# PATRONES DE DISEÑO:

- **DAO (Data Access Object):**

  - Descripción: El patrón DAO se utiliza para encapsular la lógica de acceso a datos. En este caso, EmpleadoDAO y NominaDAO representan el acceso a los datos de las tablas de empleados y nóminas.

  - Ejemplo: En EmpleadoDAO, se observa cómo la conexión a la base de datos y las operaciones de SQL están encapsuladas.

```
public class EmpleadoDAO {

private Connection connection;
private PreparedStatement statement;
private boolean estadoOperacion;

public boolean eliminarEmpleado(String dniEmpleado) throws SQLException {
// Lógica de eliminación de empleado
}

public List<Empleado> listarEmpleados() throws SQLException {
// Lógica de consulta para obtener lista de empleados
}

}
```

Este diseño permite que la lógica de acceso a datos esté separada del resto de la lógica de la aplicación, facilitando el mantenimiento y posibles cambios en el esquema de la base de datos.

- **Front Controller:**

  - El patrón de diseño Front Controller se utiliza en aplicaciones web para centralizar las solicitudes de los usuarios. En lugar de manejar cada solicitud de manera individual en diferentes servlets, el Front Controller actúa como un punto de entrada único que recibe y gestiona todas las solicitudes. Este patrón permite realizar operaciones comunes de manera consistente (como autenticación, validación o enrutamiento) y facilita la organización de la aplicación.

    En el proyecto, la clase Controller implementa el patrón Front Controller.

    Es un servlet que centraliza todas las solicitudes relacionadas con la gestión de empleados y nóminas. Aquí, el Front Controller (Controller) toma decisiones basadas en el parámetro opcion recibido en cada solicitud. Dependiendo del valor de opcion, el servlet determina qué acción ejecutar.

    Realmente, este patrón ya estaba implícito en el proyecto antes de entrar en este tema, ya que lo había configurado desde un primer momento así.

- **Singleton:**

  - Descripción: Este patrón garantiza que haya una sola instancia de una clase, útil en la clase de conexión para evitar múltiples conexiones abiertas al mismo tiempo.

  - Ejemplo: La clase Conexion usa un BasicDataSource que sigue un estilo de patrón Singleton, ya que la conexión se configura solo una vez y se reutiliza.

```
private static BasicDataSource dataSource = null;

private static DataSource getDataSource() {
   if (dataSource == null) {
      dataSource = new BasicDataSource();
      dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
      dataSource.setUsername("user");
      dataSource.setPassword("password");
   }
   return dataSource;
}
```

Este método getDataSource asegura que solo haya una instancia de BasicDataSource, lo cual es una implementación clásica de Singleton.

- **MVC (Modelo Vista Controlador)**

  - Descripción: Este patrón se refleja en la estructura del proyecto, donde el paquete controller contiene la clase Controller.java, que actúa como controlador, mientras que model contiene las entidades de negocio como Empleado y Nomina, y webapp/_views alberga las vistas en JSP.

  - Ejemplo: En Controller.java, la lógica de control se encuentra en un servlet que recibe peticiones y decide qué vista (JSP) mostrar.

```
public class Controller extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String action = request.getParameter("action");
    if (action.equals("listar")) {
      List<Empleado> empleados = new EmpleadoDAO().listarEmpleados();
      request.setAttribute("empleados", empleados);
      RequestDispatcher dispatcher = request.getRequestDispatcher("/_views/listar.jsp");
      dispatcher.forward(request, response);
    }
  }
}
```

Este servlet centraliza las peticiones, que luego son redirigidas a las vistas adecuadas, respetando la separación de MVC.

- **Factory Method:**

  Imagina que tienes una fábrica de coches. La fábrica produce coches, pero tú no sabes exactamente qué tipo de coche va a salir: puede ser un coche de carreras, un coche familiar o un coche eléctrico. Lo único que sabes es que es un coche.

  **Factory Method** es como tener una fábrica que crea objetos (empleados) por ti, sin que tengas que hacer todo el trabajo cada vez. Así, si en el futuro cambian los detalles de cómo crear esos objetos, solo cambias la fábrica y todo el sistema sigue funcionando.

