# Ejemplo Modelo Vista Controlador:

Se solicita desarrollar un sistema de administración de usuarios con un **CRUD**(Create, Read, Update, Delete) básico nos advierten que la aplicación es un prototipo inicial el cual esta propenso a cambios, pues aún no se tienen definidas las ventanas con las que se va a trabajar ni la información requerida, por le momento se trabajará con un BD **MySql**, pero posiblemente se tenga que migrar a otro sistema gestor.

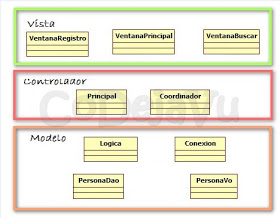
# La solución:

Se aplica el **MVC**permitiendo independizar la lógica y la parte visual del sistema usando para eso un controlador que administra los procesos sirviendo como puente entre estos.

Para la BD se usa MySql, reutilizando la BD creada en el ejemplo de conexión con Java, el Scritp es el siguiente.

|  |
| --- |
| CREATE DATABASE  USE `code`;  /\*Table structure for table `persona` \*/  DROP TABLE IF EXISTS `persona`;  CREATE TABLE `persona` (  `id` int(10) NOT NULL,  `nombre` varchar(30) default NULL,  `edad` int(3) default NULL,  `profesion` varchar(30) default NULL,  `telefono` decimal(10,0) default NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=latin1;  /\*Data for the table `persona` \*/ |

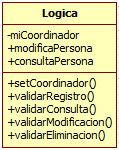
De esta forma se crea la Base de Datos a la que se va a apuntar, la cual contiene solo una tabla **Persona**con la que se realizan todas las operaciones **CRUD**.  
  
La agrupación de clases de nuestro sistema es:



Como vemos tenemos la parte visual compuesta por las clases VentanaPrincipal, VentanaRegistro, VentanaBuscar, la lógica de negocio se establece en el modelo el cual se compone por la clase Lógica y Conexión y aplicamos los patrones DAO y VO por medio de las clases PersonaDao y PersonaVo y por último el controlador se define en la clase Coordinador, encargada de establecer todas las relaciones del sistema, la clase Principal prepara las instancias iniciales del sistema y posteriormente lo ejecuta.

# El Modelo:

Como se mencionó, en el modelo tenemos la lógica de negocio, serán todas las clases vinculadas con el CRUD a nivel interno, que en ultimas es en lo que gira nuestro sistema.  
  
**Clase Lógica.**

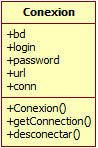


Esta clase permite realizar las operaciones asociadas a la lógica de negocio como tal, desde ella realizamos las validaciones y llamados a las operaciones CRUD del sistema.

En caso de que se requieran procesos adicionales asociados a la lógica de negocio, aquí será donde se creen los métodos para dichos procesos, por ejemplo, el método validarRegistro determina si los datos son correctos y permite registrar la persona en el Dao.

|  |
| --- |
| public void validarRegistro(PersonaVo miPersona) {  PersonaDao miPersonaDao;  /\*Valida que solo se ingresen id de 3 digitos\*/  if (miPersona.getIdPersona() > 99) {  miPersonaDao = new PersonaDao();  miPersonaDao.registrarPersona(miPersona);  }else {  JOptionPane.showMessageDialog(null,"El documento de la persona debe" +  " ser mas de 3 digitos","Advertencia",JOptionPane.WARNING\_MESSAGE);  }  } |

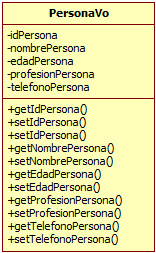
**Clase conexión.**



En esta clase tenemos la cadena de conexión con la que trabajará nuestra aplicación, en ella se define la base de datos, el usuario, password y driver de conexión, si por ejemplo en un futuro se nos pide conectarnos a una base de datos diferente o establecer un sistema gestor distinto (pero con la misma estructura de tablas y campos), tan solo modificaremos esta clase y dicho cambio será transparente para el resto del sistema.

|  |
| --- |
| public Conexion() {  try{  //obtenemos el driver de para mysql  Class.forName("com.mysql.jdbc.Driver");  //obtenemos la conexión  conn = DriverManager.getConnection(url,login,password);  if (conn!=null){  System.out.println("Conección a base de datos "+bd+" OK");  }  }  catch(SQLException e){  System.out.println(e);  }catch(ClassNotFoundException e){  System.out.println(e);  }catch(Exception e){  System.out.println(e);  }  } |

**Clase PersonaVo.**

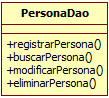


Al utilizar este tipo de clases, aplicamos el patrón Value Object o VO (Anteriormente conocidas como DTO Data Transfer Object) en el que representamos las entidades (Tablas) de la base de datos, la tabla persona tiene los campos id, nombre, edad, profesión teléfono entonces nuestra clase Vo tendrá estos mismos atributos y de esta manera podremos transportar un objeto persona con todos estos valores por medio de los métodos set y get de cada atributo.

Este patrón nos facilita enormemente el transporte de la información, evitando que se envíen gran cantidad de parámetros a un método cuando queremos hacer un registro o actualización, también en caso de que se modifique la tabla del BD, nuestra clase deberá ser modificada, así como los métodos que obtienen la información, mas no los métodos que la transportan.

|  |
| --- |
| /\*\*  \* @return the idPersona  \*/  public Integer getIdPersona() {  return idPersona;  }  /\*\*  \* @param idPersona the idPersona to set  \*/  public void setIdPersona(Integer idPersona) {  this.idPersona = idPersona;  } |

**Clase PersonaDao.**



Cuando utilizamos estas clases, aplicamos el patrón Data Access Object o DAO, básicamente este patrón consiste en centralizar los procesos de acceso a la base de datos evitando inconsistencias y posibles problemáticas cuando esto se realiza a lo largo de la aplicación.

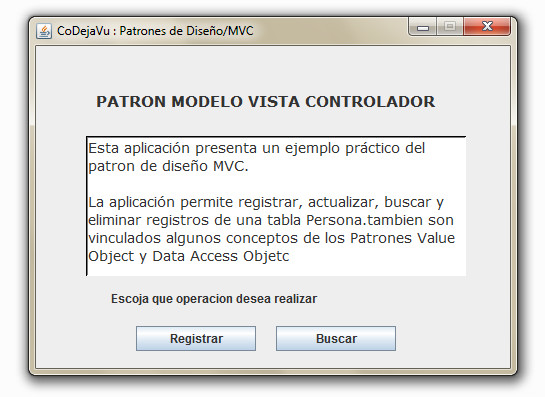
Con este patrón independizamos la lógica de negocio de la lógica de acceso a datos obteniendo mayor organización y flexibilidad en el sistema.

|  |
| --- |
| public void eliminarPersona(String codigo)  {  Conexion conex= new Conexion();  try {  Statement estatuto = conex.getConnection().createStatement();  estatuto.executeUpdate("DELETE FROM persona WHERE id='"+codigo+"'");  JOptionPane.showMessageDialog(null, " Se ha Eliminado" +  " Correctamente","Información",JOptionPane.INFORMATION\_MESSAGE);  estatuto.close();  conex.desconectar();    } catch (SQLException e) {  System.out.println(e.getMessage());  JOptionPane.showMessageDialog(null, "No se Elimino");  }  } |

**La vista.**

Aquí se define la parte visual del sistema, en la vista estableceremos todas las ventanas o interfaces gráficas de usuario, mediante las cuales representamos todo el modelo permitiendo la interacción entre la aplicación y el cliente.

**Clase Ventana principal.**



Esta clase representa la ventana inicial de la aplicación, posee un área de texto con una pequeña descripción y 2 botones que darán inicio a los eventos principales del sistema.

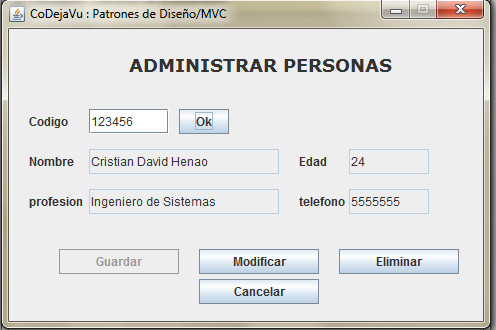
Se comunica con el modelo mediante la clase coordinador y desde está se cargan las otras ventanas de la aplicación.

**Clase VentanaRegistro.**



Permite el ingreso de información a la BD, en ella se evidencian todos los campos de la tabla Persona y presenta un medio gráfico para las clases PersonaVo y PersonaDao, al ejecutar el evento de registro se establece comunicación entre el coordinador y la clase Lógica donde se encuentran todas las validaciones y casos para el ingreso de información.

**Clase VentanaBuscar.**



Esta clase permite realizar las operaciones de Consulta, Actualización y eliminación de la tabla Persona, igual que la anterior representa la parte visual del modelo y se ejecutan los eventos para la lógica de los procesos anteriores, la forma de obtener los datos y como se le envían parámetros de consulta a la BD.

