

Patrón Data Access Object (DAO)

Patrón Data Access Object (DAO)	1
¿Qué aprenderás?	2
Introducción	2
Conocer el patrón DAO	3
Estructura del patrón DAO	4
Implementación del patrón DAO en una aplicación web	5
Base de datos	5
Aplicación Web	6
Modelo	6
DAO	8
¿Qué es una Interfaz?	8
El patrón Singleton	12
¿Qué es el patrón Singleton?	14
Patrón dao	15
Ejemplos Patrón DAO	17
Creamos la clase Value Object o DTO	18
Creamos la interface Data Access Object Interface	18
Creamos la clase de la implementación de la interfaz anterior	19
Creamos la clase "StudentDao" para demostrar el uso del patrón DAO	20



¡Comencemos!

¿Qué aprenderás?

- Conocer el patrón DAO.
- Entender la estructura del patrón DAO.
- Entender y aplicar el patrón Singleton para la conexión.
- Generar aplicación básica DAO.

Introducción

En las aplicaciones web modernas (y no tan modernas), el acceso a datos es una parte primordial que tiene que tener un tratamiento especial en cuanto estructura y formato. No es raro que en una aplicación ya puesta en producción, en algún momento de su vida tenga que cambiar de estructura de datos, de implementación o derechamente cambiar el motor de base de datos de una empresa a otra.

La aplicación web debe estar preparada para afrontar estos cambios de la manera menos dolorosa y en un periodo de tiempo acotado, ya que la continuidad operacional de un negocio depende directamente de sus datos y como la palabra lo dice, debe estar continuamente operacional. Para graficar este hecho pensemos en un sistema web que lleva años trabajando con la base de datos Sql Server de microsoft, en la cual toda la lógica de conexión y la lógica de implementación (las consultas, las inserciones, las actualizaciones de datos, etc.) están separadas en clases individuales que representan todas las acciones a realizar sobre la base de datos.

Todo funciona bien, pero por políticas de la empresa se migraron todos los datos a una base de datos Oracle 18c con todo lo que ello implica, nuevo driver de conexión, nuevos formatos de datos, modificación de sentencias. Por cómo está armada la aplicación, será necesario modificar el código ya funcionando que comparte la responsabilidad de conectar a la base de datos y además cambiar la implementación en cada clase, la cual puede ser una o decenas dependiendo de la complejidad del sistema. Esto conlleva un tiempo elevado de implementación dada la complejidad y el gran acoplamiento entre la conexión a la fuente de datos y el manejo de los mismos (cada clase maneja la conexión y las sentencias) gracias a que no hay separación de capas.

Para subsanar estas deficiencias de diseño es que se define un patrón DAO, el cual proporciona un puente entre la lógica de negocio y la fuente de datos.

Conocer el patrón DAO

El patrón DAO (*Data Access Object*) es un patrón de diseño catalogado como patrón estructural que se encarga de abstraer y encapsular el acceso a las bases de datos.

El DAO administra la conexión y la forma en que se obtienen y manipulan los elementos. En palabras más sencillas pensemos en el DAO como un puente entre la lógica de negocios y la fuente de datos (el motor de base de datos) el cual puede ser Oracle, MySQL, PostgreSQL, MongoDB, etc. Si se necesita cambiar de base de datos, solamente atacamos el DAO y no tocamos la lógica de la aplicación, ya que están separados entre capas.

El DAO implementa el mecanismo requerido de acceso para trabajar con la fuente de datos. La fuente de datos puede ser una implementación de MySQL, Oracle, o cualquier base de datos que se pueda imaginar, como también puede ser un sistema de archivos, incluso un Excel.

Los componentes de negocio esperan del DAO el acceso a los datos y les da igual en donde se guardan, solamente necesitan los datos a tal nivel que para el negocio le debe resultar transparente si se cambia el motor de base de datos. Esa es la gran función de este patrón.

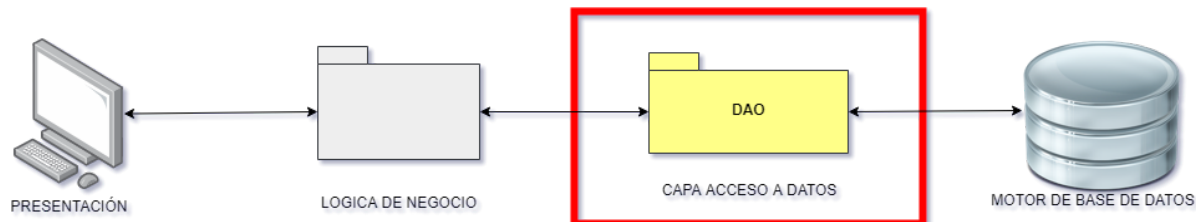


Imagen 1. Capa DAO.
Fuente: Desafío Latam.

Marcado en rojo se muestra el área del DAO, el cual es el encargado de encapsular toda la gestión a la base de datos. Es importante entender que la capa de negocio permanece intacta, sólo se tocará la capa DAO en caso de:

- Modificaciones a las queries.
- Cambio de motor de BD.
- Cambio de driver de conexión.
- Cambios de implementación JDBC.
- Otros.