  1. **Creamos una fábrica para los empleados:**

  Imagina que en lugar de tener que crear un empleado de forma manual (como hacerlo con new Empleado()), tienes una "máquina" que crea empleados por ti. Así no tienes que preocuparte por cómo se hacen.

```
package laboral.factorymethod;

import java.sql.ResultSet;
import java.sql.SQLException;

import laboral.model.Empleado;

public abstract class EmpleadoFactory {

        // Método abstracto que cada subclase debe implementar
        public abstract Empleado crearEmpleado(ResultSet resultSet) throws
        SQLException;

}
```

  2. **La máquina concreta que hace los empleados:**

La "máquina" que realmente hace el trabajo de crear los empleados es una clase llamada EmpleadoFactoryConcreto.
Esta clase sabe cómo hacer un empleado correctamente.
Como nosotros creamos los empleados realmente desde nuestra clase EmpleadoDAO obteniéndolos por una consulta SQL para filtrarlos o listarlos (debido a las especificaciones del ejercicio), basamos la “máquina” en nuestra lógica del DAO actual (con el ResultSet).

```
package laboral.factorymethod;

import java.sql.ResultSet;
import java.sql.SQLException;
import laboral.model.Empleado;

public class EmpleadoFactoryConcreto extends EmpleadoFactory {

        // Implementación del método para crear un empleado a partir de
    un ResultSet
        @Override
        public Empleado crearEmpleado(ResultSet resultSet) throws
    SQLException {
            Empleado empleado = new Empleado();
            empleado.setNombre(resultSet.getString(1));
            empleado.setDNI(resultSet.getString(2));
            empleado.setSexo(resultSet.getString(3).charAt(0));
            empleado.setCategoria(resultSet.getInt(4));
            empleado.setAnyos(resultSet.getInt(5));

            return empleado;
        }
}
```

### 3. Uso en el controlador:

Ahora, cuando quieras crear un nuevo empleado, en lugar de hacerlo manualmente, solo llamas a la "máquina" que lo hace por ti.

Por ejemplo, para nuestro método del EmpleadoDAO sobre filtrar empleados de la base de datos, ahora podemos tenerlo de la siguiente manera.

```
public List<Empleado> filtrarEmpleados(String campo, Object valor) throws
SQLException
{
        ResultSet resultSet = null;
        List<Empleado> listaEmpleados = new ArrayList<>();

        String sql = null;
        estadoOperacion = false;
        connection = obtenerConexion();

       try {
        sql = "SELECT * FROM Empleado WHERE " + campo + " = ?";
        statement = connection.prepareStatement(sql);

        // Verificar si el valor es un String o un Integer y establecer el parámetro
        correspondiente
        if (valor instanceof String) {
                statement.setString(1, (String) valor);
        } else if (valor instanceof Integer) {
               statement.setInt(1, (Integer) valor);
        }

        resultSet = statement.executeQuery();

        while (resultSet.next()) {
                // Usamos la Factory Concreta para crear el empleado
        EmpleadoFactory factory = new EmpleadoFactoryConcreto();

        Empleado empleado = factory.crearEmpleado(resultSet);
        listaEmpleados.add(empleado);

         }
                    } catch (SQLException e) {
```

```
                    e.printStackTrace();
                }

        return listaEmpleados;

}
```

Si en el futuro necesitamos cambiar cómo se crean los empleados (por ejemplo, añadiendo nuevos datos o cambiando la forma en que se calculan los sueldos), solo necesitas cambiar la "fábrica" y no todo el resto del código.

- **State:**

El patrón **State** te permite cambiar el comportamiento de un objeto cuando cambia su estado interno, de manera que el objeto parece "cambiar de clase". Este patrón es útil en situaciones donde los objetos tienen varios estados posibles y deben comportarse de manera diferente según el estado en el que se encuentren.

En nuestro caso, podemos usarlo para gestionar el estado de un *Empleado* o de una *Nómina*.

Por ejemplo, imagina que un Empleado tiene un estado relacionado con su situación laboral, como Activo, De baja, Suspendido, etc.

Dependiendo de ese estado, el comportamiento del empleado o de la nómina puede variar.

- Si un **Empleado** está **Activo**, su nómina se calcula de acuerdo con su sueldo base y sus antigüedades.

- Si el **Empleado** está **De baja**, su nómina puede calcularse de forma diferente, como con un sueldo reducido o una bonificación especial.

- Si el **Empleado** está **Suspendido**, quizás su nómina se detiene.

Nosotros vamos a controlar este estado directamente en la clase Nómina. Para ello, también hemos añadido a la base de datos un campo adicional en la tabla Nómina llamado ESTADO.

