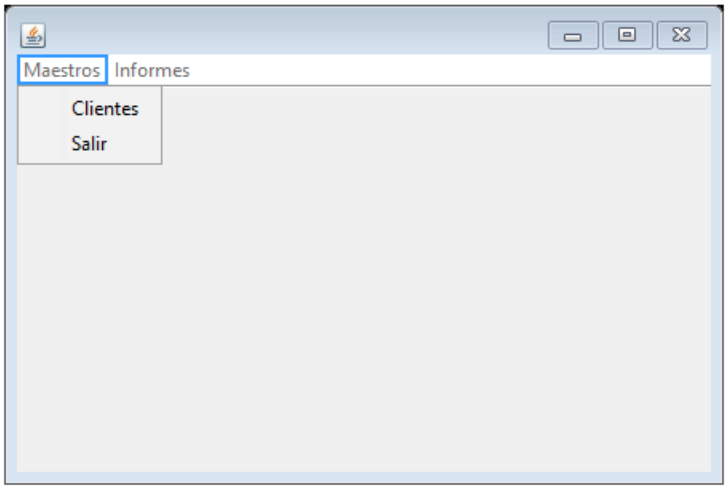
Se podrán agregar todos los que se consideren necesarios y el espacio permita

Por último, se procede a crear cada uno de los ítems que contendrá cada título Principal (JMenuItem)



Autoria Propia

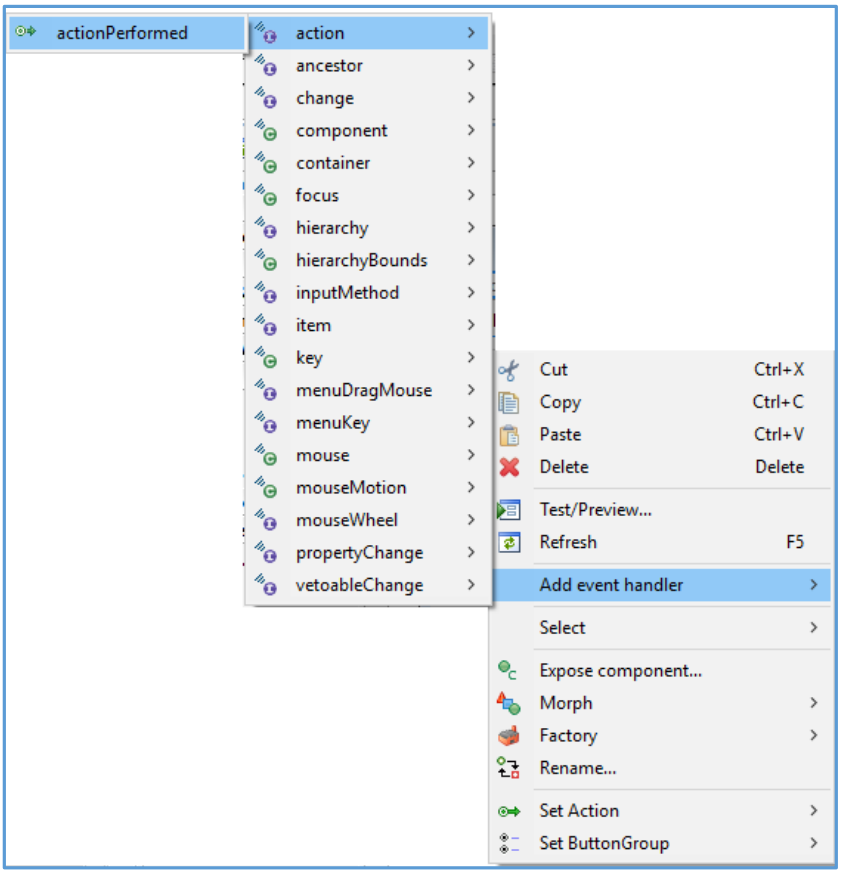
La línea roja es fundamental como guía de la ubicación del menú, cuando aparece vertical indica que es uno como en el JMenu, cuando parece horizontal indica que es desplegable, en este caso debajo de Maestros, para este caso se crearan 2, uno de Cliente y otro de Salir.



Autoria Propia

PROGRAMACIÓN DEL BOTÓN SALIR

Con el botón emergente sobre la opción Salir, seleccione las opciones correspondientes.



Autoria Propia

Dentro de la opción que se genera se agrega la instrucción

```
JMenuItem mntmSalir = new JMenuItem("Salir");
mntmSalir.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {

        System.exit(0);
    }
});
```

Autoria Propia

Esta instrucción permitirá el cerrado del proyecto

Para la invocación de cada opción del menú se aplicarán los mismos pasos con el botón emergente, para este caso sobre la opción de cliente.

```
JMenuItem mntmClientes = new JMenuItem("Clientes");
mntmClientes.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        FrmCliente frmCliente = new FrmCliente ();
        frmCliente.setVisible (true)
    }
});
```

Autoria Propia

Este se aplicará para todos los casos que conformen el menú y se tendría un proceso centralizado para el llamado y la administración de los componentes del proyecto.

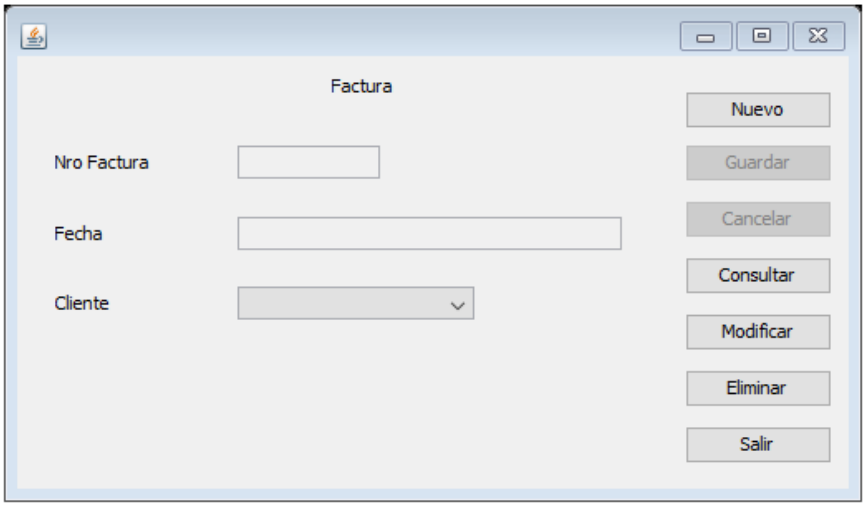
COMBOBOX

Los comboBox hacen parte de un gran número de controles disponibles en la plataforma de Java, nos permite colocar datos predeterminados o hacer un listado de opciones que provienen de una tabla

COMO CREAR Y LLENAR UN COMBOBOX

Aprovechando la tabla de factura que tiene una referencia hacia la tabla cliente veamos el proceso

DISEÑO INICIAL



Autoria Propia

En este ejemplo se muestran los 3 primeros campos de la tabla, 7 botones, dos cajas de texto y un ComboBox, para este último se utilizará el nombre cboCliente y se aplicará como un control tipo campo, para esto seleccione el combo con el menú emergente, **rename…** y luego aplica cambios después de seleccionar la letra F.

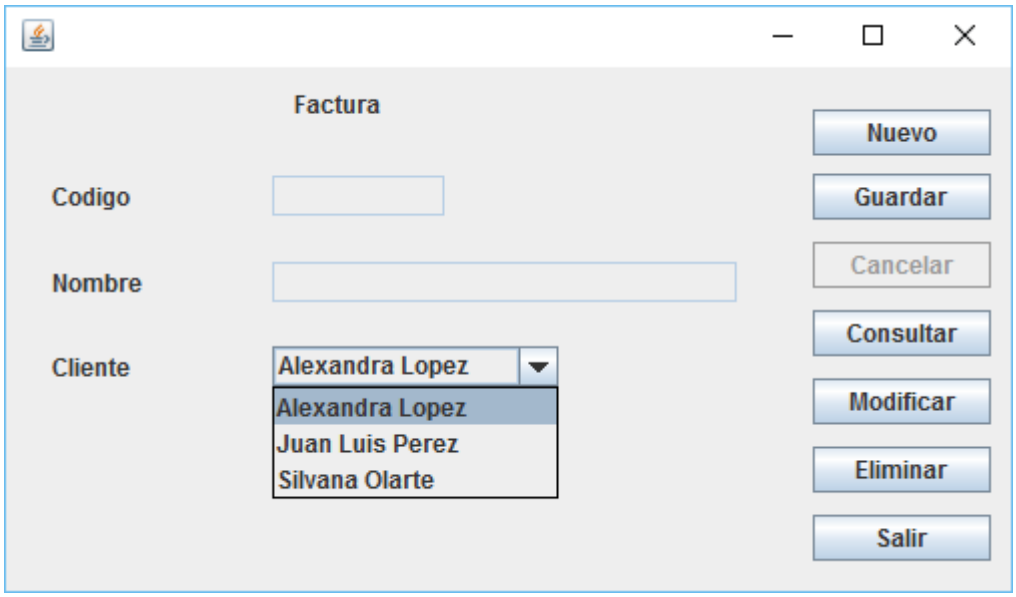
Creación de un método de llenado del ComboBox

```
private static void llenarCboCliente () {

    Cliente cliente = new Cliente ();
    Connection cnn = Conexion.getConnection();
    try{

        PreparedStatement registroCliente = cnn.prepareStatement("Select nombre from cliente "
                + "where estado ='Activo' order by nombre");
        ResultSet rs = registroCliente.executeQuery();
        while (rs.next())

            cboCliente.addItem(rs.getString("nombre"));
    }
    catch (SQLException e) {

        System.out.print ("" + e);
    }
}
```

Autoria Propia

En este método se aplican todos los conceptos vistos previamente, para el llenado del combo aplica dentro del ciclo mientras donde se encuentra cboCliente.addItem y el campo con el que se desea llenar este combo.

Para la invocación del método se ubica al final del constructor de la clase del formulario y quedaría así:



Autoria Propia

Los ComboBox están compuestos por el valor a mostrar, en este caso los nombres de los clientes, cuando se requiere almacenar el identificador de cada cliente en la tabla referencial se aplicará otro método de la siguiente forma.

```
private static String recuperarCliente (int posicionRegistro) {

    Cliente cliente = new Cliente();
    Connection cnn = Conexion.getConnection();
    try{

        PreparedStatement registroCliente = cnn.prepareStatement("Select cedula from cliente "
                + "where estado ='Activo' order by nombre limit " + posicionRegistro + ",1");
        ResultSet rs = registroCliente.executeQuery();
        if (rs.next())

            cedula = rs.getString("cedula");
    }
    catch (SQLException e) {

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

    return cedula;
}
```

Autoria Propia

En el botón de guardar se aplicaría este valor.

```
int posicionRegistro = cboCliente.getSelectedIndex();
String cedula = recuperarCliente(posicionRegistro);

cliente.setCedula(txtCedula.getText());
cliente.setNombre(txtNombre.getText());
cliente.setCedula(cedula);

if (opc == 1)

    fachada.insertarCliente(cliente);
```

Autoria Propia

La variable posicionRegistro es un index de la ubicación del nombre del cliente seleccionado, según esto se busca en el método de cboCliente, a la variable cedula se le asignará el valor que identifica ese nombre y se podrá guardar

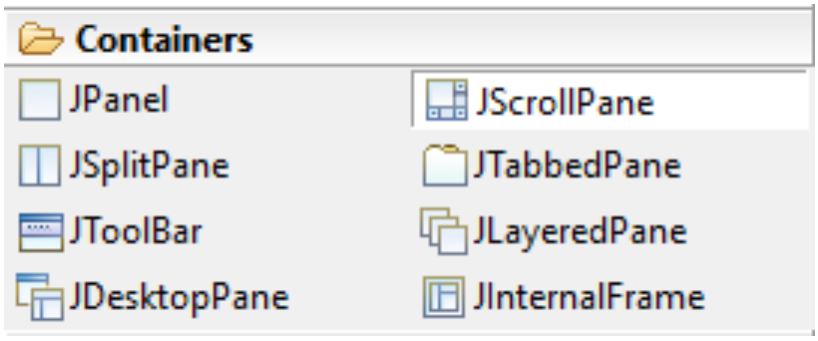
JTABLE

Un control adicional para este proyecto son los JTable, son controles que nos mostraran los datos que deseamos según una consulta o un listado general de datos, con esto podríamos realizar múltiples operaciones, dentro de ellas eliminar, facilitar la modificación entre otras.

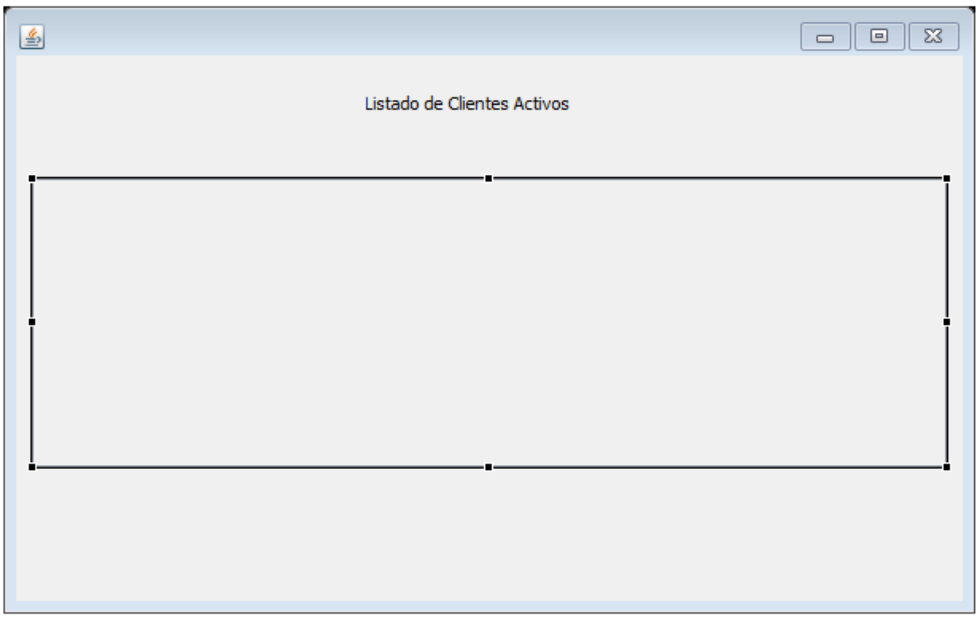
Para este ejemplo se creará un nuevo formulario en el que se puedan visualizar los valores ingresados en una tabla, tenga presente que esto se puede hacer en un formulario como el de factura para adicionar los detalles de esta, o en el de cliente para listar todos los que se tengan en un momento dado.