Estructura del patrón DAO

Para entender la estructura del patrón analizaremos su diagrama de clases simplificado.

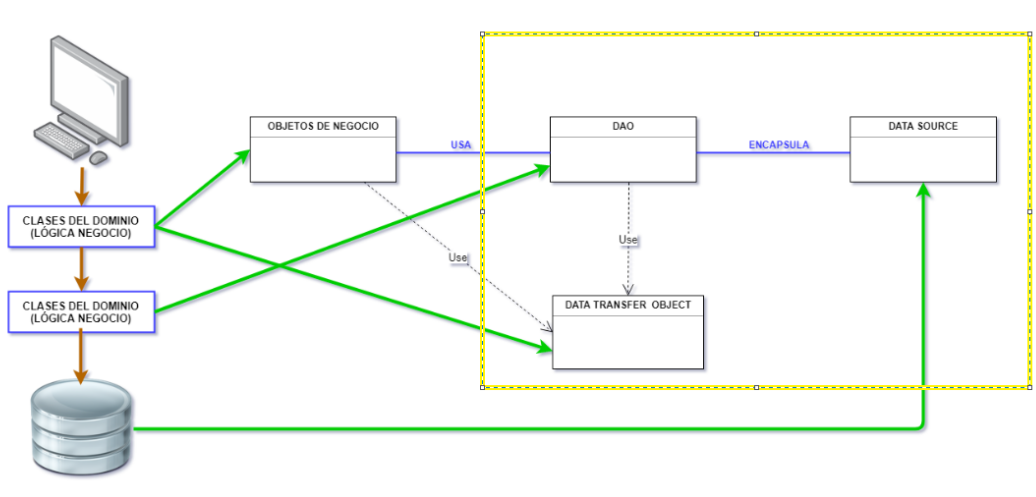


Imagen 2. Estructura del patrón DAO.

Fuente: Desafío Latam.

- **Objetos de negocio:** Representa una entidad del propio negocio (una clase que represente un empleado por ejemplo). Este objeto es quien necesita acceder a la base de datos para obtener los datos de un empleado. Esta entidad puede ser un javabean o un pojo.
- **DAO:** El data access object es el objeto principal del patrón. El DAO abstrae a los objetos de negocio la implementación de acceso a los datos (el dao se encarga de todo y devuelve los datos).
- **Data Source:** Representa la implementación de la conexión al motor de base de datos. Esta clase conoce que base de datos se está usando, además de su driver y el manejo de conexiones.
- **Data Transfer Object:** Conocido como los DTO. Estas entidades son usadas por el DAO para enviar los datos a los objetos de negocio. Piensen en los DTO como la carretilla, el DAO como el maestro constructor, los datos como el cemento y los objetos de negocio como el muro que se quiere construir.



Imagen 3. Analogía entre carretilla y DTO.
Fuente: Desafío Latam.

Implementación del patrón DAO en una aplicación web

Ahora se explica cómo implementar el patrón DAO en una aplicación web utilizando:

- JSP para la vista.
- Un servlet para procesar las peticiones.
- Patrón DAO para el acceso a datos.
- JDBC para establecer la conexión.
- Base de datos de ejemplo.

Base de datos

Utilizaremos la base de datos de ejemplo que se implementó en el ejemplo previamente de JDBC de nombre *desafio_ejemplo01*. Tener a mano el DBEAVER o el SQL DEVELOPER ya que consultaremos constantemente los datos.

Recordemos que esta base de datos cuenta solamente con dos tablas: Departamento y Empleados.



Imagen 4. Relación Departamento - Empleado.
Fuente: Desafío Latam.

Aplicación Web

Crear un nuevo proyecto en eclipse de tipo *Dynamic Web Project*. Para crear la estructura del proyecto, crear dentro de la carpeta src los siguientes packages, en donde dispondremos nuestras clases:

- com.desafiolatam.dao
- com.desafiolatam.modelo
- com.desafiolatam.procesaConexion
- com.desafiolatam.servlet

El patrón DAO se implementará obviamente en el *package dao*. El resto es una arquitectura clásica compuesta por el modelo el cual contendrá las entidades de negocio, el procedimiento de conexión con JDBC y un servlet que se encargará de procesar las peticiones.

Modelo

En el package de modelo, se crean las clases que representan las entidades del negocio las cuales son DEPARTAMENTO y EMPLEADOS.

```
1 package com.desafiolatam.modelo;
2
3 public class Empleado {
4     private int numEmpleado;
5     private String nombreEmpleado;
6     private int numDepto;
7
8     public Empleado(int numEmpleado, String nombreEmpleado, int numDepto) {
9         super();
10        this.numEmpleado = numEmpleado;
11        this.nombreEmpleado = nombreEmpleado;
12        this.numDepto = numDepto;
13    }
14
15    //getters y setters
```

Imagen 5. Empleado.
Fuente: Desafío Latam.

```

1 package com.desafiolatam.modelo;
2
3 public class Departamento {
4     private int numDepto;
5     private String nombreDepto;
6     private String ubicacionDepto;
7
8     public Departamento(int numDepto, String nombreDepto, String ubicacionDepto) {
9         super();
10        this.numDepto = numDepto;
11        this.nombreDepto = nombreDepto;
12        this.ubicacionDepto = ubicacionDepto;
13    }
14
15    //getters y setters

```

Imagen 6. Departamento.
Fuente: Desafío Latam.