1. **Definir el patron State en la clase Nómina.**

   Primero, vamos a crear una interfaz EstadoNomina para los diferentes estados de la nómina, tal como lo hicimos con el patrón State anteriormente.

```
public interface EstadoNomina {
        void calcularSueldo(Nomina nomina);
}
```

2. **Crear los diferentes estados para la clase Nómina.**

   Creamos 3 estados:

   - EstadoNormal:

```
package laboral.state;

import laboral.model.Empleado;
import laboral.model.Nomina;

public class EstadoNormal implements EstadoNomina {
        @Override
        public void calcularSueldo(Nomina nomina, Empleado empleado) {
                System.out.println("Sueldo calculado con estado normal: " +
        nomina.getSueldoCalculado());
        }
}
```

   - EstadoEnfermedad:

Le quitamos un 25% de su sueldo anual base.

```
package laboral.state;

import laboral.model.Empleado;
import laboral.model.Nomina;

public class EstadoEnfermedad implements EstadoNomina {
        @Override
        public void calcularSueldo(Nomina nomina, Empleado
    empleado) {

            nomina.setSueldoCalculado(nomina.getSueldoCalculado()*
    0.75);
                System.out.println("Sueldo calculado con estado
    enfermedad: " + nomina.getSueldoCalculado());
        }
}
```

- EstadoSuspendido:

  El empleado está suspendido de empleo y sueldo.

```
package laboral.state;

import laboral.model.Empleado;
import laboral.model.Nomina;

public class EstadoSuspendido implements EstadoNomina {
        @Override
        public void calcularSueldo(Nomina nomina, Empleado
    empleado) {
            nomina.setSueldoCalculado(0);
            System.out.println("Sueldo calculado con estado suspendido: "
    + nomina.getSueldoCalculado());
        }
}
```

3. **Modificamos la clase Nomina para poder controlar los estados.**

   La clase Nómina ahora tiene un estado que puede cambiar, y dependiendo de este estado, el cálculo del sueldo se realizará de forma diferente.

```
package laboral.model;

import laboral.state.EstadoNomina;
import laboral.state.EstadoNormal;

public class Nomina {
        private String dniEmpleado;
        private double sueldoCalculado;
        private EstadoNomina estado;

        public static final double[] SUELDO_BASE = {1200, 1400, 1600, 1800,
      2000, 2200, 2400, 2600, 2800, 3000};

        // Getters y setters
        public void setEstado(EstadoNomina estado) {
            this.estado = estado;
        }

        public void calcularSueldo(Empleado empleado) {
            if (estado != null) {
                estado.calcularSueldo(this, empleado);
            }
        }

        public void setSueldoCalculado(double sueldo) {
            this.sueldoCalculado = sueldo;
        }

        public double getSueldoCalculado() {
            return sueldoCalculado;
        }

        public String getDniEmpleado() {
            return dniEmpleado;
        }
```

```
    public void setDniEmpleado(String dniEmpleado) {
        this.dniEmpleado = dniEmpleado;
    }

    public String imprime() {
        return "Nómina de " + dniEmpleado + " con sueldo " +
sueldoCalculado;
    }
}
```

4. **Controlamos el estado del empleado al recoger los datos de la base de datos.**

   Ahora que tenemos la lógica implementada en la clase como tal, a la hora de recoger los datos con nuestro NominaDAO, tenemos que controlar el estado que nos llega de la base de datos y aplicarle al sueldo un estado u otro.

   Lo vamos a hacer de la siguiente forma:

```
public Nomina obtenerNomina(String dniEmpleado) throws SQLException {
        ResultSet resultSet = null;
        Nomina nomina = new Nomina();
        Empleado empleado = null;  // Variable para almacenar los
datos del empleado

        String sql = "SELECT * FROM Nomina WHERE empleado_dni = ?";
        connection = obtenerConexion();

        try {
            statement = connection.prepareStatement(sql);
            statement.setString(1, dniEmpleado);
            resultSet = statement.executeQuery();

            if (resultSet.next()) {
                // Obtener la nómina
```

```
        nomina.setDniEmpleado(resultSet.getString("empleado_dni"));

        nomina.setSueldoCalculado(resultSet.getDouble("sueldoCalculado"
        ));

                // Obtener el estado de la nómina desde la base de datos
                String estado = resultSet.getString("estado_nomina");
                switch (estado) {
                   case "Normal":
                      nomina.setEstado(new EstadoNormal());
                      break;
                   case "Enfermedad":
                      nomina.setEstado(new EstadoEnfermedad());
                      break;
                   case "Suspendido":
                      nomina.setEstado(new EstadoSuspendido());
                      break;
                   default:
                      nomina.setEstado(new EstadoNormal()); // Default a
Normal si no se encuentra el estado
                }

                EmpleadoDAO empleadoDAO = new EmpleadoDAO();

                empleado =
empleadoDAO.obtenerEmpleado(dniEmpleado);

                // Ahora que tienes el objeto empleado con la categoría,
pasa el empleado a calcular el sueldo
                if (empleado != null) {
                   nomina.calcularSueldo(empleado); // Pasa el empleado
con sus datos completos
                } else {
                   // Si no se encontró el empleado, puedes manejar el
error de alguna forma
                   System.out.println("Empleado no encontrado para el DNI:
" + dniEmpleado);
                }
             }

          } catch (SQLException e) {
             e.printStackTrace();
          }
          return nomina;
   }
```