**El controlador.**

Esta parte del patrón es la que define la lógica de administración del sistema, establece la conexión entre la vista y el modelo.

**Clase principal.**

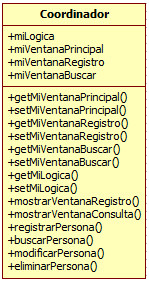


Esta clase contiene el método main que ejecuta la aplicación, el método hace un llamado al método iniciar el cual crea las instancias de las clases ventanas y la clase Lógica estableciendo las relaciones con la clase Coordinador.

A cada instancia de las clases se les envía una instancia de la clase Coordinador, y a la instancia de Coordinador se le envía cada instancia de las clases, esto por medio de los métodos set y get estableciendo las relaciones necesarias y por último se usa la instancia de la clase VentanaPrincipal para cargarla en pantalla.

|  |
| --- |
| private void iniciar() {  /\*\*Se instancian las clases\*/  miVentanaPrincipal=new VentanaPrincipal();  miVentanaRegistro=new VentanaRegistro();  miVentanaBuscar= new VentanaBuscar();  miLogica=new Logica();  miCoordinador= new Coordinador();  /\*\*Se establecen las relaciones entre clases\*/  miVentanaPrincipal.setCoordinador(miCoordinador);  miVentanaRegistro.setCoordinador(miCoordinador);  miVentanaBuscar.setCoordinador(miCoordinador);  miLogica.setCoordinador(miCoordinador);    /\*\*Se establecen relaciones con la clase coordinador\*/  miCoordinador.setMiVentanaPrincipal(miVentanaPrincipal);  miCoordinador.setMiVentanaRegistro(miVentanaRegistro);  miCoordinador.setMiVentanaBuscar(miVentanaBuscar);  miCoordinador.setMiLogica(miLogica);    miVentanaPrincipal.setVisible(true);  } |

**Clase coordinador.**



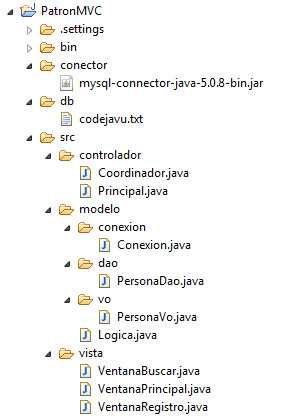
Esta clase contiene toda la lógica de relaciones en el aplicativo, es el puente entre el modelo y las vistas.

Puede contener instancias locales tanto de clases de la vista como de clases del modelo, estas instancias tienen sus respectivos métodos set y get permitiendo el flujo de llamados del sistema.

Cuando se desea registrar una persona desde la clase VentanaRegistro se hace un llamado al método registrarPersona () de la clase Coordinador, posteriormente esta clase llama al método ValidarRegistro () de la clase Lógica y esta a su vez realiza las validaciones correspondientes para determinar si se llama o no al método RegistrarPersona () de la clase PersonaDao.

|  |
| --- |
| public Logica getMiLogica() {  return miLogica;  }  public void setMiLogica(Logica miLogica) {  this.miLogica = miLogica;  }    public void mostrarVentanaRegistro() {  miVentanaRegistro.setVisible(true);  }  public void mostrarVentanaConsulta() {  miVentanaBuscar.setVisible(true);  }  public void registrarPersona(PersonaVo miPersona) {  miLogica.validarRegistro(miPersona);  } |

**Aplicación.**



Se crearon tres paquetes principales y de esta forma se tiene independiente la parte de la vista, el modelo y el controlador, el paquete modelo posee los paquetes conexión dao y vo ya que aunque pertenecen al modelo y la lógica de negocio debe establecerse también su independencia con respecto a las operaciones, es decir, las clases Vo por ejemplo no poseen métodos de operaciones asociadas a la lógica de negocio, tan solo deben representar las tablas de la BD, entonces en el paquete vo se agruparán todas las clases equivalente a entidades.

El proyecto tiene los directorios db y conector, estos directorios se crearon solamente para agregar el script de creación de la BD y el mysql connector, este último debe ser agregado al build path para que el sistema pueda funcionar (sobre el proyecto, clic derecho/configure Build path... ).

**Conclusiones.**

Como vimos tenemos un proyecto funcional que realiza las operaciones básicas del CRUD, y donde todas sus partes se encuentran separadas por funcionalidades, separando la parte visual de la lógica y estableciendo relaciones entre ellas, existen validaciones muy básicas y procesos simples pero pensado en modo de ejemplo sin mayor complejidad.