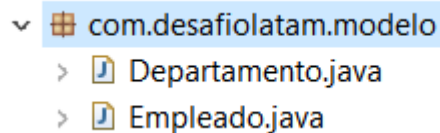


Imagen 7. Paquete modelo.

Los objetos de negocio, deben ser un fiel reflejo de las tablas del modelo de datos en el cual se va a trabajar. El siguiente paso es crear las clases respectivas que representen a las tablas DEPARTAMENTO y EMPLEADO.

Los modelos son las clases principales que se comunicaran con el patrón DAO. Se marca en amarillo el área que cubre dentro del diagrama del patrón.

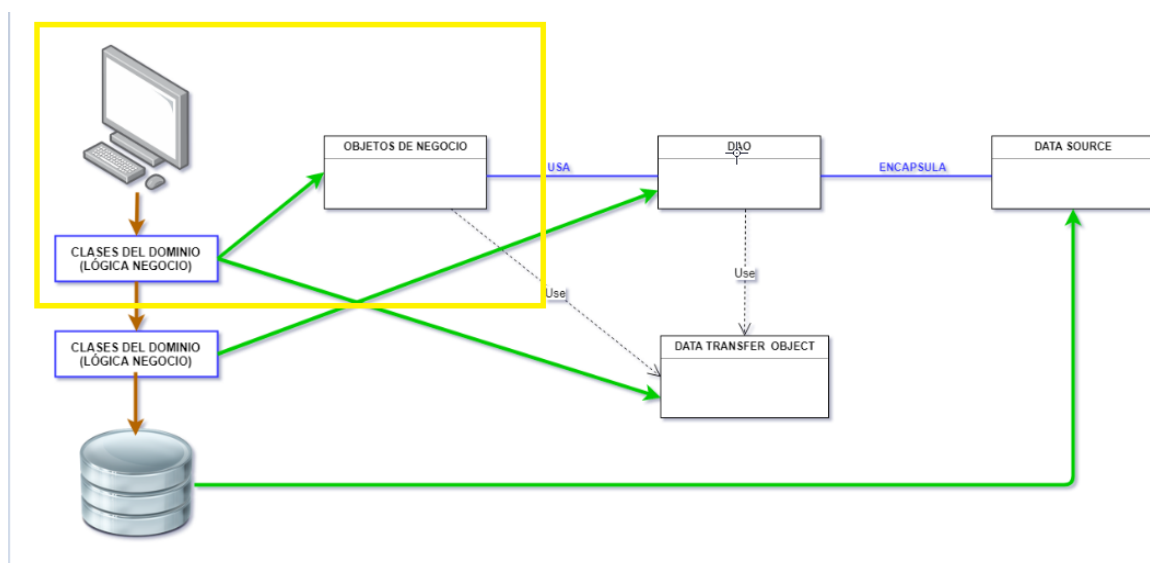


Imagen 8. Diagrama DAO.
Fuente: Desafío Latam.

DAO

¿Qué es una Interfaz?

Una interfaz es similar a una clase abstracta donde puede estar compuesta por uno o más métodos especificados, pero no implementados, por lo tanto estos métodos deben ser implementados por cualquier clase que implemente dicha interfaz.

Las principales características de una interfaz son:

- Herencia.
- Polimorfismo.
- Modularización del código.
- Hacen más simple el mantenimiento del código.
- Facilitan el uso del código.

Ejemplo de una Interfaz.

```
public interface Impresora{
    public abstract print();
    public abstract scan();
    public abstract copy();
}

public class Canon extends Impresora{ //clase donde se haria la
implementación de cada método }

public class Epson extends Impresora{ //clase donde se haria la
implementación de cada método }

public class Samsung extends Impresora{ //clase donde se haria la
implementación de cada método }
```

La capa de acceso a datos es quien abstrae de las complejidades de la conexión a la base de datos o a las implementaciones de manipulación de datos.

La forma de hacerlo es mediante interfaces que como se verá en el ejemplo, tienen el nombre del objeto de negocio a la cual representan, y a la vez con su clase concreta o implementación de la lógica para manipular la información.

Los métodos comunes en una implementación DAO son los siguientes:

- **create()**: Crea un registro en la base de datos.
- **update(id)**: Modifica un registro en la base de datos.
- **delete(id)**: Elimina un registro en la base de datos.
- **getAll()**: Trae todos los registros de una tabla de la base de datos.
- **getById(id)**: Trae un registro específico de una tabla de la base de datos.

Ejemplo de una implementación DAO estándar con una interfaz:

```
public interface Crud{
    public abstract create();
    public abstract update(Integer id);
    public abstract delete(Integer id);
    public abstract getAll();
    public abstract getById(Integer id);
}

public class Usuario extends Crud{
    //Aca debemos implementar cada método de la interfaz con la
    //lógica correspondiente para gestionar usuarios en la base de datos.
    public abstract create(){
    }
    public abstract update(Integer id){
    }
    public abstract delete(Integer id){
    }
    public abstract getAll(){
    }
    public abstract getById(Integer id){
    }
}

public class Auto extends Crud{
    //Aca debemos implementar cada método de la interfaz con la
    //lógica correspondiente para gestionar autos en la base de datos.
    public abstract create(){
    }
    public abstract update(Integer id){
    }
    public abstract delete(Integer id){
    }
    public abstract getAll(){
    }
    public abstract getById(Integer id){
    }
}
```

```
}
```

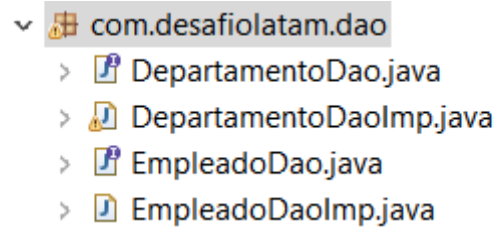


Imagen 9. Objetos DAO.
Fuente: Desafío Latam.

```
1 package com.desafiolatam.dao;
2
3 import java.util.List;
4
5
6
7 public interface DepartamentoDao {
8     public List<Departamento> obtieneDepartamento();
9 }
10
```

Imagen 10. Creando Interfaz Departamento DAO.
Fuente: Desafío Latam.

La interface *DepartamentoDao* se usa para exponer los métodos necesarios para manipular los datos, pero sin implementación. Esto permite que quien quiera utilizar tales métodos solo implemente esta clase y asigne el comportamiento que estime conveniente.

En este caso quien implementa esta interfaz es la clase *DepartamentoDaoImp*.

```
1 package com.desafiolatam.dao;
2
3 import java.sql.Connection;
4
5
6
7
8
9
10
11 public class DepartamentoDaoImp extends AdministradorConexion implements DepartamentoDao{
12
13
14     public DepartamentoDaoImp() {
15         Connection conn = generaConexion();
16     }
17
18     @Override
19     public List<Departamento> obtieneDepartamento() {
20         String sql = "SELECT * FROM departamento";
21         List<Departamento> deptos = new ArrayList<Departamento>();
22         try {
23             pstmt = conn.prepareStatement(sql);
24             rs = pstmt.executeQuery();
25             while(rs.next()) {
26                 Departamento depto = new Departamento(rs.getInt("NUMDEPTO"),rs.getString("NOMDEPTO"),rs.getString("UBICACIONDPTO"));
27                 deptos.add(depto);
28             }
29         } catch (SQLException e) {
30             // TODO Auto-generated catch block
31             e.printStackTrace();
32         }
33         return deptos;
34     }
35 }
36
37
38
```

Imagen 11. Implementación departamento DAO.

Fuente: Desafío Latam.

La clase *DepartamentoDaoImp* implementa el único método que existe actualmente utilizado para obtener todos los departamentos. Una particularidad es el constructor, el cual inicializa la conexión utilizando el método *generaConexion*, el cual es parte de la clase padre *AdministradorConexion*. Cada vez que se utilice a esta clase y se genere una instancia, se creará una conexión a la base de datos de forma automática y las clases del negocio ni se enterarán de ello.

La implementación del método *obtieneDepartamento* incluye la *query* con el *select* a la tabla y una lista de tipo *Departamento* que recibirá los datos de la consulta para luego devolverla como parámetro hacia la clase que lo esté llamando.

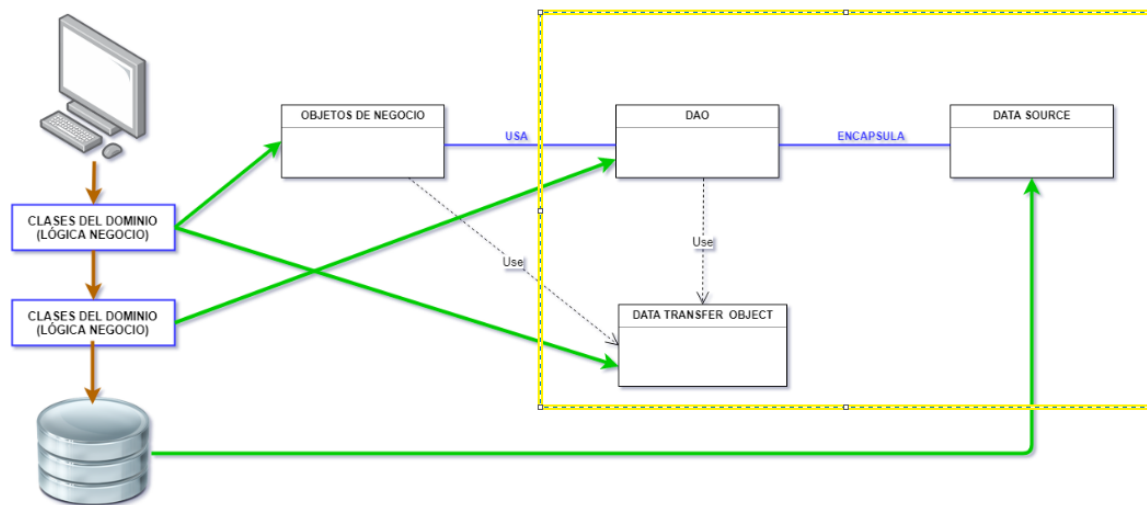


Imagen 12. Diagrama incorporando DATASOURCE.

Fuente: Desafío Latam.

La clase denominada DATASOURCE en el diagrama, se llama *AdministradorConexion* y se encarga de inicializar la conexión mediante JDBC.

```
1 package com.desafiolatam.procesaConexion;
2
3 import java.sql.Connection;
4
5
6
7
8 public class AdministradorConexion {
9
10     protected Connection conn = null;
11     protected PreparedStatement pstmt = null;
12     protected ResultSet rs = null;
13
14     protected Connection generaConexion() {
15         String usr = "sys as sysdba";
16         String pwd = "admin";
17         String driver = "oracle.jdbc.driver.OracleDriver";
18         String url = "jdbc:oracle:thin:@//localhost:1521/desafio_ejemplo01";
19
20         try {
21             Class.forName(driver);
22             conn = DriverManager.getConnection(url,usr,pwd);
23         } catch (Exception ex) {
24             ex.printStackTrace();
25         }
26         return conn;
27     }
28 }
29
30
```

Imagen 13. AdministradorConexion.

Fuente: Desafío Latam.

Muy bien hasta acá, el código del patrón dao está bien encapsulado y una conexión se generará mediante el constructor de la clase *AdministradorConexion*. Pero con este diseño tenemos dos problemas.

El patrón Singleton

¿Qué pasaría si genero muchos objetos de la clase *AdministradorConexion*? Pues generamos muchas conexiones simultáneas a la base de datos y a raíz de eso colgaremos el programa. Lo ideal sería que la conexión se genere una sola vez y nada más, independiente de cuantas instancias de la clase *AdministradorConexion* genere.

Este análisis que hicimos, alguien hace muchos años también lo hizo, y para nuestra fortuna documentó la solución en un patrón de diseño, el cual nosotros solo tenemos que usar. La solución a este problema se llama *Patrón singleton* el cual como su nombre trata de explicar, es un patrón que nos permite generar una y solo una instancia de alguna clase.


Vamos a entender este problema. La clase *AdministradorConexion* genera una conexión utilizando el método *getConnection* y luego la retorna a quien lo solicite:

```
20 protected Connection generaConexion() {  
21     String usr = "sys as sysdba";  
22     String pwd = "admin";  
23     String driver = "oracle.jdbc.driver.OracleDriver";  
24     String url = "jdbc:oracle:thin:@//localhost:1521/desafio_ejemplo01";  
25  
26     try {  
27         Class.forName(driver);  
28         conn = DriverManager.getConnection(url,usr,pwd);  
29     } catch (Exception ex) {  
30         ex.printStackTrace();  
31     }  
32     return conn;  
33 }
```

Imagen 14. Creando una conexión.

Fuente: Desafío Latam.

Esto significa que cada vez que se envíe un requerimiento que tenga que comunicarse con la base de datos, se abrirá una conexión. Miremos este ejemplo en el cual solamente imprimimos por pantalla cuantas veces se abre una conexión.



```
Tomcat v9.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (Sep 28, 2019, 11:47:10 AM)  
Sep 28, 2019 8:59:38 PM org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory getObjectInstance  
WARNING: Name = ConexionOracle Property maxActive is not used in DBCP2, use maxTotal instead. maxTotal default  
Sep 28, 2019 8:59:38 PM org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory getObjectInstance  
WARNING: Name = ConexionOracle Property maxWait is not used in DBCP2, use maxWaitMillis instead. maxWaitMillis  
Sep 28, 2019 8:59:38 PM org.apache.jasper.servlet.TldScanner scanJars  
INFO: At least one JAR was scanned for TLDs yet contained no TLDs. Enable debug logging for this logger for a c  
Sep 28, 2019 8:59:38 PM org.apache.catalina.core.StandardContext reload  
INFO: Reloading Context with name [/ControlaDeptosEmpresa] is completed  
CREACION DE CONEXION CON GetConnection  
CREACION DE CONEXION CON GetConnection  
CREACION DE CONEXION CON GetConnection  
CREACION DE CONEXION CON GetConnection  
CREACION DE CONEXION CON GetConnection  
CREACION DE CONEXION CON GetConnection  
CREACION DE CONEXION CON GetConnection  
CREACION DE CONEXION CON GetConnection
```

Imagen 15. Resultado a la ejecución del código.

Fuente: Desafío Latam.

Por cada acción se abre una conexión, mala idea si pensamos en el rendimiento de la aplicación. Para solucionar esto existe el patrón Singleton.

¿Qué es el patrón Singleton?

Es uno de los tantos patrones de diseños elaborados por la banda de los 4. De todos los existentes es el más fácil de comprender e implementar, ya que lo único que hace es impedir que se generen nuevas instancias de clases si es que existe una. Está catalogado dentro de los patrones creacionales. Para este proyecto tenemos que:

1. Declarar la instancia de *Connection* como *static*. Con esto esta variable de conexión es parte de la clase y será compartida entre métodos.
2. Verificar si el objeto de tipo *Connection* es nulo o no. En caso de ser nulo generamos la conexión con todo lo que conlleva, y si resulta ser válida, retornamos la misma variable, ya que no es necesario crearla nuevamente.

```
35 protected Connection generaPoolConexion() {  
36     Context initContext;  
37     if(conn == null) {  
38         try {  
39             initContext = new InitialContext();  
40             DataSource ds = (DataSource) initContext.lookup("java:/comp/env/jdbc/ConexionOracle");  
41             try {  
42                 conn = ds.getConnection();  
43                 System.out.println("CREACION DE CONEXION CON GetConnection");  
44             } catch (SQLException e) {  
45                 e.printStackTrace();  
46             }  
47         } catch (NamingException e) {  
48             e.printStackTrace();  
49         }  
50         return conn;  
51     } else {  
52         return conn;  
53     }  
54 }  
55 }  
56 }  
57 }  
58 }
```

Imagen 16. Verificando que no sea nulo.

Fuente: Desafío Latam.

Con este cambio, las instancias serán creadas solamente una vez, ya que si existen previamente no dejamos que se ejecute la creación, en su lugar devolvemos dicha instancia estática.

```
Sep 28, 2019 9:50:18 PM org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory getObjectInstance
WARNING: Name = ConexionOracle Property maxActive is not used in DBCP2, use maxTotal instead. maxTotal default
Sep 28, 2019 9:50:18 PM org.apache.tomcat.dbcp.dbcp2.BasicDataSourceFactory getObjectInstance
WARNING: Name = ConexionOracle Property maxWait is not used in DBCP2, use maxWaitMillis instead. maxWaitMill
Sep 28, 2019 9:50:18 PM org.apache.jasper.servlet.TldScanner scanJars
INFO: At least one JAR was scanned for TLDs yet contained no TLDs. Enable debug logging for this logger for a
Sep 28, 2019 9:50:18 PM org.apache.catalina.core.StandardContext reload
INFO: Reloading Context with name [/ControlaDeptosEmpresa] is completed
CREACION DE CONEXION CON GetConnection
```

Imagen 17. Una conexión creada.
Fuente: Desafío Latam.

Patrón dao

El patrón dao es muy simple, y se verá cuando se descubra quien llama a esta jerarquía de clases. Se implementa un servlet que recibe un request con la llamada a la petición de departamentos.

```
1 package com.desafiolatam.servlet;
2
3 import java.io.IOException;
4
5
6 /**
7  * Servlet implementation class ProcesaDepartamento
8  */
9 @WebServlet("/procesaDepartamento")
10 public class ProcesaDepartamento extends HttpServlet {
11     private static final long serialVersionUID = 1L;
12
13
14     /**
15      * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
16      */
17     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
18         // TODO Auto-generated method stub
19         List<Departamento> listaDeptos = new ArrayList<Departamento>();
20         DepartamentoDaoImp deptoDao = new DepartamentoDaoImp();
21         listaDeptos = deptoDao.obtieneDepartamento();
22         request.setAttribute("deptoDao", listaDeptos);
23         request.getRequestDispatcher("resultado.jsp").forward(request, response);
24     }
25 }
26
27
28
29
30
31
32
33
34
35
36
37
38
```

Imagen 18. Patrón DAO.
Fuente: Desafío Latam.

El servlet crea una instancia de *DepartamentoDaoImpl* y utiliza el método expuesto por la interfaz de nombre *obtieneDepartamento*. Para recibir los valores se crea una lista de tipo departamento la cual al obtener dicha información la guarda en una lista de retorno.

La línea 33 redirecciona al jsp que desplegará los datos obtenidos por el DAO en la pantalla.

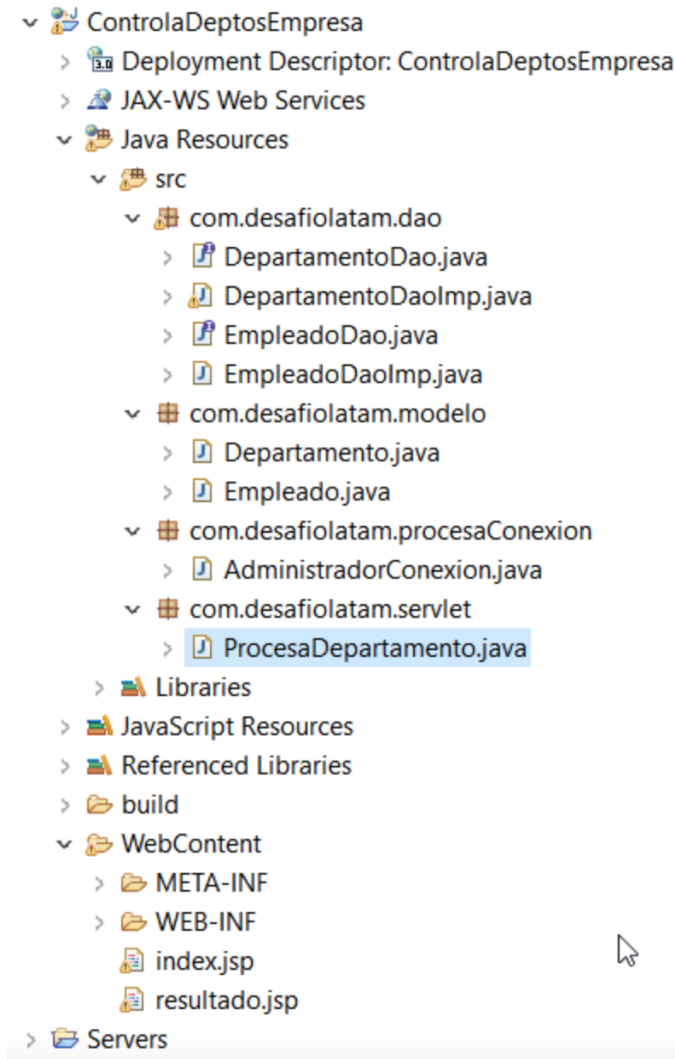


Imagen 19. Estructura del proyecto.
Fuente: Desafío Latam.

Al ejecutar el proyecto se desplegará la información contenida en la tabla de departamentos en una sencilla tabla del jsp resultado.

localhost:8080/ControlaDeptosEmpresa/procesaDepartamento

departamentos		
Nombre Depto	Numero Depto	Ubicación
11	INFORMATICA	@CHILE
12	CONTABILIDAD	@PERU
13	RRHH	@COLOMBIA

Imagen 20. JSP resultante.
Fuente: Desafío Latam.

Ejemplos Patrón DAO

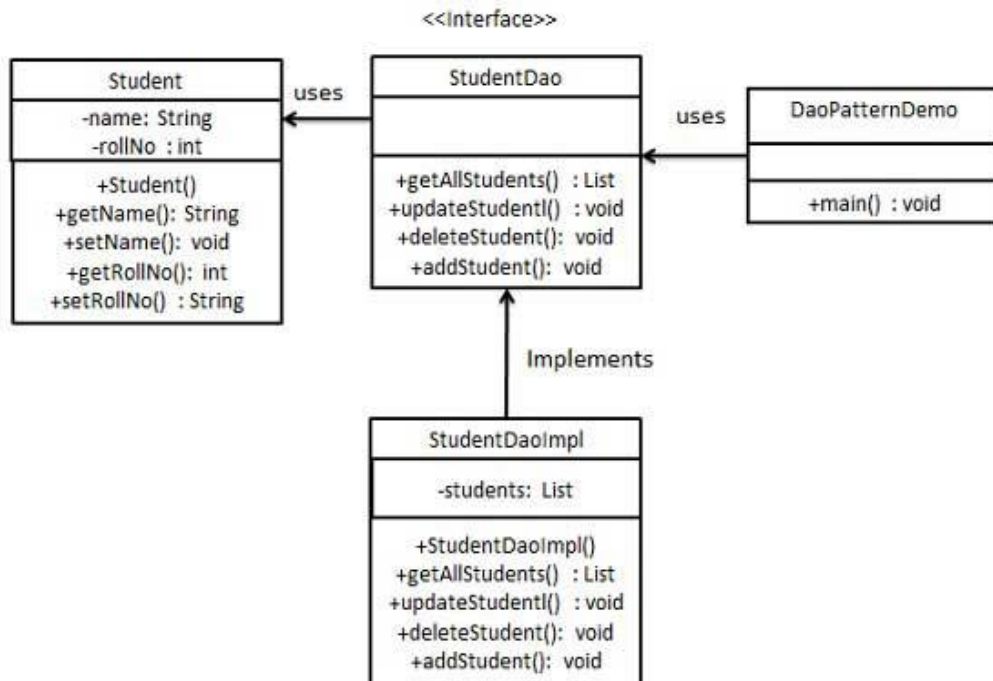


Imagen 21. Ejemplo Patrón DAO 1.

Fuente: tutorialspoint.com.

Creamos la clase Value Object o DTO

Student.java

```
public class Student {  
    private String name;  
    private int rollNo;  
    Student(String name, int rollNo){  
        this.name = name;  
        this.rollNo = rollNo;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getRollNo() {  
        return rollNo;  
    }  
    public void setRollNo(int rollNo) {  
        this.rollNo = rollNo;  
    }  
}
```

Creamos la interface Data Access Object Interface

StudentDao.java

```
import java.util.List;  
public interface StudentDao {  
    public List<Student> getAllStudents();  
    public Student getStudent(int rollNo);  
    public void updateStudent(Student student);  
    public void deleteStudent(Student student);  
}
```

Creamos la clase de la implementación de la interfaz anterior

StudentDaoImpl.java

```
import java.util.ArrayList;
import java.util.List;
public class StudentDaoImpl implements StudentDao {
    //list is working as a database
    List<Student> students;
    public StudentDaoImpl(){
        students = new ArrayList<Student>();
        Student student1 = new Student("Juan",0);
        Student student2 = new Student("Marcos",1);
        students.add(student1);
        students.add(student2);
    }
    @Override
    public void deleteStudent(Student student) {
        students.remove(student.getRollNo());
        System.out.println("Student: Roll No " + student.getRollNo() +
            ", deleted from database");
    }
    //retrive list of students from the database
    @Override
    public List<Student> getAllStudents() {
        return students;
    }
    @Override
    public Student getStudent(int rollNo) {
        return students.get(rollNo);
    }
    @Override
    public void updateStudent(Student student) {
        students.get(student.getRollNo()).setName(student.getName());
        System.out.println("Student: Roll No " + student.getRollNo() +
            ", updated in the database");
    }
}
```

Creamos la clase "StudentDao" para demostrar el uso del patrón DAO

DaoPatternDemo.java

```
public class DaoPatternDemo {  
    public static void main(String[] args) {  
        StudentDao studentDao = new StudentDaoImpl();  
        //print all students  
        for (Student student : studentDao.getAllStudents()) {  
            System.out.println("Student: [RollNo : " +  
student.getRollNo() + ", Name : " + student.getName() + " ]");  
        }  
        //update student  
        Student student = studentDao.getAllStudents().get(0);  
        student.setName("David");  
        studentDao.updateStudent(student);  
        //get the student  
        studentDao.getStudent(0);  
        System.out.println("Student: [RollNo : " + student.getRollNo()  
+ ", Name : " + student.getName() + " ]");  
    }  
}
```

Luego verificamos la salida de nuestro demo:

```
Student: [RollNo : 0, Name : Juan ]  
  
Student: [RollNo : 1, Name : Marcos ]  
  
Student: Roll No 0, updated in the database  
  
Student: [RollNo : 0, Name : David ]
```

A continuación otros ejemplos del Patrón DAO representados como diagramas de clases y diagrama de componentes.

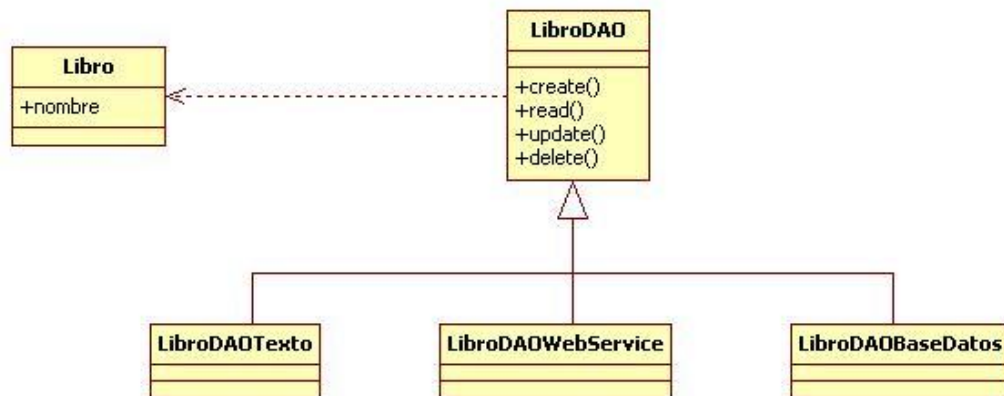


Imagen 22. Ejemplo Patrón DAO 2.
Fuente: orgullo.users.sourceforge.net.

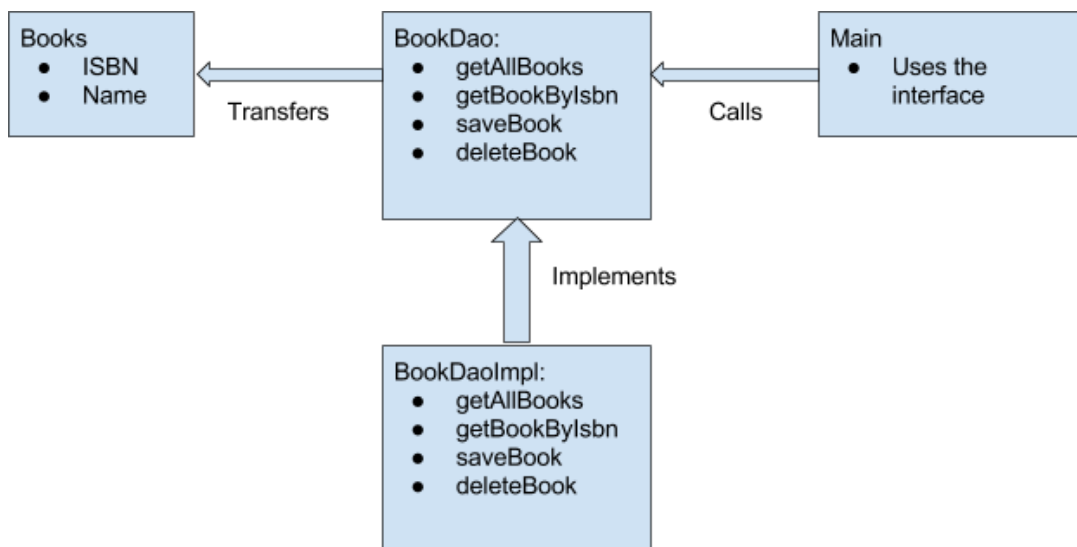


Imagen 23. Ejemplo Patrón DAO 3.
Fuente: es.w3ki.com.