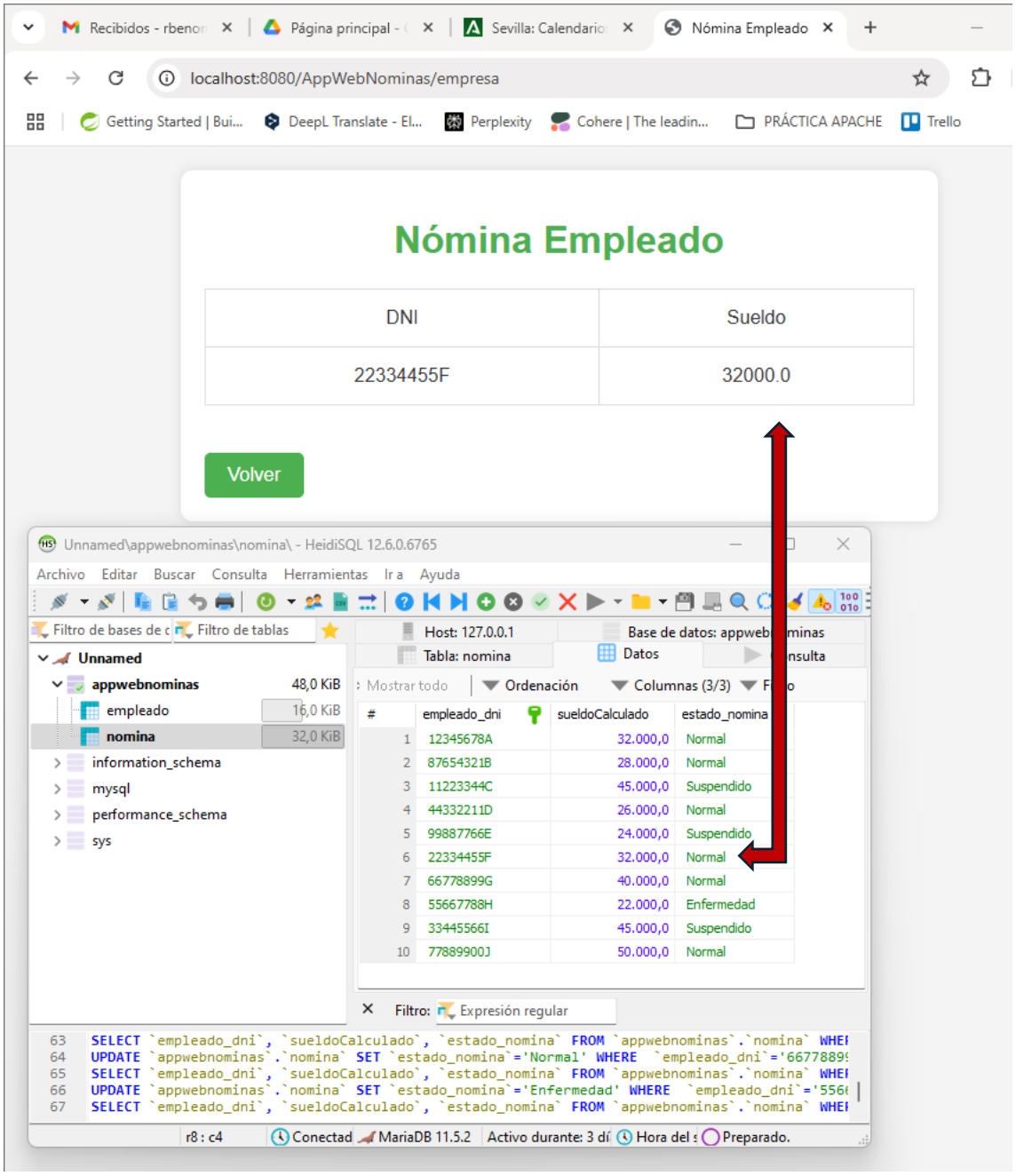
Lo que estamos haciendo básicamente es ver que hay escrito en el campo **ESTADO** de la base de datos al recoger los datos de una nómina con el DNI de su empleado, y asignarle al estado de la nómina pues un estadoNormal, estadoEnfermedad o estadoSuspendido.