Cree un formulario llamado FrmClienteTotales, agregando el Layout utilizado anteriormente (Absolute Layout)

Agregue un control (componente) JScrollPane que está ubicado en la pestaña Containers, que ocupe el ancho del formulario y más o menos unos 5 centímetros del alto

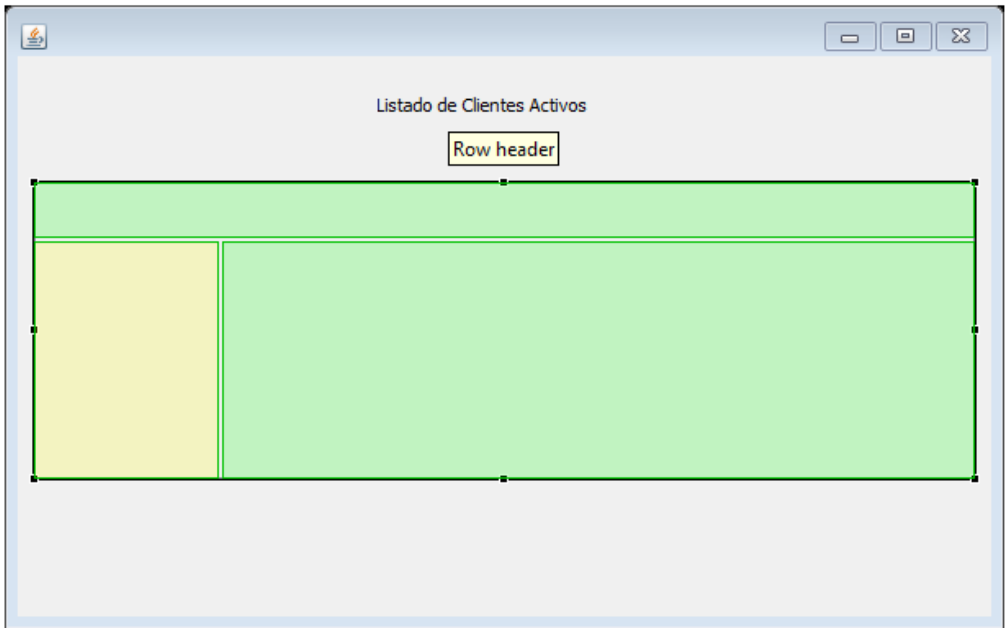


Autoria Propia



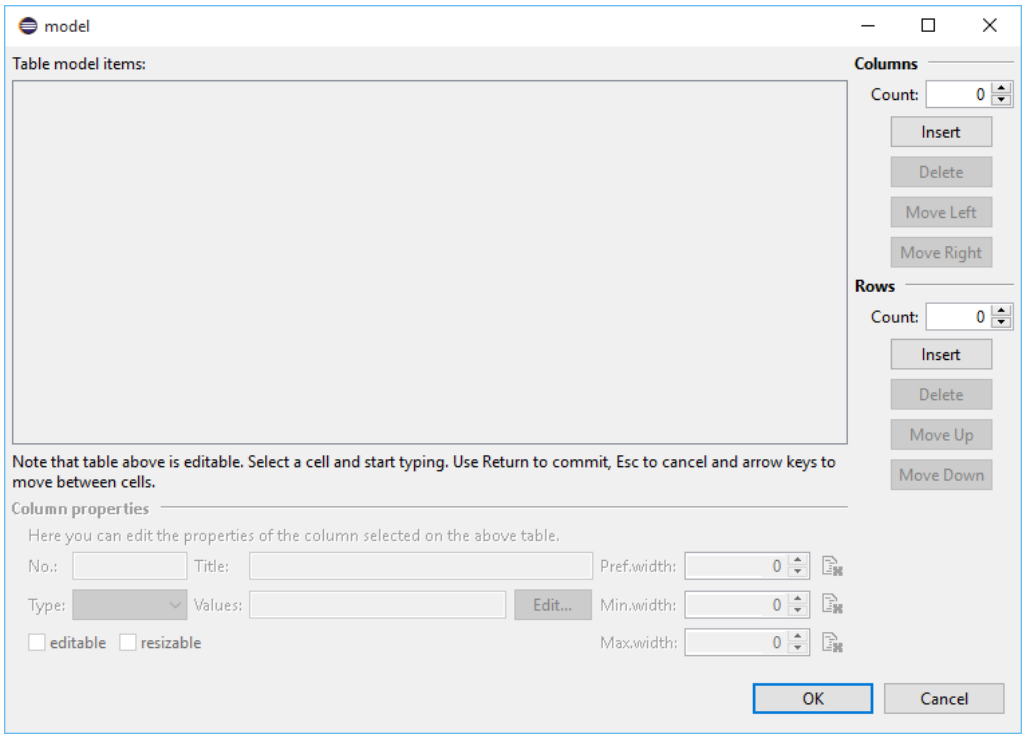
Autoria Propia

Luego seleccione el JTable, ubíquelo encima del control agregado recientemente



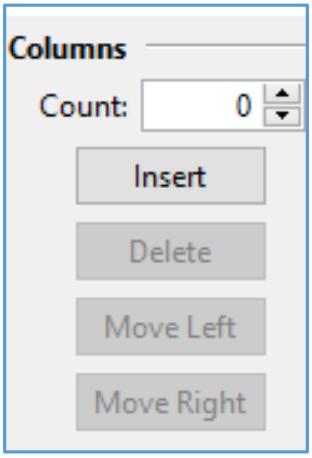
Autoria Propia

Seleccione el control JTable y en la ventana de propiedades elija la opción **model**



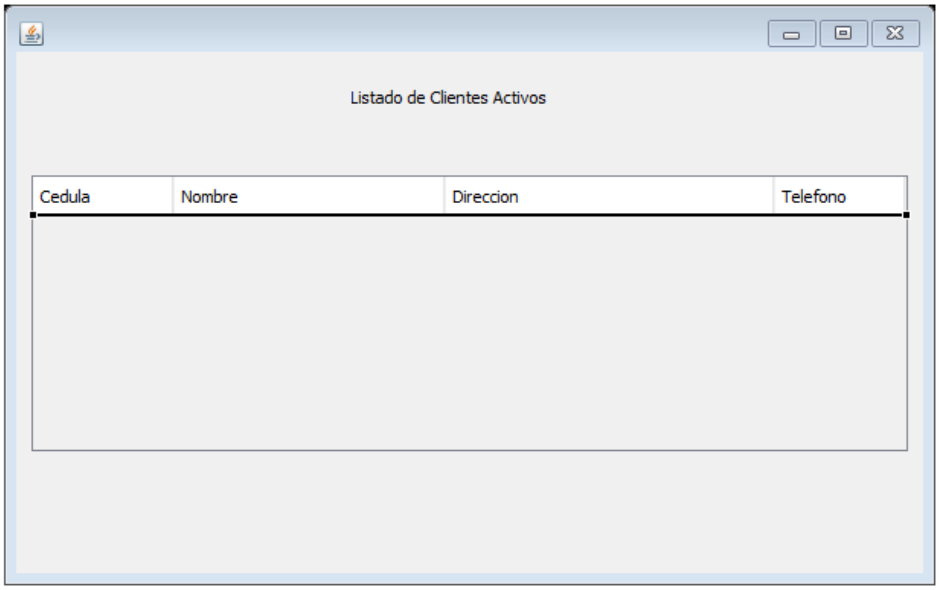
Autoria Propia

En la parte superior derecha selección insert para agregar los títulos de las columnas que llevara el JTabla



Autoria Propia

Para este caso se agregarán Cedula, Nombre, Dirección y Teléfono y luego seleccione el botón OK



Autoria Propia

Después de tener el diseño del formulario con el JTable, proceda a colocarle el nombre del control, para esto se ubica en las propiedades y en la primara opción **variable**, coloque tblCliente, este proceso trae un nuevo tema, se llama **ArrayList**, también conocido como un arreglo dinámico, maneja unas características similares a los arreglos tradicionales, pero con algunas ventajas en el manejo, en los "índices", la adición y eliminación de información.

El primer paso es ubicarse en el archivo DAO de cliente y proceder a crear un nuevo método.

```
public static ArrayList <Cliente> listarClientes (){

    ArrayList <Cliente> clientes = new ArrayList <Cliente> ();
    Cliente cliente = null;
    Connection cnn = Conexion.getConnection();

    try {

        PreparedStatement resultadoCliente = cnn.prepareStatement("select cedula, nombre, direccion, "
                + "telefono from Cliente where estado = 'Activo'");
        ResultSet rs = resultadoCliente.executeQuery();
        while(rs.next()){

            cliente = new Cliente();
            cliente.setCedula(rs.getString("cedula"));
            cliente.setNombre(rs.getString("nombre"));
            cliente.setDireccion(rs.getString("direccion"));
            cliente.setTelefono(rs.getString("telefono"));
            clientes.add (cliente);
            cliente = null;
        }
    } catch (SQLException sqle) {

        System.out.print("\n" + sqle);
    }

    return clientes;

}
```

Autoria Propia

Este método inicia con **ArrayList** <Cliente>, esto indica que retornara un arreglo dinámico, el nombre de este método es listarClientes y no tiene parámetros de entrada, en la segunda línea se encuentra nuevamente ArrayList <Cliente> clientes, este **clientes** es la referencia o el elemento que se encarga de recolectar todos los registros de la tabla que cumplan la condición de estar activos, las demás líneas ya son conocidas hasta la penúltima línea de **try** en la que aparece la referencia **clientes** nuevamente acompañada del método **add**, es acá donde llevamos un registro a el arrayList, por ultimo retornamos clientes, tengan presente que acá solo se está llenado un arreglo dinámico con la información, todavía no se muestra en pantalla.

Como segundo paso se ubicará en el archivo facade o fachada y se realizará el llamado el método anteriormente creado.

```
public static ArrayList listarClientes(){

    return clienteDAO.listar();
}
```

Autoria Propia

Se indica nuevamente que se retornara un **ArrayList**

Y como tercer paso se creará un método para mostrar la información en pantalla, este se realiza en el formulario, en la parte inferior del código antes de la última llave.

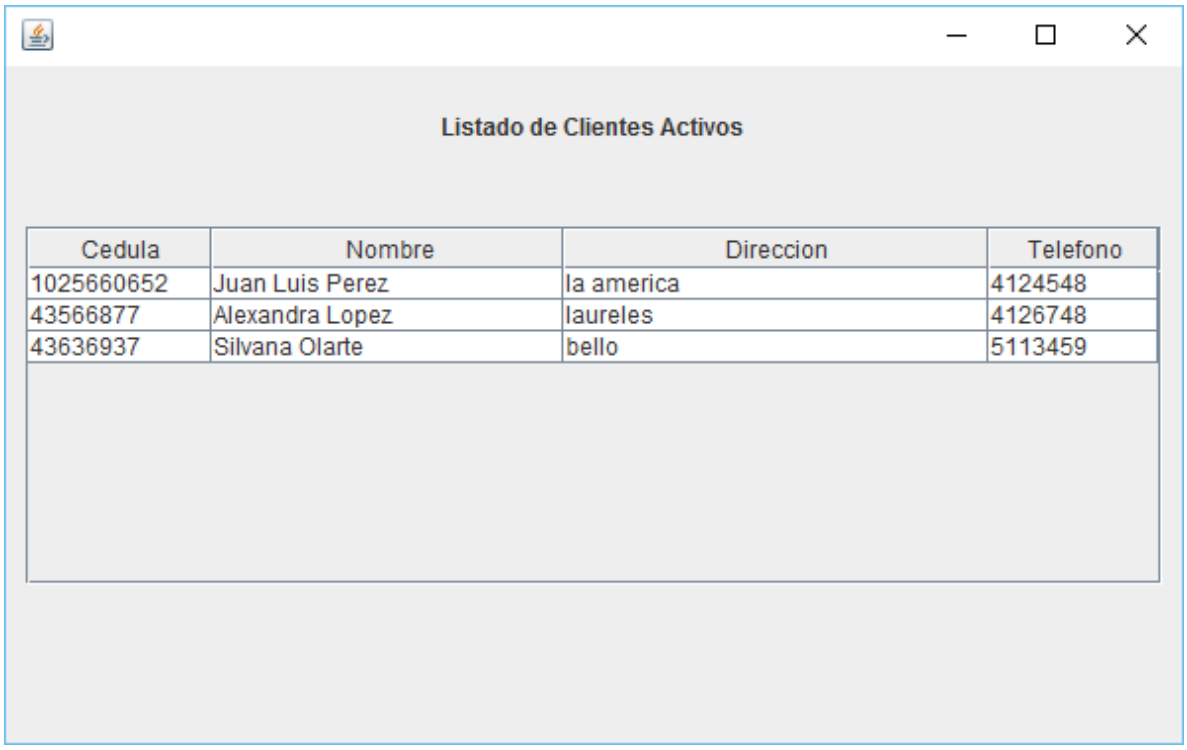
```
private static void mostrarClientes (){

    ArrayList <Cliente> clientes = new ArrayList<Cliente> ();
    clientes = fachada.listarClientes();

    DefaultTableModel modelo = (DefaultTableModel) tblCliente.getModel();

    for(int i = 0; i < clientes.size(); i++){

        Vector <String> vector = new Vector<String>();
        vector.add(clientes.get(i).getCedula());
        vector.add(clientes.get(i).getNombre());
        vector.add(clientes.get(i).getDireccion());
        vector.add(clientes.get(i).getTelefono());
        modelo.addRow(vector);
    }
}
```

Autoria Propia

En este método encontramos nuevamente la referencia a un **ArrayList** con la aplicación del elemento de **clientes**, luego aparece el llamado de la fachada

En la tercera línea se encuentra la definición del JTabla donde se depositarán los datos, se crea otra referencia de la tblCliente llamado modelo.

En el ciclo **para** se realiza un recorrido de todos los datos a mostrar, para saber la totalidad de los datos se tiene el método **size**, adicional dentro del ciclo se trabaja con la clase Vector y se procede a "pintar" la información en el JTable, el resultado es el siguiente.



Autoria Propia

PISTAS DE APRENDIZAJE

CRUD: Acrónimo de Crear, Leer, Actualizar y Eliminar
BD: Base de Datos
Conector: Driver controlador entre el aplicativo y la BD
Paquete: Espacio que permite la clasificación de los procesos
Validación: Herramienta Fundamental para el control de la información
Facade: Patrón de Diseño
DAO: Data Access Object
Instancia: Creación de una referencia de una clase
Clase Referencia: Clase que contiene los Getters y los Setters
Gettes / Setters: Herramienta que permite tener mas control de los datos, así como un mayor alcance

4.3 TEMA 2 POOL DE CONEXIONES

Un pool de conexiones es **un conjunto limitado de conexiones** a una **BD**, un pool permite **centralizar** y **controlar** el acceso que se tenga por parte del sistema, con esto podemos concluir que la cantidad de conexiones abiertas es limitada teniendo en cuenta que estas consumen muchos recursos, memoria y tiempo de procesador, este proceso adema favorece **la escalabilidad de la aplicación.**

Dentro de las ventajas que se pueden observar en un Pool de Conexiones además de las ya descritas está el **poder acceder o cambiar algunos parámetros sin necesidad de volver a compilar el aplicativo**, cuando creamos una conexión tradicional como la vista en la primera unidad esta tiene los valores “quemados”, esto quiere decir que el conector, usuario, contraseña y la base de datos están predeterminados, si sucede algún cambio después de compilar y empaquetar el aplicativo habrá que volver a realizar el proceso, para evitar estos datos “quedamos” se procede a crear un archivo **jdbc.properties** en la raíz del **src** con la siguiente estructura.

jdbc.properties

```
1 usuario=root
2 clave=admin
3 controlador=com.mysql.jdbc.Driver
4 ruta=jdbc:mysql://localhost/facturacion
```

Autoria Propia

Posterior a este archivo se creará una clase llamada Conexion.java con la siguiente estructura.

```
package utilidad;

import java.sql.Connection;
import java.sql.DriverManager;
import java.util.ResourceBundle;
public class Conexion {

    private static Connection con = null;
    public static Connection getConnection () {

        try {

            if (con == null) {

                Runtime.getRuntime().addShutdownHook(new ShutdownHook());
                ResourceBundle res = ResourceBundle.getBundle("jdbc");
                String controlador = res.getString("controlador");
                String ruta = res.getString("ruta");
                String usuario = res.getString("usuario");
                String clave = res.getString("clave");
                Class.forName(controlador);
                con = DriverManager.getConnection(ruta, usuario, clave);
            }
            return con;
        }
        catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException("Error al crear la conexion!", ex);
        }
    }
```

Autoria Propia

Obsérvese que la clase recibe unos parámetros que están ubicados en el archivo **properties** luego de abrir el archivo **jdbc**, se leerán los datos, en este caso ruta contiene el **jdbc** y la Base de Datos, el usuario, la contraseña y el controlador, esto hará que se pueda manipular de una forma más simple todo el sistema.

Posterior a la última clase se agregará el siguiente fragmento de código complementario

```
static class ShutdownHook extends Thread {

    public void run() {

        try {

            Connection con = Conexion.getConnection ();
            con.close();
        }
        catch( Exception ex ) {

            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }
}
```

Autoria Propia

PISTAS DE APRENDIZAJE

Conexión todo el sistema de BD en una aplicación requiere el vínculo o la comunicación, esta tarea se realiza con un "puente" llamado conector.

Propertie en un tipo de extensión aplicada a un archivo que puede contener valores no "quemados" y que se puedan cambiar sin necesidad de volver a compilar el aplicativo

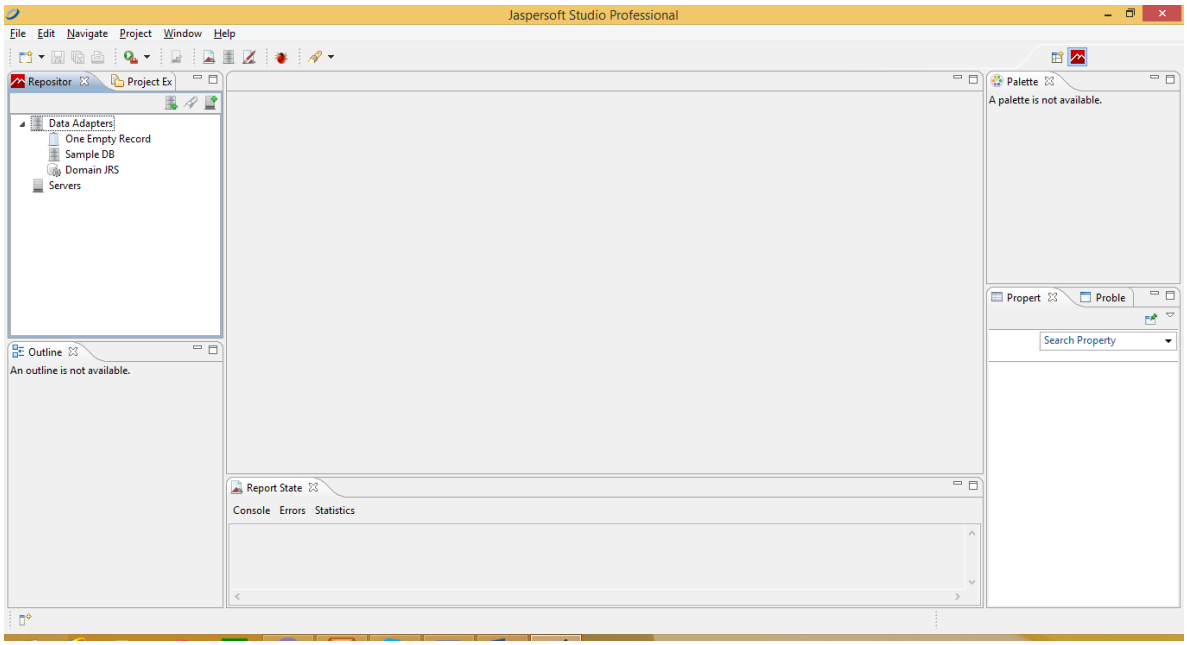
Escalabilidad diseño de un aplicativo a BD que permite que en un futuro pueda seguir creciendo sin necesidad de volver a iniciar o hacer cambios grandes

4.4 TEMA 3 REPORTES

La creación de reportes en un aplicativo se convierte en una necesidad fundamental, es la posibilidad de expresar los resultados de una sentencia SQL en papel o en un documento electrónico, mediante un archivo pdf, xlsx, html, docx entre otras extensiones.

Para la generación de reportes se utilizará la herramienta **jasper-studio**, es un **IDE** muy simple de manejar y de gran alcance.

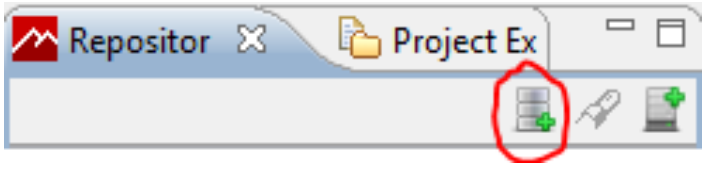
El **Jasper Studio** tiene un aspecto muy similar al eclipse, observémoslo



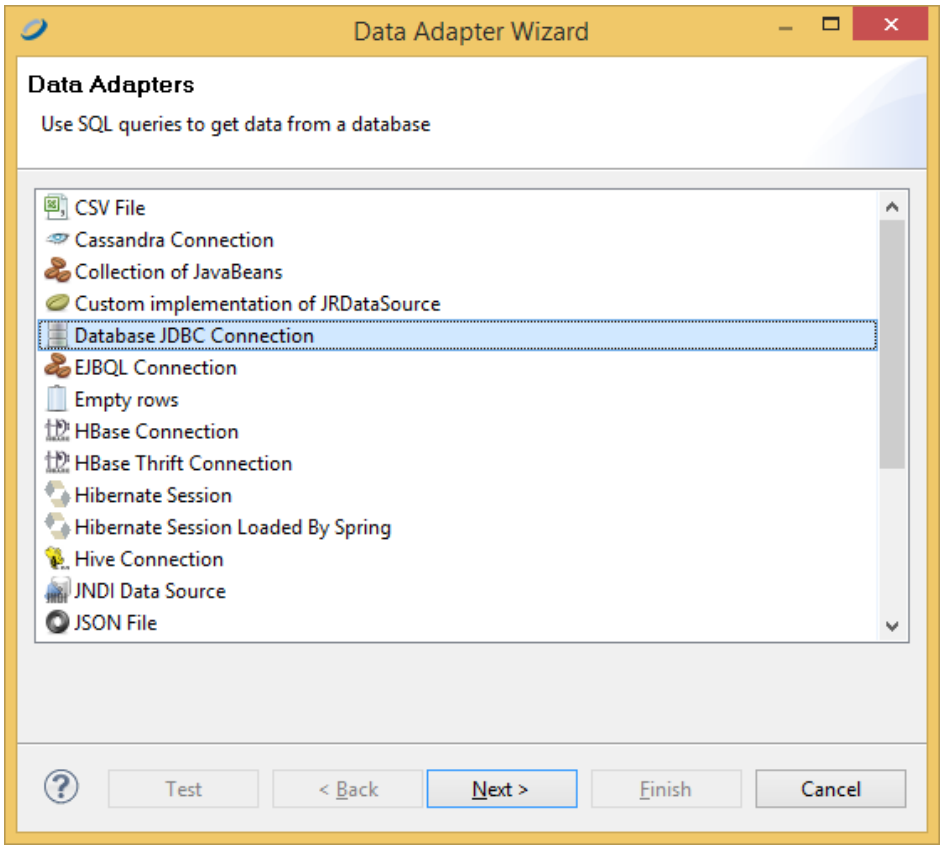
Autoria Propia

CONFIGURACIÓN

Para la configuración del **Jasper** se requiere especificar el motor de bases de datos que se esta utilizando en el proyecto de **Eclipse**, para esto se selecciona el siguiente icono (**Create Data Adapter**)

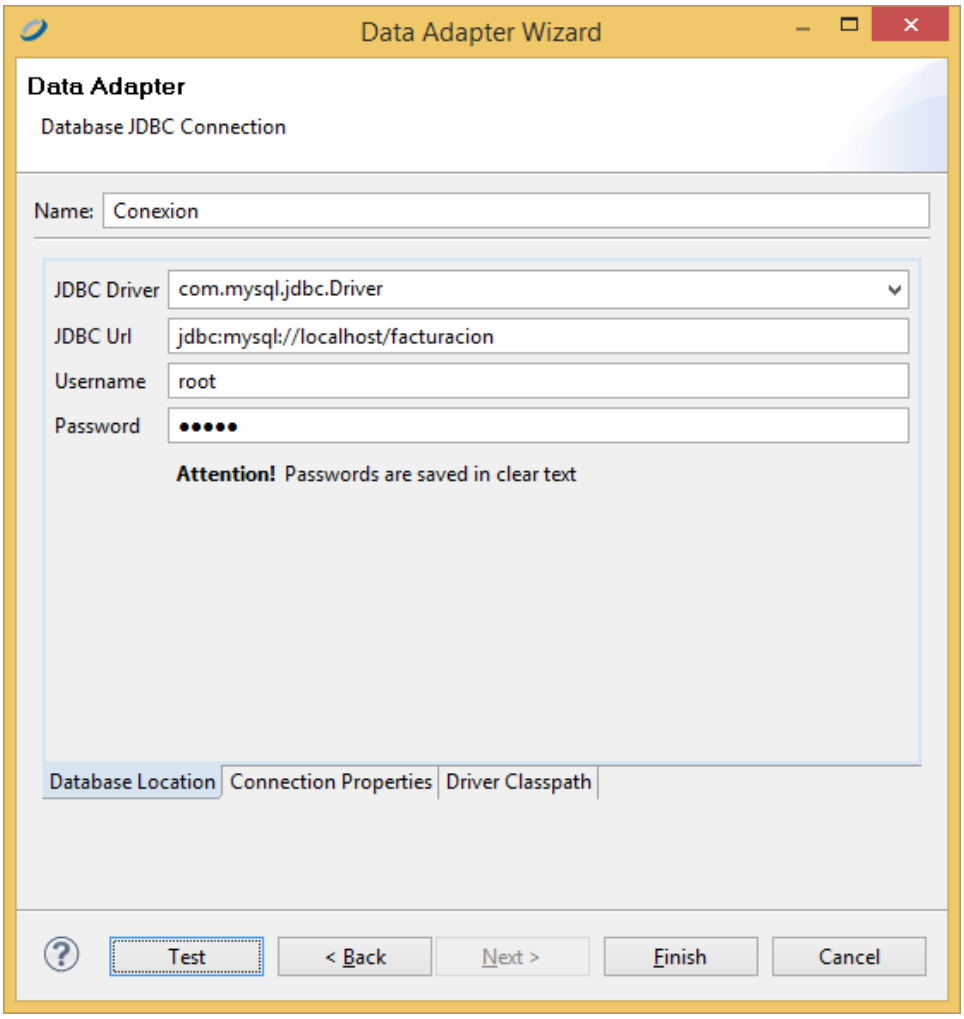


Autoria Propia



Autoria Propia

Se selecciona **Database JDBC Connection** y se elige el botón **Next**



Autoria Propia

En esta ventana van las principales opciones de configuración

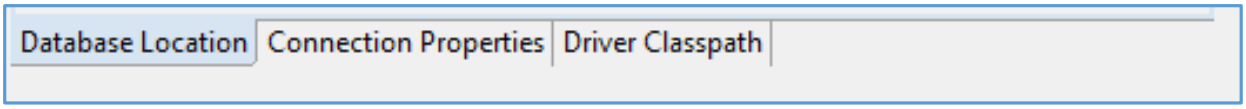
Name: especifica un nombre de conexión para cada proyecto o cada motor de base de datos

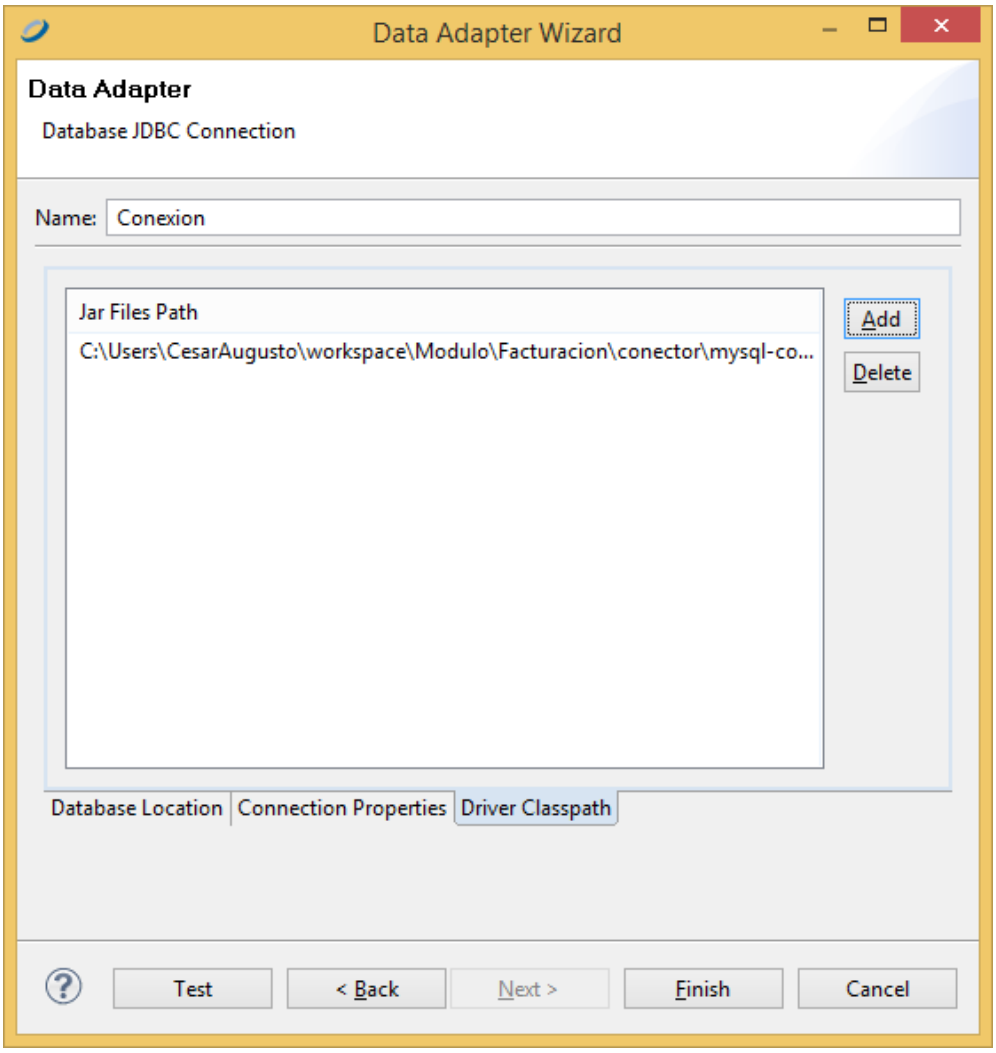
JDBC Url: solo se cambia después de la palabra localhost/ donde se ubica el nombre de la base de datos de trabajo

Username: nombre de usuario de MySQL

Password: contraseña de MySQL

Luego se seleccionan en la parte inferior la pestana Driver Classpath

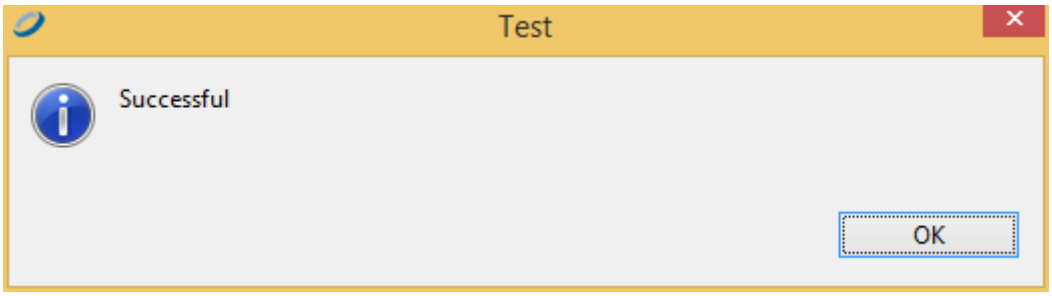




Autoria Propia

Mediante le botón **Add** se agrega el driver o conector que se esta utilizando para comunicar a el proyecto de **java** con **MySQL**

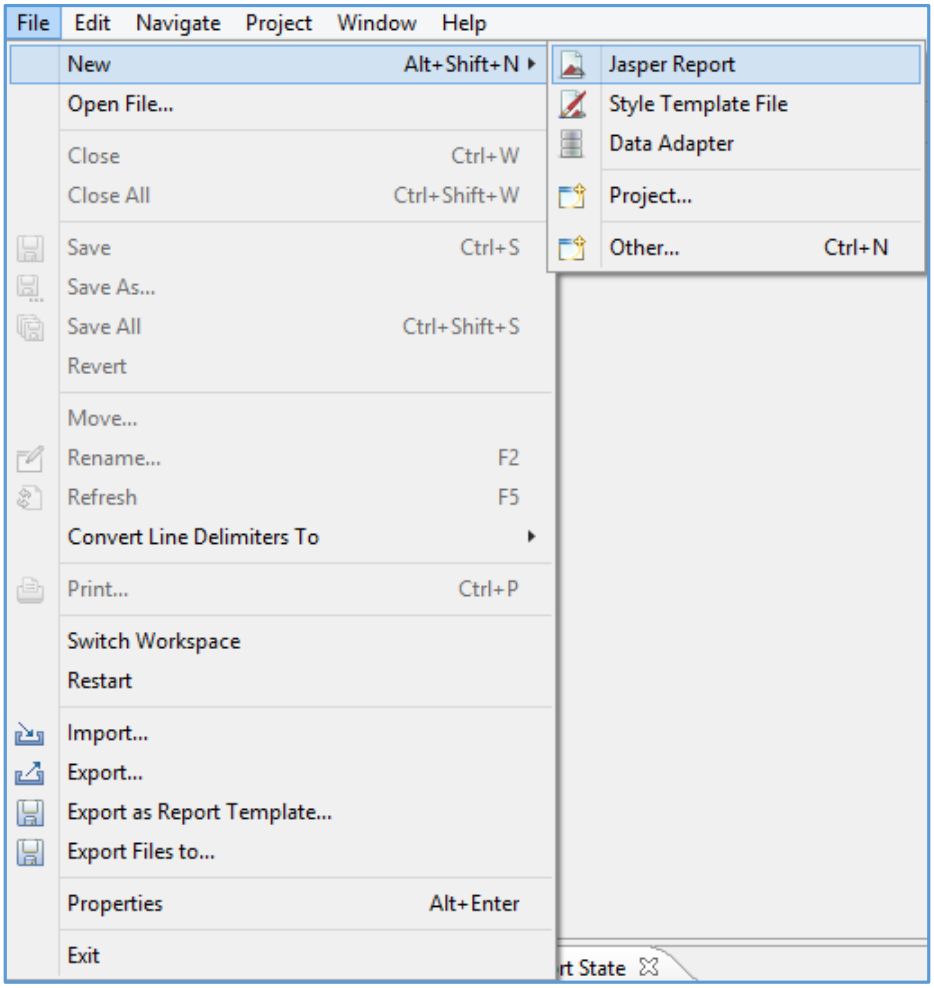
Cuando esto está establecido, se prueba mediante el botón **Test**



Autoria Propia

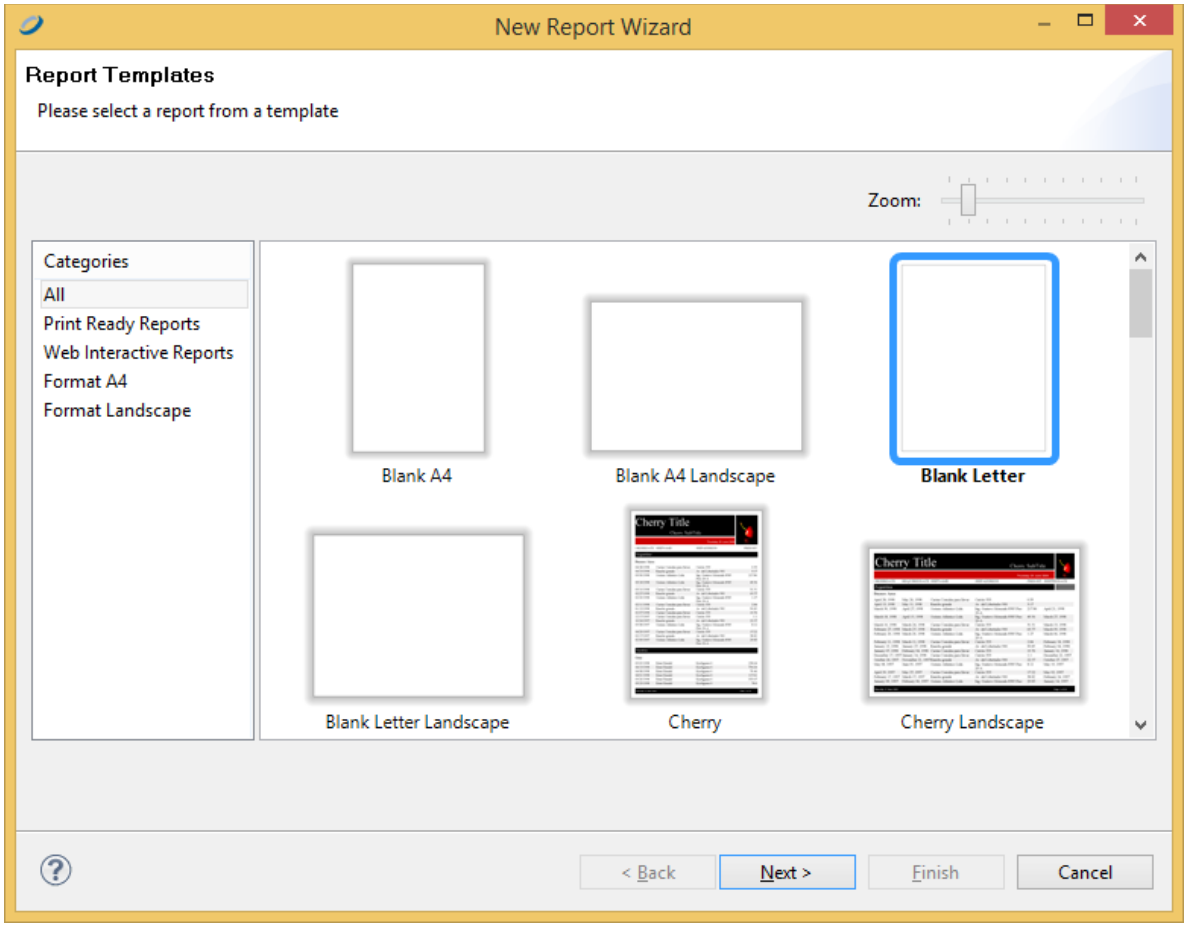
Y si es correcto el proceso deberá visualizarse el mensaje de satisfactorio, **Ok**, y **Finish**.

CREACIÓN DE UN REPORTE EN BLANCO



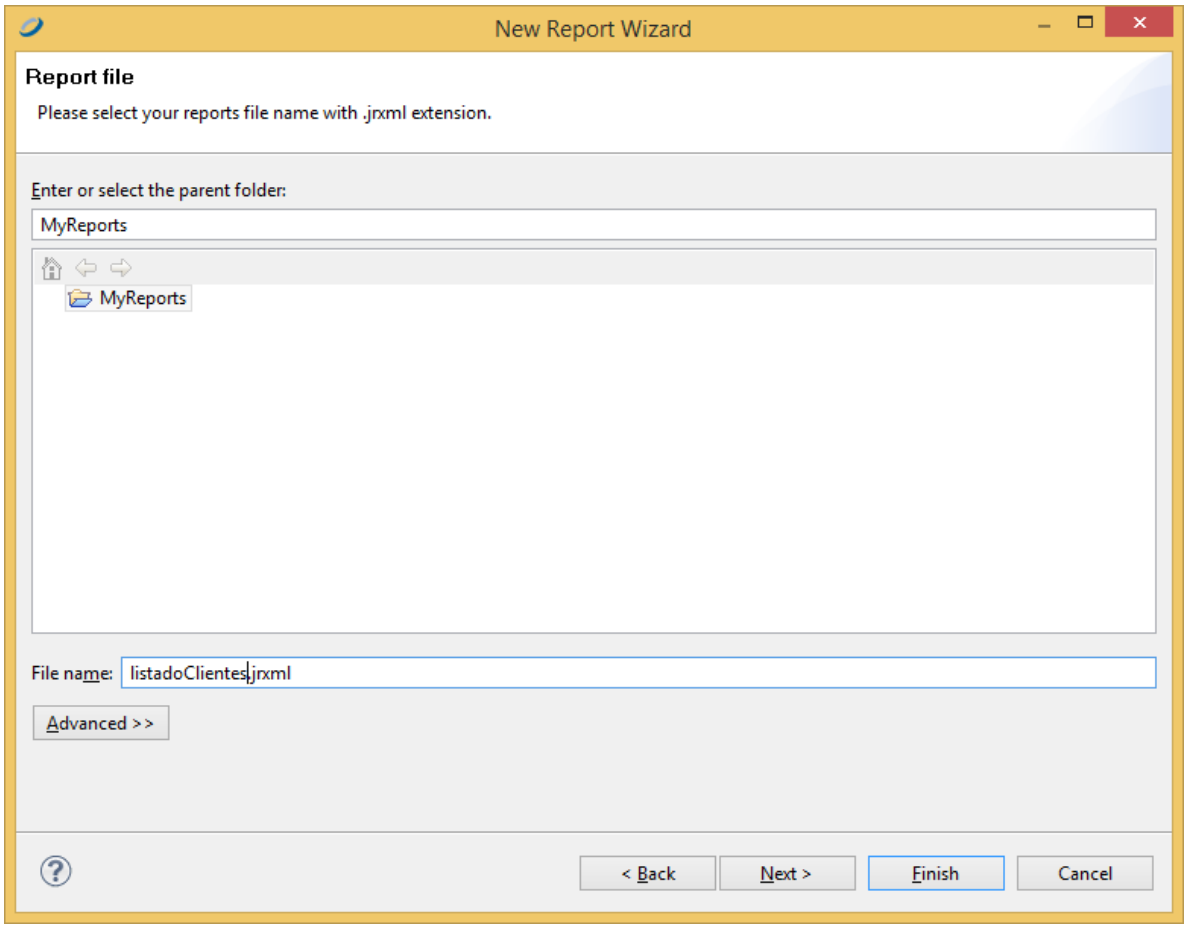
Autoria Propia

Se encuentran muchos tipos de reportes, algunos de ellos pre configurados, con formatos establecidos, para este caso se utilizará uno en blanco tamaño carta



Autoria Propia

Se establecerá uno con el nombre de **listado Clientes**



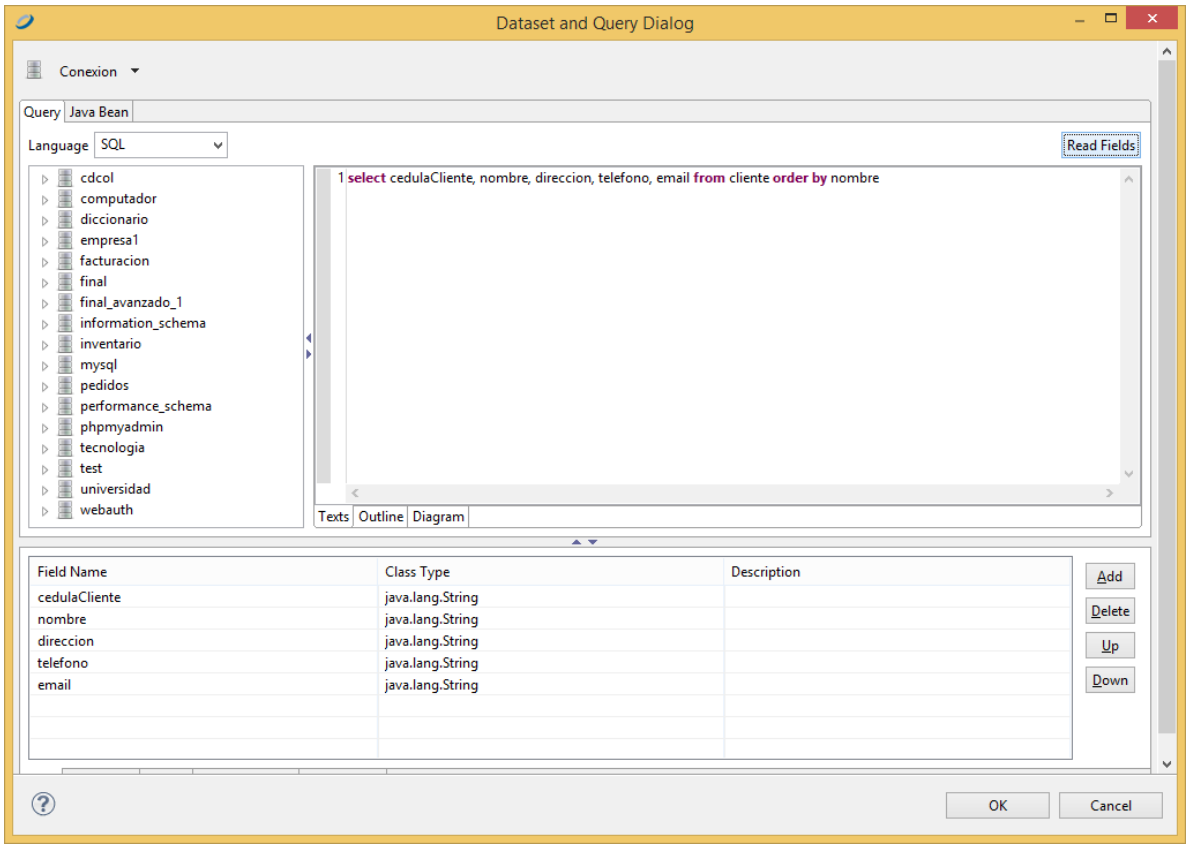
Autoria Propia

Se encuentra con la extensión **jrxml**, al ser un archivo **xml** se podrá editar manualmente o mediante asistente.

ESTABLECER SENTENCIA SQL

Autoria Propia

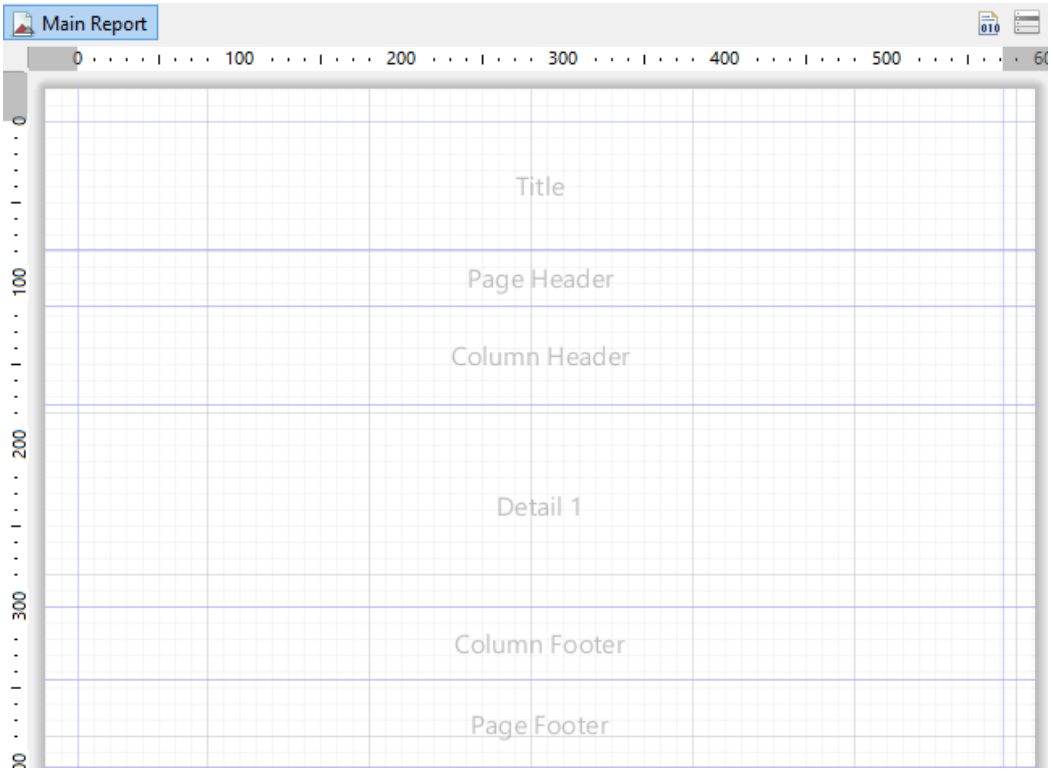
Mediante el icono **Dataset and Query Editor dialog** se creará la instrucción **SQL** de trabajo, inicialmente para una sentencia general.



Autoria Propia

En la esquina superior izquierda se selecciona el nombre de la conexión establecida para este proyecto (**Conexion**), en el dialogo principal se escribe la sentencia SQL sin punto y coma (;) al final para probarla y establecerla se selecciona el botón **Read Fields**, al pulsarlo aparecerán los campos en el dialogo de la parte inferior, OK para terminar

ÁREA DE TRABAJO DEL REPORTE

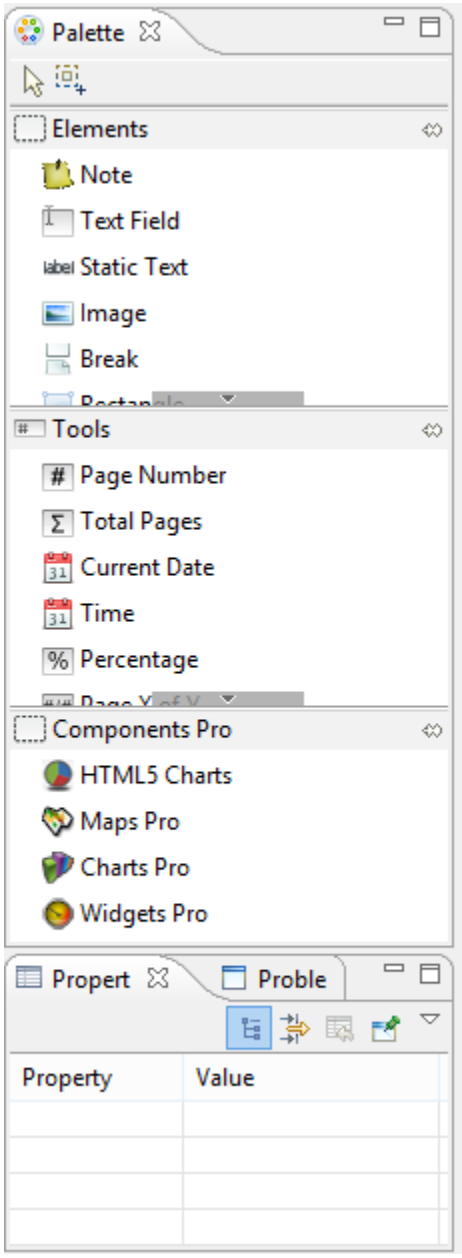


Autoria Propia

En el área de trabajo se encuentran varios ítems en un gris tenue, cada una de ellas para un tema puntual

Title	título principal o cabecera de la empresa
Page Header	títulos secundarios
Column Header	títulos de los campos o campos para información maestra
Detail 1	detallado de la información, esta área representa un ciclo con la información de la sentencia SQL
Column Footer	información de pie de pagina
Page Footer	Pie de Pagina

En el lado derecho se encuentran los controles con sus respectivas propiedades

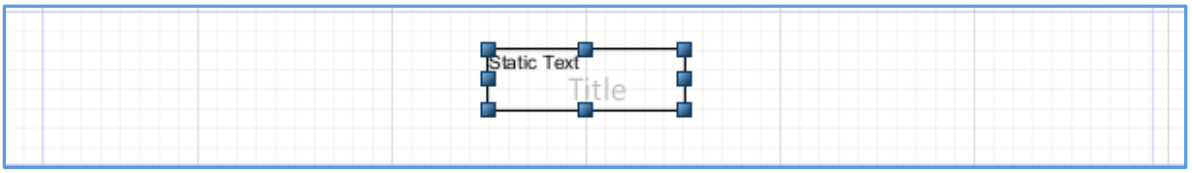


Autoria Propia

Y en el lado izquierdo otros componentes como campos y parámetros

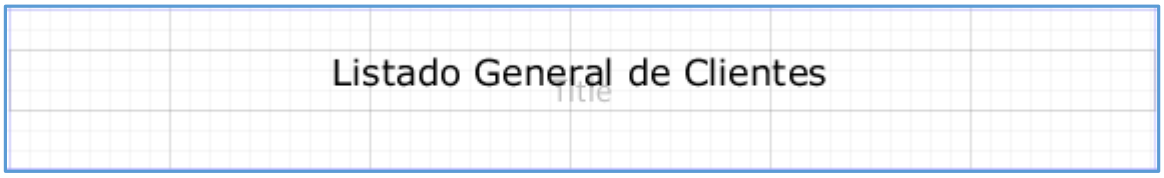
Diseño Básico de un Reporte

Seleccione el elemento **Static Text (label)** del lado derecho de los componentes **(Elements)** y ubíquelo en el área tenue de **Title**



Autoria Propia

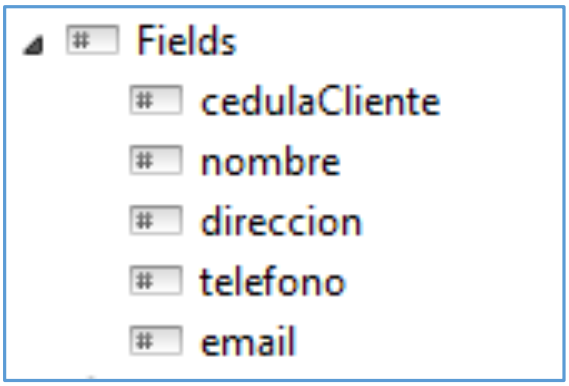
Edite el control y coloque el titulo o mensaje deseado para este informe, se pueden cambiar tipos de fuentes, tamaño y alineación



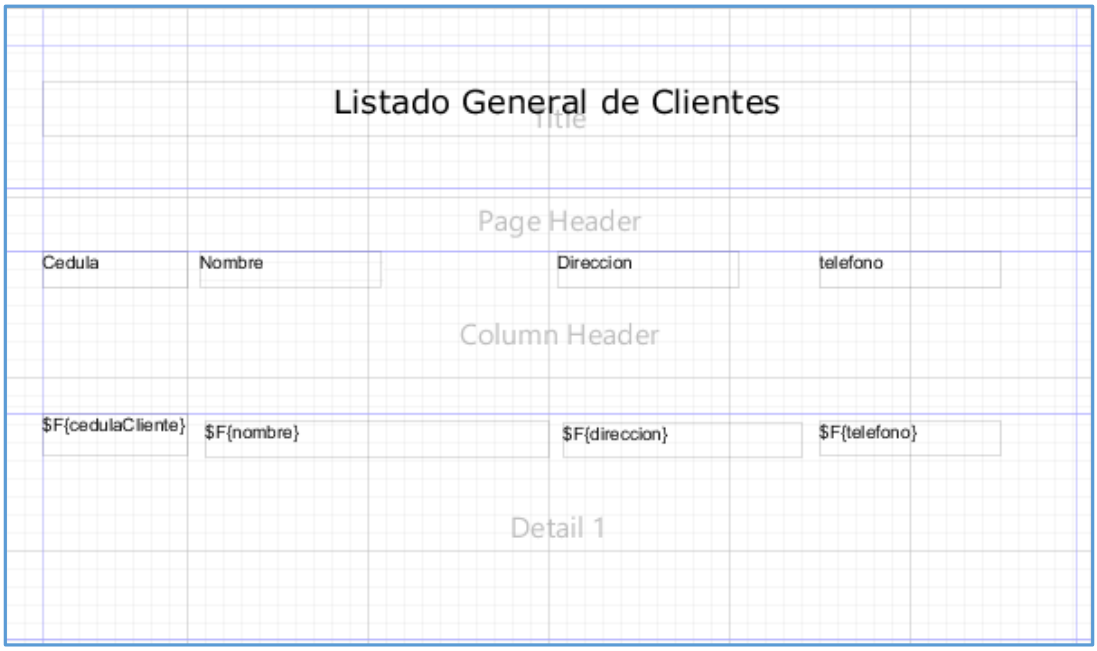
Autoria Propia

El resto del diseño depende exclusivamente de las necesidades visuales y creativas que se deseen aplicar, para este caso se prosigue con la ubicación de los campos.

Al lado izquierdo se encuentra una herramienta llamada **Fields**, al desplegar esta opción aparecen los capos de la sentencia **SQL** y se podrán arrastrar al área de **Detal**

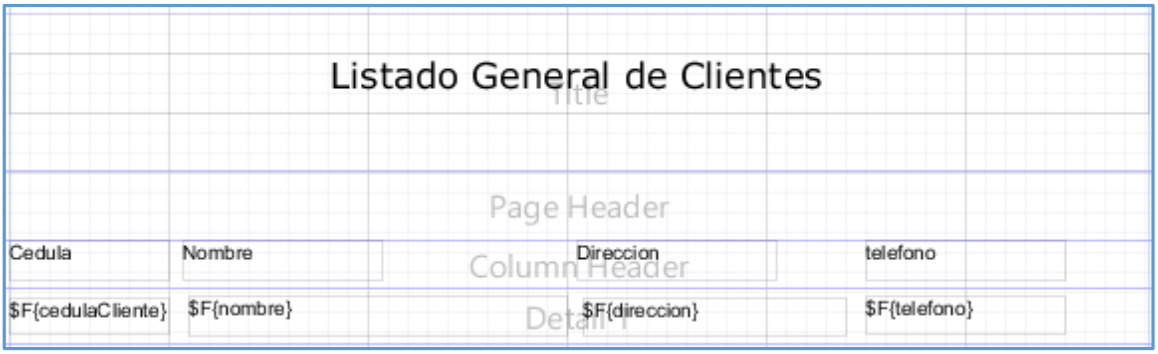


Autoria Propia



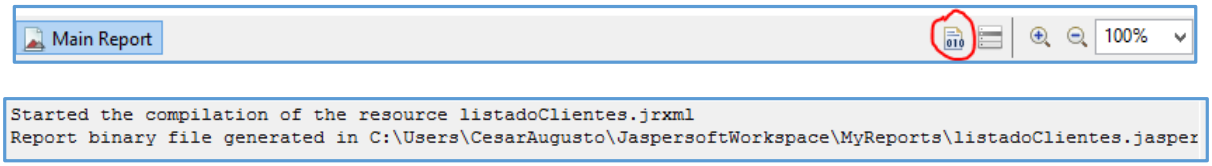
Autoria Propia

Disminuya los espacios de las áreas para una mejor distribución de los datos



Autoria Propia

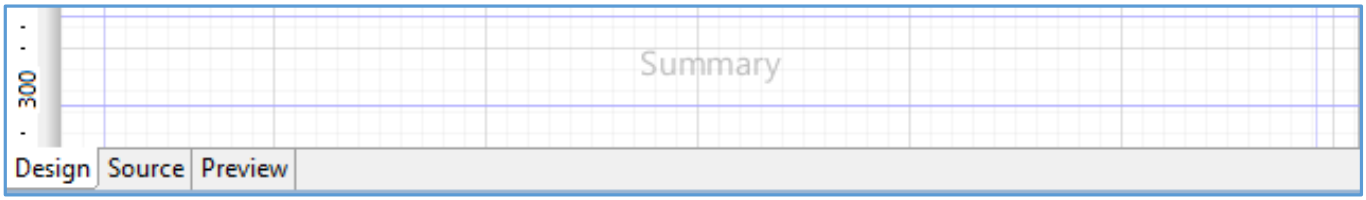
Guarde los cambios y compile el archivo (**Compile Report**)



Autoria Propia

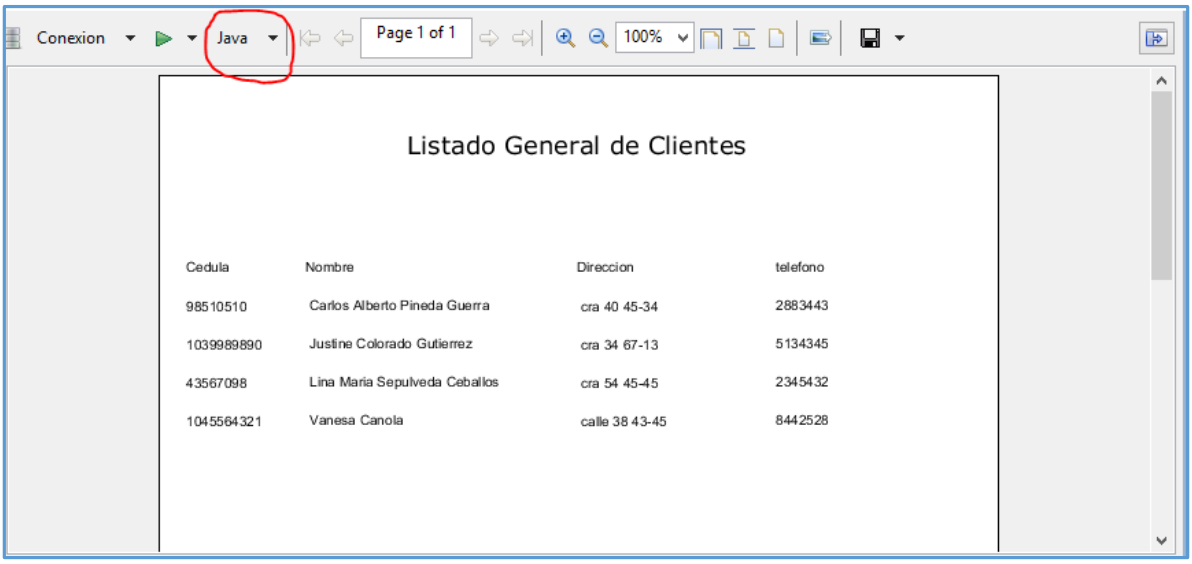
Este es el mensaje de compilación y la ruta de ubicación

VISUALIZACIÓN DEL REPORTE (PREVIEW)



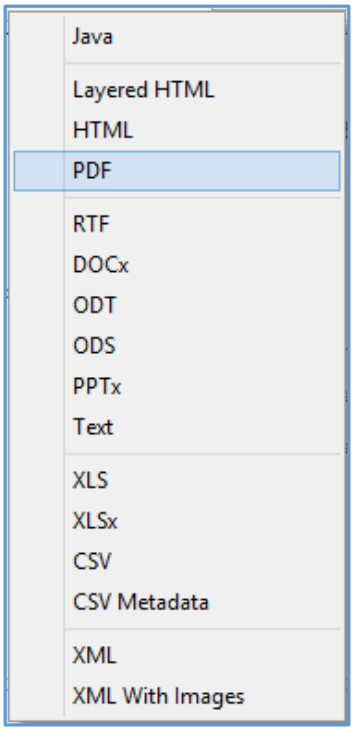
Autoria Propia

Elija la pestana **Preview**



Autoria Propia

Este es el informe generado por el **Jasper**, en el área marcada en la parte superior despliegue y elija el formato del archivo



Autoria Propia

PARA ESTE EJEMPLO SE UTILIZARÁ PDF

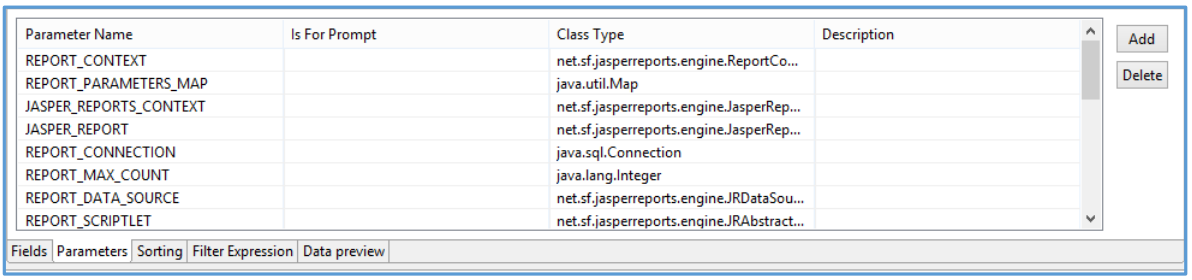
Recompile el proyecto para que tome los últimos cambios y el reporte está listo para ser vinculado al proyecto

El archivo creado es de extensión jasper, el otro archivo que el sistema genera jrxml es un archivo editable.

CREACIÓN DE UN REPORTE CON PARÁMETROS

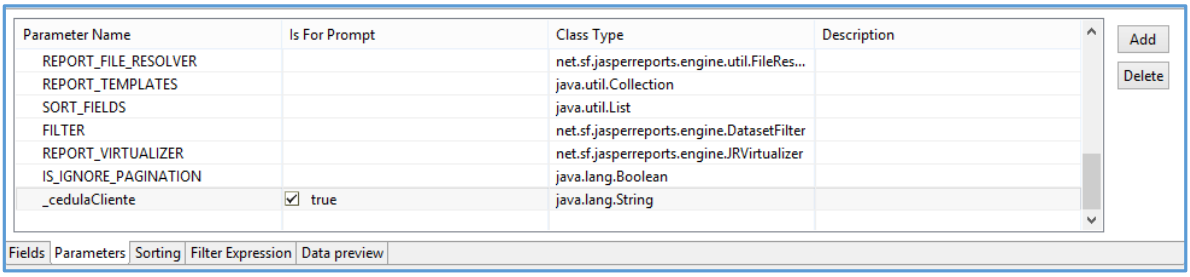
La creación de un reporte con parámetros solo cambia en la sentencia SQL, todos los procesos de diseño son los mismos del reporte anterior.

CREACIÓN DE UN PARÁMETRO



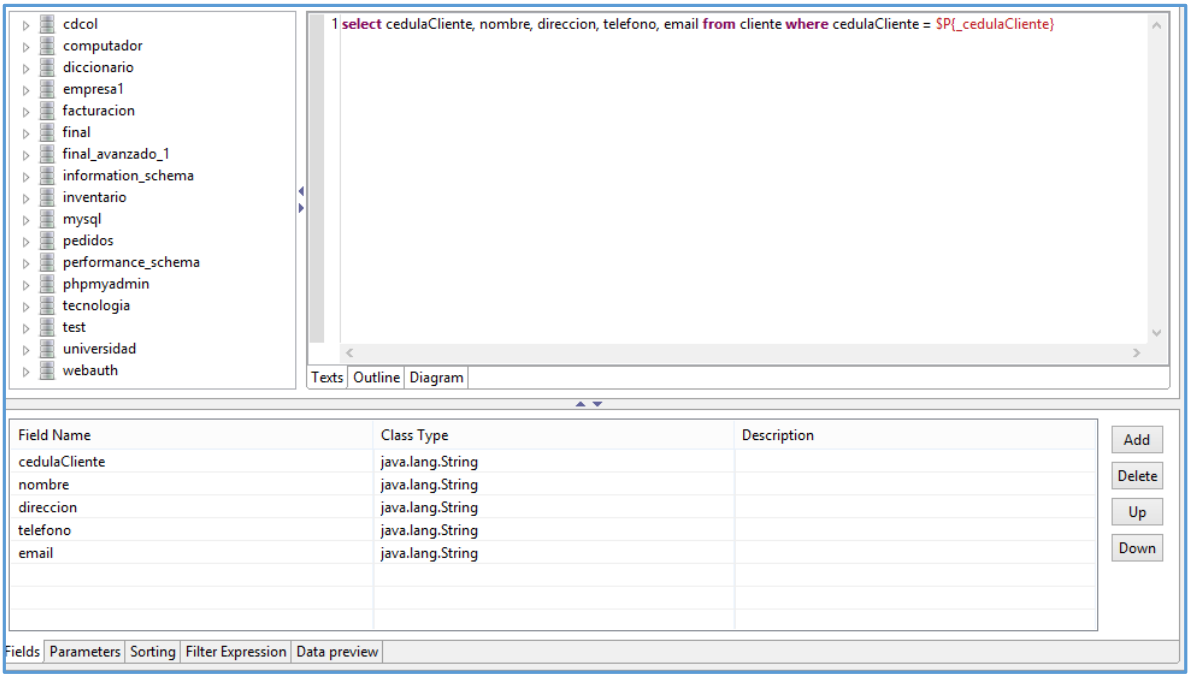
Autoria Propia

En la parte baja de la ventana **DataSet and Query editor dialog** se encuentra una pestaña **Parameters**, elija el botón **add** y escriba el nombre del parámetro



Autoria Propia

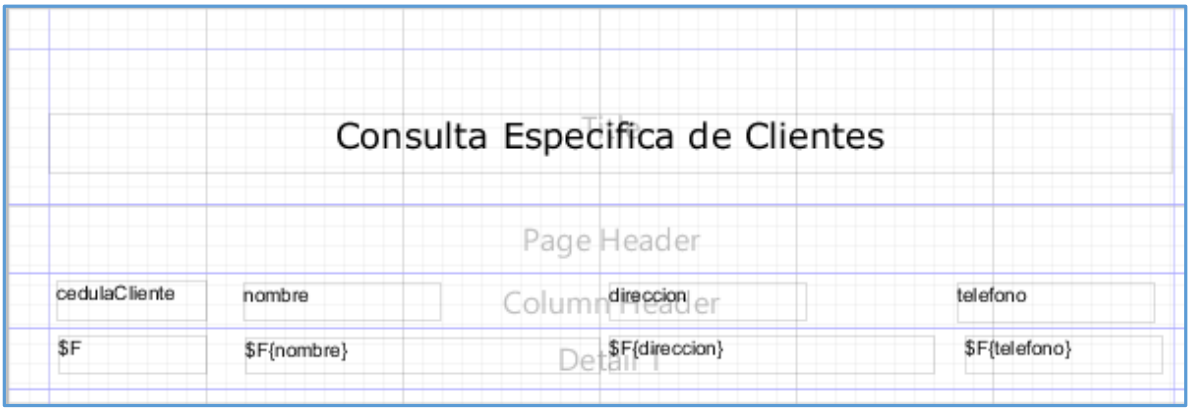
VINCULACIÓN DE UN PARÁMETRO A LA SENTENCIA SQL



Autoria Propia

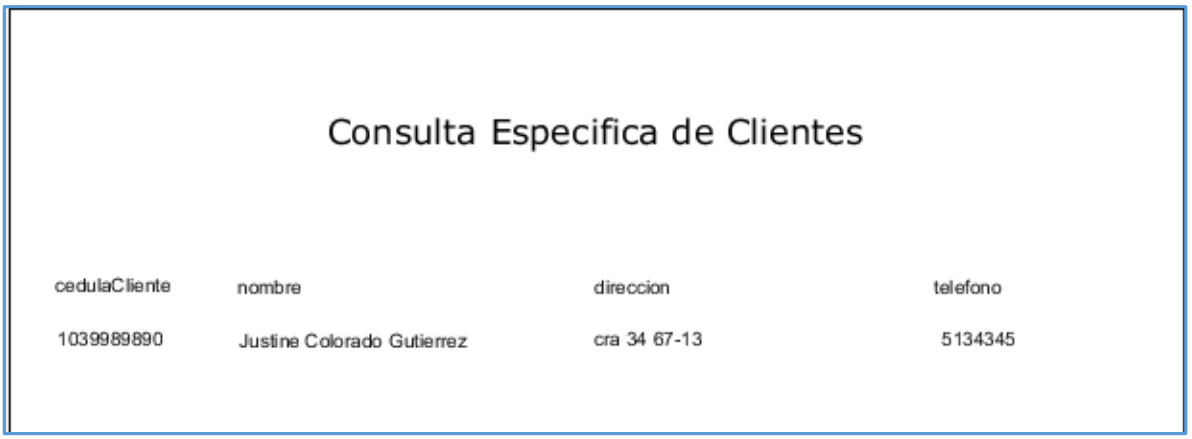
La sentencia se complementa con el parámetro, para este se utilizan los símbolos $P{parámetro}

Después de aplicar el botón **Read Fields** mostrara los campos en la parte inferior y esta listo el proceso para el diseño.



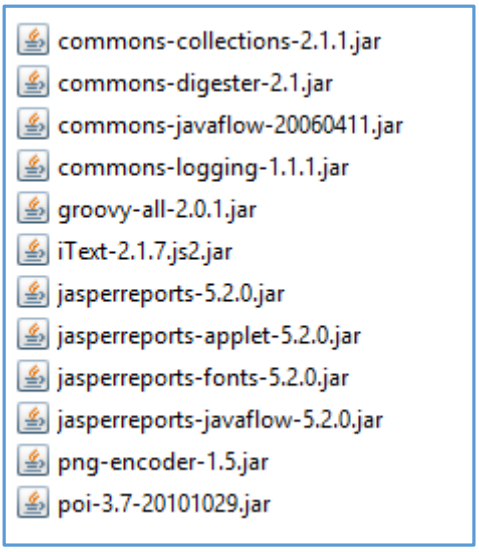
Autoria Propia

DATO CONSULTADO



Autoria Propia

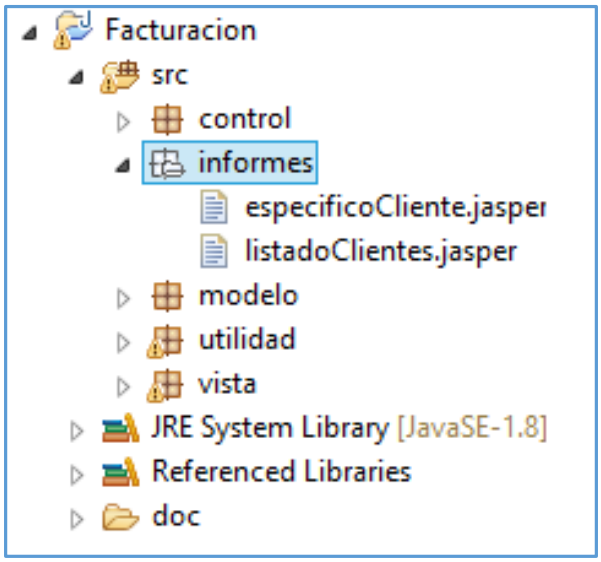
Antes de vincular los reportes al proyecto de Facturación se debe tener presenta un requerimiento importante, adicionar las librerías que acompañen este proceso, las librerías o archivos .jar que se requieren son



Autoria Propia

Puede crear una carpeta dentro del proyecto donde los almacene y luego se vinculan mediante la opción **Build Path**

Cree un paquete informes y arrastre los archivos jasper a este paquete



Autoria Propia

- CREACIÓN DE UNA CLASE INFORMES EN ESTE MISMO PAQUETE

- CONFIGURACIÓN DE LA CLASE INFORMES

```
import java.net.URL;
import java.sql.Connection;
import java.util.HashMap;
import java.util.Map;

import javax.swing.JOptionPane;

import utilidad.Conexion;
import net.sf.jasperreports.engine.JasperFillManager;
import net.sf.jasperreports.engine.JasperPrint;
import net.sf.jasperreports.engine.JasperReport;
import net.sf.jasperreports.engine.util.JRLoader;
import net.sf.jasperreports.view.JasperViewer;
```

Autoria Propia

- MÉTODO DE LLAMADO DE UN INFORME GENERAL

```
public void informeCliente  ()  {

    try {

        URL  ruta = this.getClass().getResource("listadoClientes.jasper");
        JasperReport archivo = (JasperReport) JRLoader.loadObject( ruta );
        JasperPrint imprimir = JasperFillManager.fillReport(archivo, null, Conexion.getConnection());
        JasperViewer verReporte = new JasperViewer(imprimir, false);
        verReporte.setVisible(true);
    }
    catch (Exception e) {

        JOptionPane.showMessageDialog(null,"Se produjo un error al leer el archivo .jasper" + e);
    }
}
```

Autoria Propia

MÉTODO DE LLAMADO DE UN INFORME CON PARÁMETROS

```
public void especificoClientes (String cedula)  {

    try {

        URL  ruta = this.getClass().getResource("especificoCliente.jasper");
        JasperReport archivo = (JasperReport) JRLoader.loadObject( ruta );
        Map parametros = new HashMap();
        parametros.clear();
        parametros.put("_cedulaCliente", cedula);
        JasperPrint imprimir = JasperFillManager.fillReport(archivo, parametros, Conexion.getConnection());
        JasperViewer verReporte = new JasperViewer(imprimir, false);
        verReporte.setVisible(true);
     }
    catch(Exception e){

        JOptionPane.showMessageDialog(null,"Se produjo un error al leer el archivo .jasper" + e);
    }
}
```

Autoria Propia

El informe especifico podrá ser llamado desde el formulario de cliente y el general desde el menú principal, para el primero bastará con consultar una cedula y luego seleccionar el botón de informe, tenga presente instanciar la clase informe y la invocación del método que requiere.

PISTAS DE APRENDIZAJE

Informe : Es un archivo complementario al aplicativo que permite la visualización de datos en papel o mediante un documento electrónico.

Jasper : Herramienta que permite la creación y edición de reportes

Vinculación : Permite que un aplicativo externo se una al trabajo de otro

Compilar : Creación de un nuevo archivo, este archivo no es editable

4.5 TEMA 4 DOCUMENTACIÓN

Desarrollar aplicativos no comprende solamente el diseño de formularios, validaciones, y los procesos de funcionamiento del aplicativo, es mucho más que esto, hay que tener presente la ingeniería de requerimientos,

el levantamiento de datos, el estudio pormenorizado de lo que se pretende hacer y cómo hacerlo, además de los manuales para el usuario final y la documentación o los manuales para el programador.

Java tiene una particularidad muy interesante que se denomina **documentación**, consiste en que a medida que se realiza el código del aplicativo se puede realizar dicha documentación y al final este se compila y tendríamos un formato de gran ayuda para el control del aplicativo.

Para generar este tipo de ayudas del programador, hay que tener muy presente a quien va dirigido este proceso, desarrollares y afines, son procesos muy técnicos que deberían de ubicarse en todos los procesos que comprenden el desarrollo

- IDENTIFICACIÓN DE UN COMENTARIO Y / O DOCUMENTACIÓN

En los aplicativos de desarrollo cercanos al C++ como pueden ser JavaScript, entre otros coinciden en algunos símbolos como son

// que permite el comentario de una línea o anular una línea de código

/*

Este otro símbolo permite el comentario o anulación de múltiples líneas de código

*/

/**

Esta instrucción a pesar de que es similar al anterior se utiliza para representar comentarios

*/

- IDENTIFICADORES DE LOS COMENTARIOS

@author

Especifica el autor (es) del aplicativo, modulo o fragmento de código

@version

Versión del aplicativo, módulo o fragmento de código

@see

Indica que referencias a otras clases o métodos existen

@param

Nombre de parámetro y descripción de uso y significado

@return

Describe lo que se devuelve

@exception

Nombre de la excepción que se está utilizando y excepción que puede lanzarse

@throws

Nombre de la excepción que se está utilizando y excepción que puede lanzarse

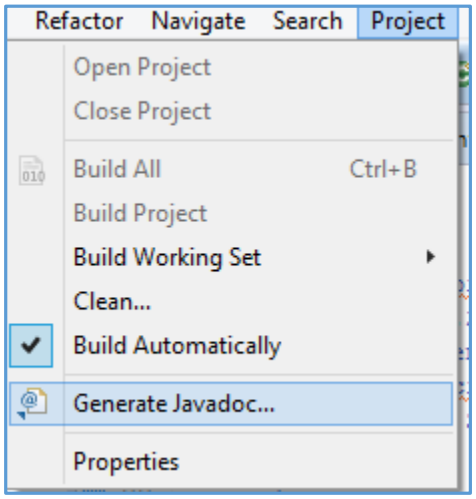
Ejemplo

```
/**
 * @author Pepito Perez
 * @version 1.2
 * @param cliente
 * @return vacio
 * @exception SQLException si la sentencia SQL es incorrecta mostratra el mensaje de excepcion
 *
 */
public static void modificar (Cliente cliente) {

    try {

        PreparedStatement registroCliente = cnn.prepareStatement("call modificarCliente (?, ?, ?, ?, ?)");
        registroCliente.setString(1, cliente.getCedulaCliente());
        registroCliente.setString(2, cliente.getNombre());
        registroCliente.setString(3, cliente.getDireccion());
        registroCliente.setString(4, cliente.getTelefono());
        registroCliente.setString(5, cliente.getEmail());
        registroCliente.executeUpdate ();
        JOptionPane.showMessageDialog(null, "Registro Actualizado");
    }
    catch (SQLException sqle) {

        sqle.printStackTrace ();
    }
}
```

Autoria Propia

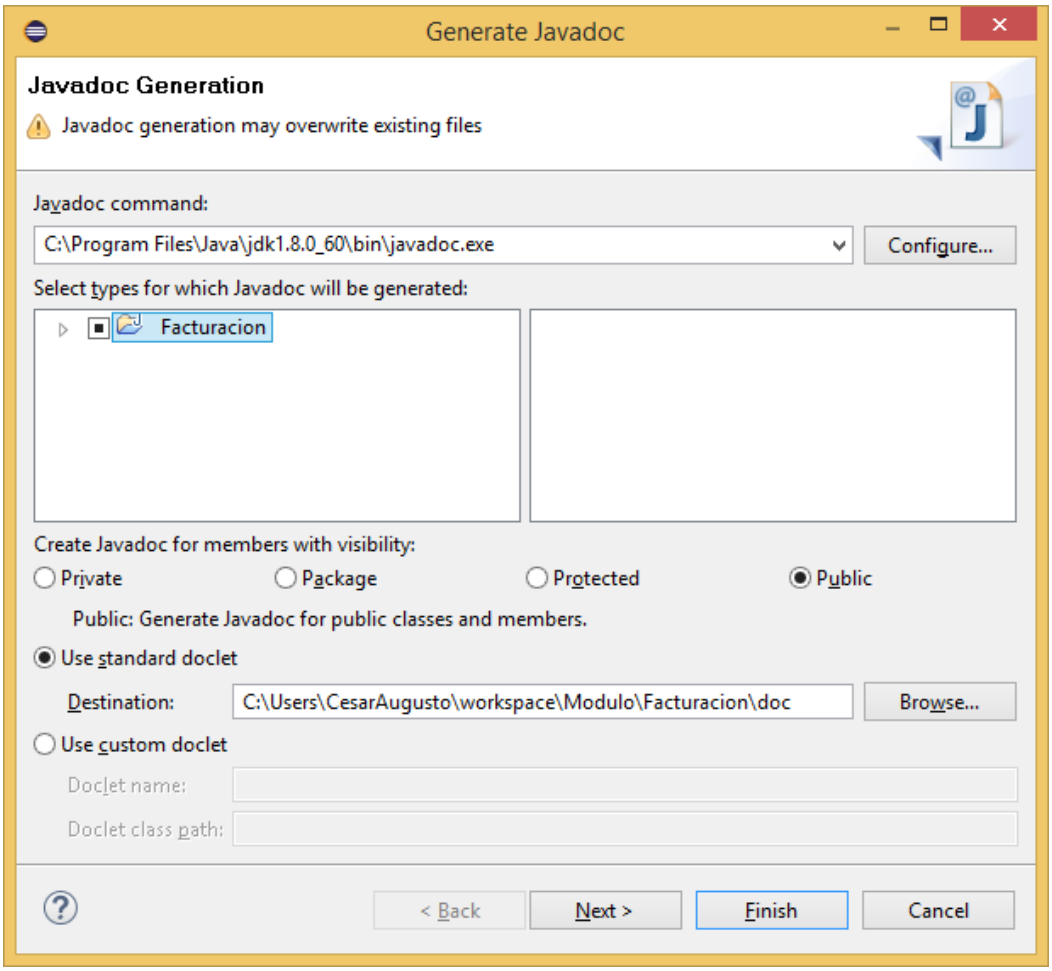
Se genera una documentación básica para uno de los métodos del archivo DAO, se especifica el autor, la versión, el parámetro, el retorno y una excepción, hay que tener presente que la documentación no debería de ser opcional, todos los procesos que se hagan deben de llevarlo con el fin de controlar y asegurar un funcionamiento optimo.

Cuando se realiza el proceso de documentación, se procede a la implementación de este.



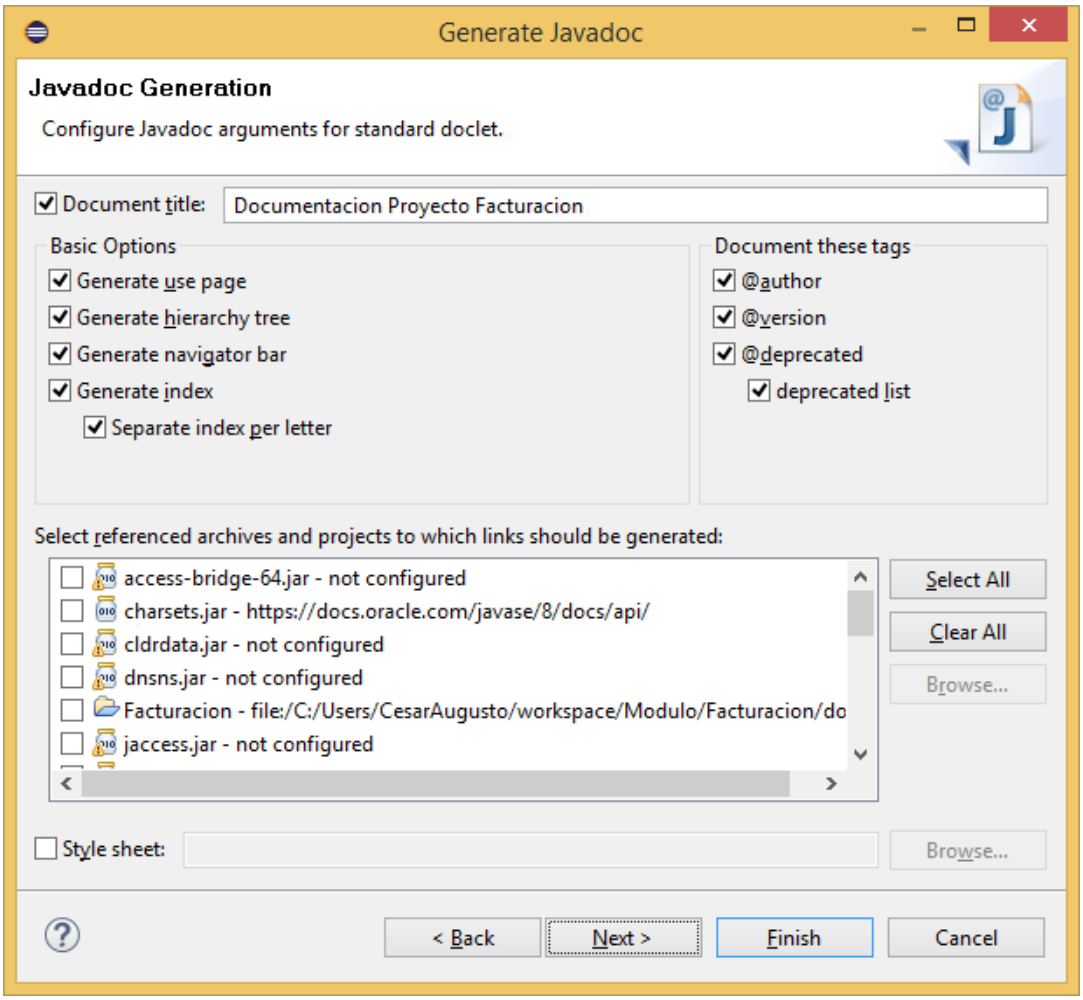
Autoria Propia

En el menú principal se selecciona la opción **Project** y la opción **Generate Javadoc…**



Autoria Propia

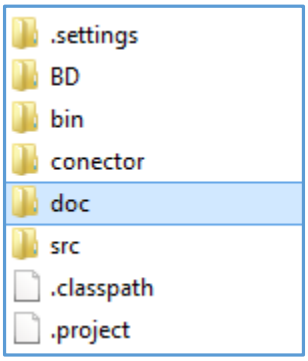
Se especifica el archivo que creara la documentación, **javadoc.exe** que se ubica en la carpeta **bin del JDK**



Autoria Propia

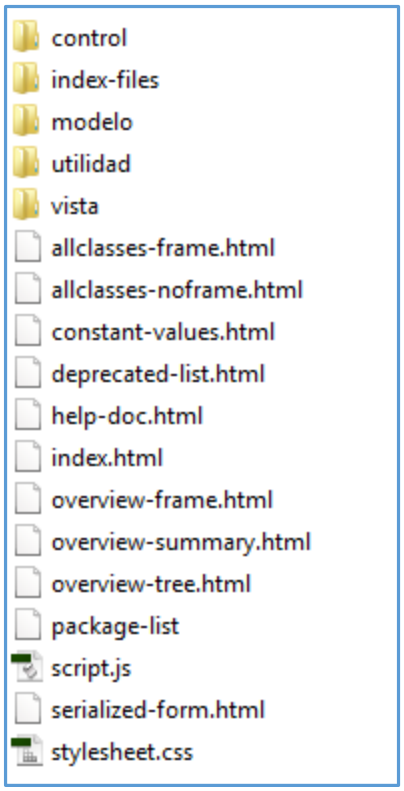
Se da un título de trabajo, y se podrá asignar un archivo de hoja de estilos si no se desea trabajar con el formato por defecto del java y luego el botón Finish

El sistema se creará con un formato HTML dentro del proyecto que se está trabajando o en otra ruta si esta se especificó previamente.



Autoria Propia

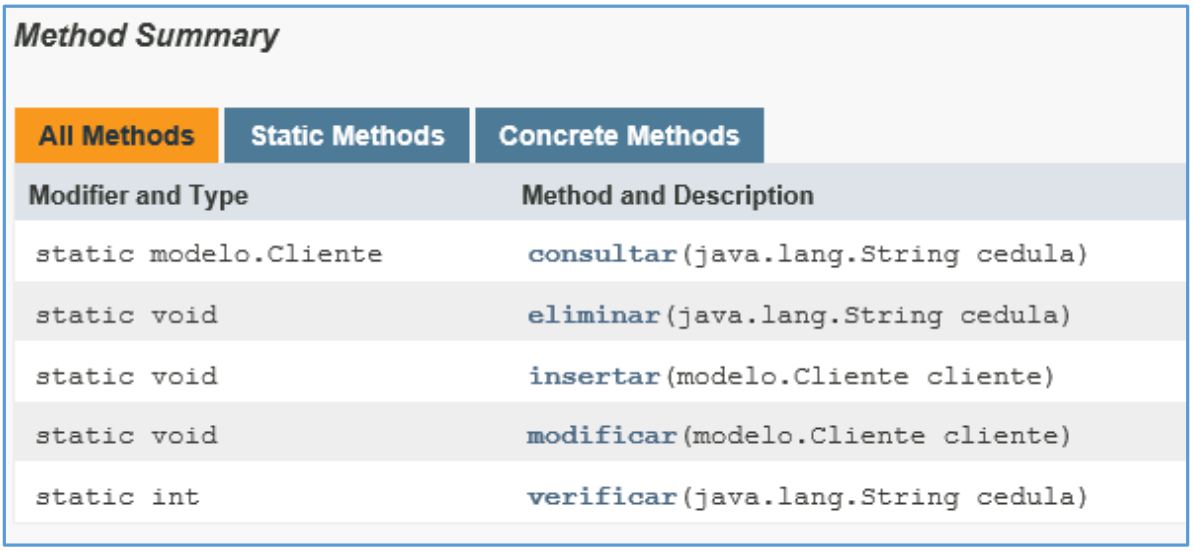
Dentro de la carpeta **doc** encontramos



Autoria Propia

Y podemos ejecutar el archivo index.html

Encontramos algunos apartes como el siguiente



Method Summary

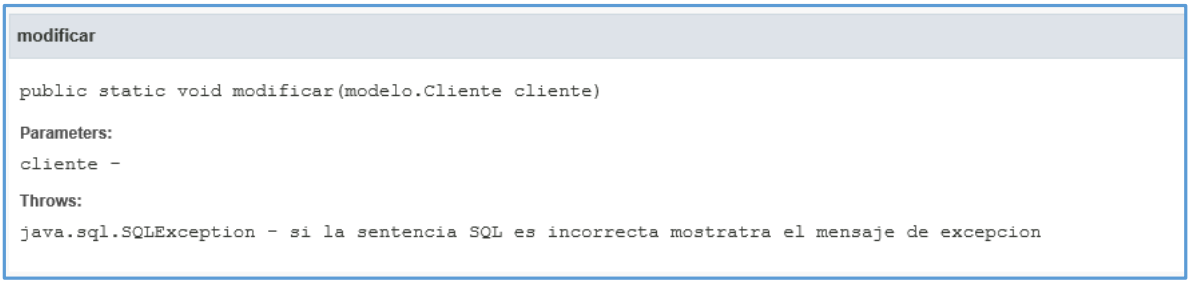
All Methods | Static Methods | Concrete Methods

Modifier and Type	Method and Description
static modelo.Cliente	consultar(java.lang.String cedula)
static void	eliminar(java.lang.String cedula)
static void	insertar(modelo.Cliente cliente)
static void	modificar(modelo.Cliente cliente)
static int	verificar(java.lang.String cedula)

Autoria Propia

En el aparecen los métodos utilizados, el nombre y el parámetro que recibe

Y algunos apartes de lo que se especificó en el método



modificar

```
public static void modificar(modelo.Cliente cliente)
```

Parameters:
cliente –

Throws:
java.sql.SQLException – si la sentencia SQL es incorrecta mostratra el mensaje de excepcion

Autoria Propia

> Es de vital importancia del manejo de esta herramienta en aplicaciones que involucran varios desarrolladores o que exista la posibilidad de que otras personas lleguen a manejarlo, el trabajo en equipo es de gran utilidad, y esta herramienta facilita esta tarea.

PISTAS DE APRENDIZAJE

Documentación **Herramienta fundamental para llevar un historial de trabajo sobre una herramienta, además de seguimiento y control de los procesos realizados.**

Manual del Usuario **Herramienta impresa o digital que brinda al usuario el paso a paso del funcionamiento del aplicativo**

Manual del Programador **Herramienta impresa o digital que facilita el seguimiento del aplicativo pasando por el autor, parámetros, métodos, excepciones, permite a los demás desarrolladores identificar el porqué de las cosas.**

4.5.1 EJERCICIO DE APRENDIZAJE

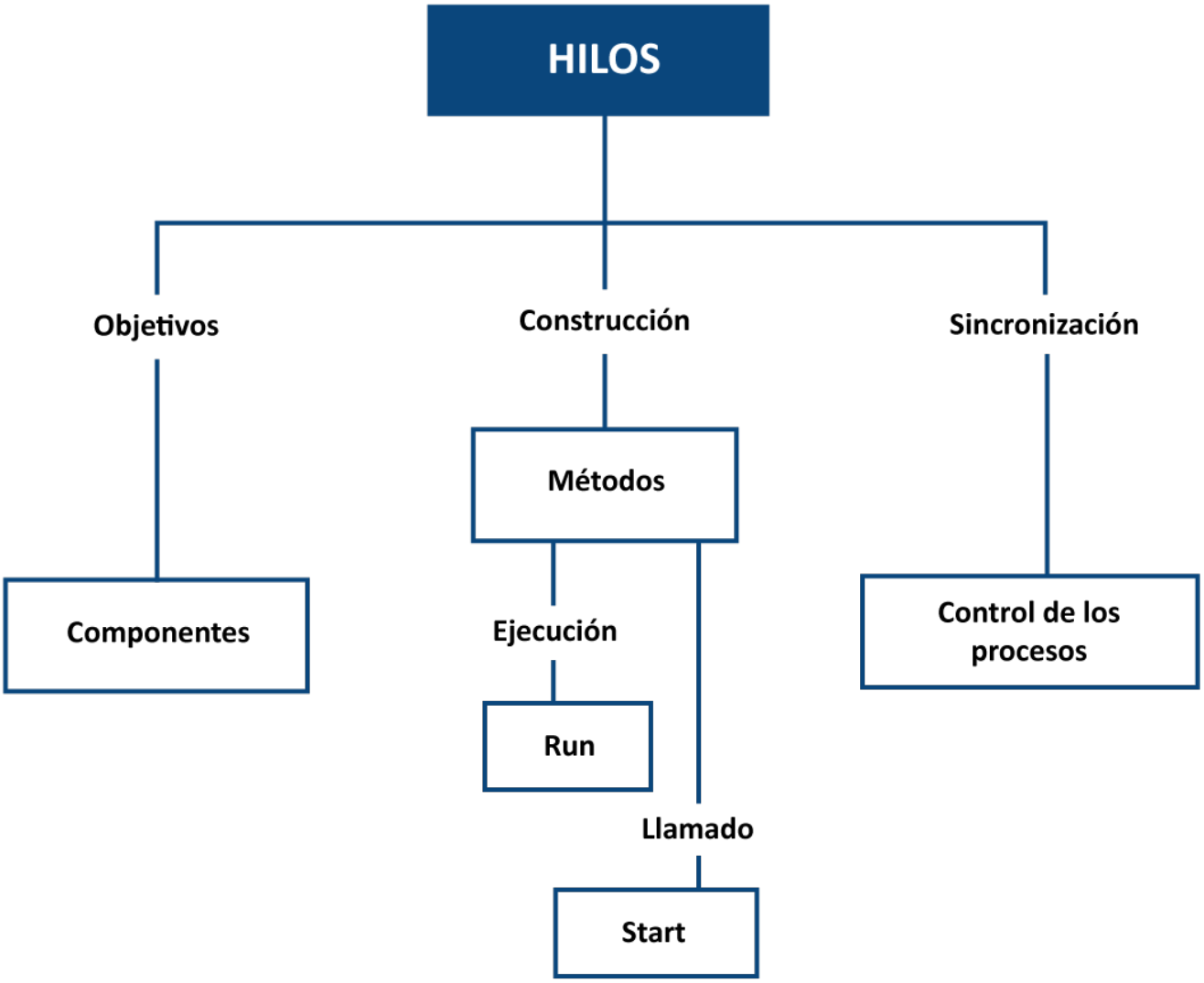
Nombre del taller de aprendizaje: CRUD	**Datos del autor del taller:** Cesar Augusto Jaramillo Henao
Escriba o plantee el caso, problema o pregunta: Se realiza un CRUD aplicando distintos modelos, estructuras y patrones de diseño, identifique que capas se utilizan y que patrones se aplican.	
Solución del taller: Las capas utilizadas son Modelo, Vista y Control, los patrones que se aplican son Facade, DAO	

4.5.2 TALLER DE ENTRENAMIENTO

Nombre del taller: Hospital	**Modalidad de trabajo:**
Actividad previa: Realice con detalle el ejercicio planteado en la primera unidad	
Describa la actividad: Cree un modelo relacional del funcionamiento básico de un hospital, en el que tome elementos esenciales de paciente, medico, enfermedades, medicamentos, incapacidades, cirugías, aplique este modelo en el lenguaje java con los conceptos de reportes, documentación, pool de conexiones	

5 UNIDAD 3 HILOS

5.1 RELACIÓN DE CONCEPTOS



Componentes: Son los elementos que permiten realizar la utilización de los hilos

Métodos: Espacio de trabajo en código para realizar distintas tareas lógicas

Run: Método principal que ejecuta un hilo

Start: Método que da inicio o llamado a un método principal

Sincronización: Organización del llamado de los procesos

OBJETIVO GENERAL

Aprender a utilizar los Hilos, herramienta fundamental en la POO, las características de aprovechamiento del reloj del sistema y las características del procesador.

OBJETIVOS ESPECÍFICOS

- Identificar los principales componentes del manejo de hilos
- Identificar que es un hilo
- Desarrollar habilidades de la ubicación de los hilos y que ventajas prestas en la etapa de programación.

5.2 TEMA 1 DEFINICIÓN Y OBJETIVOS

En los aplicativos que se han desarrollado hasta esa etapa están diseñados de forma secuencial (línea por línea), en muchas ocasiones se encontraran que este tipo de desarrollo no es el más útil o más aconsejable por la lentitud que puede generar y la poca efectividad que mostrara, para esta etapa se pasara a un tema de concurrencia o de procesos paralelos llamados hilos (**threads**).

PISTAS DE APRENDIZAJE

Que es hilo **los hilos son clase que permiten optimizar los procesos tradicionales en algo más eficiente**

Optimizar **los hilos permiten que se hagan tareas con prioridades o sin ellas, pero buscando siempre la mejoría en las tareas.**

5.3 TEMA 2 COMPONENTES

Existe una gran cantidad de componentes de los hilos, pero dentro de los más comunes para esta tarea son

Start() inicia la ejecución de un hilo, este se ubica en el run

Run() método principal del hilo

Runnable () implementación de la interfaz

Thread es una clase padre de la que dependerá nuestro hilo

PISTAS DE APRENDIZAJE

Componentes son los elementos que permiten que un hilo trabaje tan amplio o tan limitado como nosotros deseemos

Métodos son los bloques lógicos que permiten contener cada proceso que se desea ejecutar

5.4 TEMA 3 IMPLEMENTACIÓN DE LA INTERFAZ RUNNABLE

En java los hilos heredan de la clase Thread, de esta hereda el método run, en este método es donde se debe de programar el proceso que deseamos.

Dentro de este método se encontrará un método **sleep** que representa dormir o esperar para ejecutar otro proceso.

Para el ejemplo se mostrarán 5 nombres y mediante la función **random** se generará un tiempo de espera entre 0 y 9999 milisegundos

```
package contruccion;

public class EjemploHilo extends Thread{

    private String nombre;

    public EjemploHilo (String nombre) {

        this.nombre = nombre;
    }

    public void run () {

        try {

            int tiempoEspera = (int) (Math.random()*10000);
            Thread.sleep(tiempoEspera);
            System.out.print ("\nEl Nombre es: " + nombre +" el tiempo de espera es " + tiempoEspera + " milesegundos");
        }
        catch (Exception ex) {

            ex.printStackTrace();
        }
    }
```

Autoria Propia

```
    public static void main (String args []) {

        EjemploHilo ejemploHilo1 = new EjemploHilo ("Lina");
        EjemploHilo ejemploHilo2 = new EjemploHilo ("Miguel");
        EjemploHilo ejemploHilo3 = new EjemploHilo ("Johana");
        EjemploHilo ejemploHilo4 = new EjemploHilo ("Isabel");
        EjemploHilo ejemploHilo5 = new EjemploHilo ("Jose");

        ejemploHilo1.start();
        ejemploHilo2.start();
        ejemploHilo3.start();
        ejemploHilo4.start();
        ejemploHilo5.start();
    }
}
```

Autoria Propia

El proceso principal se ejecuta desde el método run que es el principal para esta tarea particular, en el void main se especifican 5 hilos con 5 nombres y la respectiva ejecución con el método **start()** que invoca el método run(), esta tomara un valor aleatorio y mostrara la información.

RUNNABLE

Dentro de la clase Thread se implementa la interfaz Runnable que hereda del método run().

Acoplemos el ejemplo anterior a esta interfaz

```
package contruccion;

public class EjemploHilo implements Runnable{

    private String nombre;

    public EjemploHilo (String nombre) {

        this.nombre = nombre;
    }

    public void run () {

        try {

            int tiempoEspera = (int) (Math.random()*10000);
            Thread.sleep(tiempoEspera);
            System.out.print ("\nEl Nombre es: " + nombre +" el tiempo de espera es " + tiempoEspera + " milesegundos");
        }
        catch (Exception ex) {

            ex.printStackTrace();
        }
    }
```

Autoria Propia

```
    public static void main (String args []) {

        Thread ejemploHilo1 = new Thread (new EjemploHilo ("Lina"));
        Thread ejemploHilo2 = new Thread (new EjemploHilo ("Miguel"));
        Thread ejemploHilo3 = new Thread (new EjemploHilo ("Johana"));
        Thread ejemploHilo4 = new Thread (new EjemploHilo ("Isabel"));
        Thread ejemploHilo5 = new Thread (new EjemploHilo ("Jose"));

        ejemploHilo1.start();
        ejemploHilo2.start();
        ejemploHilo3.start();
        ejemploHilo4.start();
        ejemploHilo5.start();
    }
}
```

Autoria Propia

En este ejemplo la clase EjemploHilo no hereda de Thread pero si implementa la interfaz Runnable de la que aplica una sobreescritura del metodo run().

Esta nueva version del ejemplo es mas practica que la primera y mas flexible en su uso al no limitar la herencia de la clase.

PISTAS DE APRENDIZAJE

Implementación la aplicación de los hilos hacen parte de las tareas esenciales en procesos con uso de muchos recursos

Uso los hilos no son para todos los casos, solo para los que requieran de procesos de múltiples actividades.

5.5 TEMA 4 CICLO DE VIDA

Los hilos o **thread** tiene varias etapas, **las más llamativas son la instancia y la ejecución**, pero existen otras etapas hasta finalizar o morir, todo este recorrido se llama ciclo de vida.

Cuando se instancia un hilo se denomina que esta creado, cuando realizamos el llamado se aplica el método **start()**, y pasa al estado de lectura o **ready**, luego de este estado llega el running que esta administrado por el sistema operativo, existen otros como el **scheduler o yield**.

Cuando está en running está haciendo uso del procesador, el hilo puede ejecutar la última línea del método run y finalizaría la tarea programada, pero también se pueden ejecutar otros métodos como **wait o sleep** que se mencionó en el ejemplo visto anteriormente.

Si el hilo entra en el método **wait** saldrá de el cuándo se ejecute uno de los métodos **notify** o **notifyall.**

Si está dormido (**sleep**) saldrá cuando finalice el tiempo de estar dormido

PISTAS DE APRENDIZAJE

Ciclo de vida **conozca las características de que recorrido tiene un hilo desde el inicio hasta que termina**

Métodos **identifica los distintos tipos de métodos propios, sus características y cuando aplicarlos**

5.6 TEMA 5 PRIORIDADES

Dentro de las prioridades y la asignación de mayor o menor posibilidad de ejecución se determina mediante la instrucción **scheduler**, este proceso favorece al momento de asignar el tiempo del procesador, en este caso se encuentran elementos como **Thread.MIN_PROPERTY** o **Thread.MAX_PROPERTY**, estos procesos manejan unas constantes que van de 1 a 10

Ejemplo

```
package contruccion;

public class EjemploHilos2 {

    public static void main (String args []){

        Hilo hilo1 = new Hilo ("Miguel");
        Hilo hilo2 = new Hilo ("Luna");

        hilo1.setPriority(Thread.MAX_PRIORITY);

        System.out.print (Thread.MAX_PRIORITY);
        System.out.print (Thread.MIN_PRIORITY);

        hilo1.start();
        hilo2.start();
    }

    static class Hilo extends Thread {

        String nombre;

        public Hilo (String nombre) {

            this.nombre = nombre;
        }
```

Autoria Propia

```
        public void run () {

            for (int i = 0; i < 5; i++) {

                System.out.print ("\n " + nombre + "\t" + i);
                yield();
            }
        }
    }
}
```

Autoria Propia

Obtendrá un resultado así

```
101
 Miguel 0
 Miguel 1
 Luna   0
 Miguel 2
 Luna   1
 Miguel 3
 Luna   2
 Miguel 4
 Luna   3
 Luna   4
```

Autoria Propia

Haga la prueba con un **MIN_PROPERTY** y vera los cambios en las prioridades

PISTAS DE APRENDIZAJE

Prioridades **identifique que prioridades puede asignar y cuando debe de utilizarlas.**

5.7 TEMA 6 SINCRONIZACIÓN

En muchas ocasiones se presentarán situaciones en las que un hilo intente acceder a los recursos de otro, esto puede ocasionar problemas porque podría afectar el resultado definitivo, si un hilo está haciendo una suma y otro una división los resultados probablemente no serán los esperados, es acá donde se procede a realizar un proceso de sincronización de tareas.

La sentencia utilizada para este tipo de caso es **synchronized** y permitirá que los métodos estén relacionados con el reloj del procesador.

Un ejemplo de un método sincronizado seria el siguiente.

```
public synchronized void poner (char c) throws Exception {

    while (lleno) {

        wait ();
    }

    buffer [tope++] = c;
    vacio = false;
    lleno = tope >= buffer.length;

    notifyAll ();
}
```

Autoria Propia

PISTAS DE APRENDIZAJE

Sincronización determine cuando sincronizar y que tareas puede aplicar para este tipo de casos, también existe la posibilidad de que el hilo sea asíncrono.

5.7.1 EJERICICIO DE APRENDIZAJE

Nombre del taller de aprendizaje: componentes	**Datos del autor del taller:** Cesar Augusto Jaramillo
Escriba o plantee el caso, problema o pregunta: Considera que trabajar con hilos puede mejorar el rendimiento para aplicaciones simples o solo para aplicaciones con gran carga de procesos	

Solución del taller: En un trabajo simple se pueden aplicar hilos, aunque no se verá tan representado, cuando un aplicativo tiene gran cantidad de procesos será más significativo y de mayor uso él trabaja con hilos.

5.7.2 TALLER DE ENTRENAMIENTO

Nombre del taller: Primos	**Modalidad de trabajo:** Individual
Actividad previa: Elaborar un aplicativo que calcule numero primos	
Describa la actividad: Imprimir los primeros 500 números primos, calcular con y sin hilos y compare la diferencia en tiempo	

6 PISTAS DE APRENDIZAJE

Recuerde que: programación web es un recurso muy amplio que maneja múltiples lenguajes y elementos

Tenga en cuenta: la programación utilizada es basada en java tanto para java SE como para Java EE

Traiga a la memoria: que la mayor parte de comando e instrucciones son los mismos en java SE que en java EE

7 GLOSARIO

- Java SE : es la versión estándar de java, esta versión es la base de todo el trabajo en java
- Java EE : es la versión Enterprise o empresarial, es utilizada para la programación web
- Eclipse : es un IDE de desarrollo que permite facilitar algunas tareas de la programación en Java
- Proyecto: es un conjunto de archivos que componen una aplicación
- Paquete: es un área de trabajo que permite la clasificación de archivos o clases
- DAO: es un modelo de desarrollo o patrón de diseño, standard de trabajo
- Getters/setters hacen parte de una clase principal que permite accede a la información
- Método: espacio de código que realiza una funciona especifica
- Façade : patrón de diseño que administra un conjunto de clases
- Hibernate: FrameWork de java que permite realizar procesos standard o web de una forma mas simplificada

8 BIBLIOGRAFÍA

- Eckel, Bruce. (2008). Piensa en Java, Madrid. ISBN: 978-84-8966-034-2
- Villalobos, Jorge (2006), Fundamentos de Programación, Bogotá. ISBN: 970-26-0846-5
- Deitel, Paul. (2012), Java, como programar, México. ISBN: 978-607-32-1150-5