En ningún momento cambiamos la información de la base de datos ya que el campo sueldo es un campo calculado que no se puede modificar, lo que hacemos únicamente es cambiar la info que se muestra por pantalla.

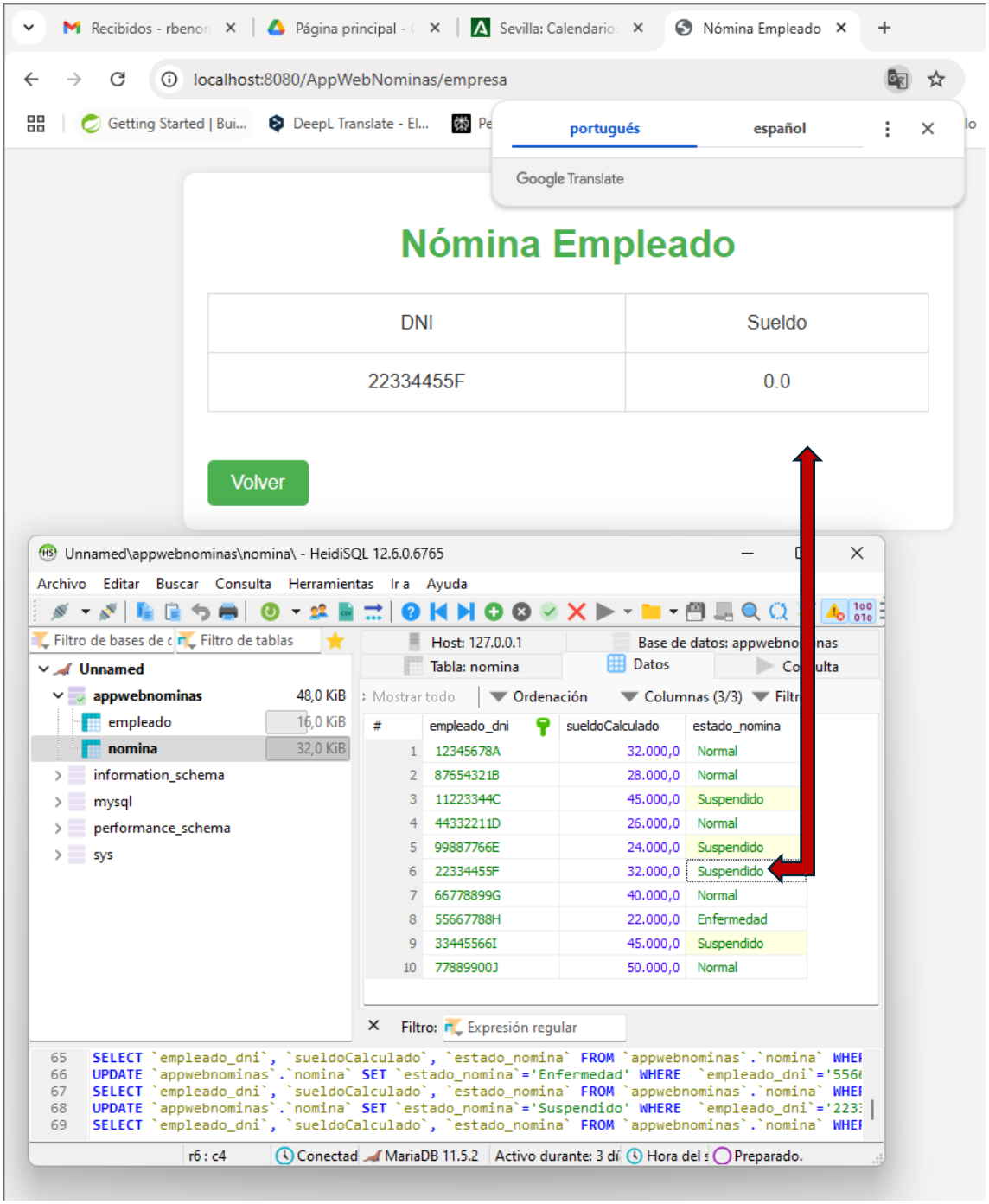
Entonces, si nos vamos al menú de la aplicación donde mostrábamos el sueldo de un empleado por el DNI, ahora podemos cambiar su estado y que el sueldo que muestre por pantalla sea en base a su estado.

Ejemplos:

# EstadoNormal:



# EstadoSuspendido:



# EstadoEnfermedad: