UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Universidad Distrital Francisco José de Caldas

Facultad de Ingeniería

Ingeniería de Sistemas

Principios y Patrones de Diseño

Proyecto: *Odally* | *Blog*

Estudiantes: Dylan David Silva Orrego – 20242020130
Maria Alejandra Munevar Barrera – 20242020145
Sergio Leonardo Moreno Granado – 20242020091

Profesora: Lilia Marcela Espinosa Rodríguez

Programación Avanzada

Semestre 2025-3

Diciembre de 2025, Bogotá D.C.

Índice

1. Introducción

Este documento presenta una exploración exhaustiva de los principios de ingeniería de software y patrones de diseño aplicados en el proyecto **Odally | Blog**, un sistema de gestión de contenidos desarrollado con tecnologías JavaWeb (Servlets/JSP).

El proyecto no solo implementa los conocidos principios SOLID, sino que también aplica un espectro más amplio de buenas prácticas de ingeniería de software, incluyendo principios de arquitectura de paquetes, principios de diseño general y patrones de diseño clásicos de la industria.

1.1. Objetivos del Documento

- Documentar la aplicación práctica de principios SOLID en código real
- Explicar los principios de arquitectura de paquetes en el contexto del proyecto
- Demostrar la implementación de principios de diseño como DRY, KISS, YAGNI, SoC y LoD
- Describir los patrones de diseño implementados y su justificación
- Proporcionar referencias cruzadas entre teoría y código fuente

1.2. Arquitectura del Sistema

El sistema Odally | Blog sigue una arquitectura MVC (Model-View-Controller) implementada con:

- **Modelo:** POJOs (Plain Old Java Objects) para entidades del dominio
- **Vista:** JSPs con JSTL para presentación
- **Controlador:** Servlets de Jakarta EE
- **Persistencia:** Patrón DAO con MySQL

2. Principios SOLID

SOLID es un acrónimo que representa cinco principios fundamentales de diseño orientado a objetos, introducidos por Robert C. Martin [1]. Estos principios buscan hacer el software más comprensible, flexible y mantenible.

2.1. S - Single Responsibility Principle (Principio de Responsabilidad Única)

2.1.1. Definición

"Una clase debe tener una, y solo una, razón para cambiar" [2].

Este principio establece que cada clase debe tener una única responsabilidad bien definida. Si una clase tiene múltiples responsabilidades, los cambios en una pueden afectar a las otras, creando acoplamiento y fragilidad.

2.1.2. Aplicación en el Proyecto

Clase `ConexionBD` La clase `ConexionBD` tiene **una única responsabilidad**: gestionar el pool de conexiones a la base de datos.

```
/**
 * Clase Singleton para gestionar el pool de conexiones
 * RESPONSABILIDAD: Únicamente proporcionar conexiones JDBC.
 * No valida usuarios, no formatea fechas, solo conecta.
 */
public class ConexionBD {
    private static volatile ConexionBD instancia;
    private final List<PooledConnection> connectionPool;

    public Connection getConexion() throws SQLException {
        // Lógica de pool y reintentos
    }

    public void cerrarConexion(Connection conn) {
        // Devolver conexión al pool
    }
}
```

Listing 1: ConexionBD.java - Responsabilidad Única

¿Por qué cumple SRP?

- Solo maneja conexiones a base de datos
- No contiene lógica de negocio
- No valida datos de usuarios
- No ejecuta consultas SQL
- Cambiaría solo si cambia la estrategia de conexión

Clase `PasswordUtil` Esta clase utilitaria tiene como **única responsabilidad**: hashear y verificar contraseñas.

```
public class PasswordUtil {
    /**
```

```
     * RESPONSABILIDAD ÚNICA: Hashear contraseñas con SHA-256
     */
    public static String hashPassword(String password) {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        // ... lógica de hasheo
        return hexString.toString();
    }

    public static boolean verificarPassword(String password,
                                            String hashedPassword) {
        return hashPassword(password).equals(hashedPassword);
    }
}
```

Listing 2: PasswordUtil.java - Una Sola Responsabilidad

Clases Modelo (POJOs) Las clases `Usuario` y `Articulo` son contenedores de datos puros:

```
/**
 * POJO que solo contiene datos de un artículo.
 * No sabe cómo guardarse, no valida, solo almacena.
 */
public class Articulo implements Serializable {
    private int id;
    private String titulo;
    private String contenido;
    private LocalDateTime fechaPublicacion;

    // Getters y Setters únicamente
}
```

Listing 3: Articulo.java - POJO Simple

2.2. O - Open/Closed Principle (Principio Abierto/Cerrado)

2.2.1. Definición

"Las entidades de software deben estar abiertas para extensión, pero cerradas para modificación" [1].

Este principio sugiere que debemos poder agregar nueva funcionalidad sin modificar el código existente, típicamente mediante el uso de interfaces y polimorfismo.

2.2.2. Aplicación en el Proyecto

Interfaces DAO El sistema define interfaces `IArticuloDAO` e `IUsuarioDAO` que están cerradas a modificación pero abiertas a extensión:

```
/**
 * Interfaz cerrada a modificación.
 * Podemos extenderla con nuevas implementaciones
 * sin tocar el código existente.
 */
public interface IArticuloDAO {
    List<Articulo> listarTodos() throws SQLException;
    Articulo obtenerPorId(int id) throws SQLException;
    boolean crear(Articulo articulo) throws SQLException;
```

```
    // ... más métodos
}
```

Listing 4: IArticuloDAO.java - Abierto/Cerrado

Extensión sin modificación:

```
// Implementación actual para MySQL
public class MySQLArticuloDAO implements IArticuloDAO {
    // Implementación con MySQL
}

// Podríamos agregar PostgreSQL sin modificar código existente
public class PostgreSQLArticuloDAO implements IArticuloDAO {
    // Nueva implementación con PostgreSQL
}

// O incluso MongoDB
public class MongoArticuloDAO implements IArticuloDAO {
    // Implementación NoSQL
}
```

Listing 5: Múltiples implementaciones

Uso en Controladores Los servlets dependen de la interfaz, no de la implementación concreta:

```
public class ArticuloServlet extends HttpServlet {
    // Depende de la interfaz, no de MySQL específicamente
    private IArticuloDAO articuloDAO;

    @Override
    public void init() {
        // Podemos inyectar cualquier implementación
        this.articuloDAO = new MySQLArticuloDAO();
        // Mañana: this.articuloDAO = new PostgreSQLArticuloDAO();
    }
}
```

Listing 6: ArticuloServlet.java

Beneficio: Si necesitamos cambiar de MySQL a PostgreSQL o Oracle, solo modificamos la línea de inyección. El resto del servlet permanece intacto.

2.3. L - Liskov Substitution Principle (Principio de Sustitución de Liskov)

2.3.1. Definición

“Los objetos de una clase derivada deben poder reemplazar objetos de la clase base sin alterar el correcto funcionamiento del programa” [6].

En términos simples: si una clase B extiende o implementa una clase/interfaz A, entonces B debe poder usarse en cualquier lugar donde se espera A sin romper la funcionalidad.

2.3.2. Aplicación en el Proyecto

Implementaciones DAO Intercambiables Todas las implementaciones de `IArticuloDAO` son intercambiables:

```java
public class MySQLArticuloDAO implements IArticuloDAO {
    /**
     * Cumple estrictamente el contrato de IArticuloDAO.
     * - listarTodos() devuelve List<Articulo>, nunca null
     * - obtenerPorId() devuelve Articulo o null
     * - crear() devuelve boolean indicando éxito
     * - No lanza excepciones no declaradas
     */
    @Override
    public List<Articulo> listarTodos() throws SQLException {
        // Siempre devuelve List, nunca null
        return new ArrayList<>();
    }
}
```

Listing 7: LSP en acción

Garantías del contrato:

- Métodos de lectura no modifican estado
- Excepciones declaradas en la interfaz
- Tipos de retorno consistentes
- Precondiciones no más estrictas
- Postcondiciones no más débiles

2.4. I - Interface Segregation Principle (Principio de Segregación de Interfaces)

2.4.1. Definición

"Los clientes no deberían verse forzados a depender de interfaces que no usan" [1].

Es mejor tener varias interfaces específicas que una interfaz general muy grande ("God Interface").

2.4.2. Aplicación en el Proyecto

Interfaces Específicas por Entidad En lugar de tener un `IDAO` genérico gigante, tenemos interfaces específicas:

```java
// Interfaz específica para artículos
public interface IArticuloDAO {
    List<Articulo> listarTodos() throws SQLException;
    Articulo obtenerPorId(int id) throws SQLException;
    boolean crear(Articulo articulo) throws SQLException;
    boolean actualizar(Articulo articulo) throws SQLException;
    boolean eliminar(int id) throws SQLException;
    int contarTotal() throws SQLException;
}

// Interfaz específica para usuarios (separada)
public interface IUsuarioDAO {
    Usuario buscarPorUsername(String username) throws SQLException;
    Usuario obtenerPorId(int id) throws SQLException;
    boolean crear(Usuario usuario) throws SQLException;
    List<Usuario> listarTodos() throws SQLException;
```

```
    boolean actualizarRol(int id, String nuevoRol) throws SQLException;
    // ... métodos específicos de usuarios
}
```

Listing 8: Interfaces segregadas

Ventajas:

- `MySQLArticuloDAO` solo implementa métodos de artículos
- `MySQLUsuarioDAO` solo implementa métodos de usuarios
- No hay métodos "dummy" o sin implementar
- Interfaces cohesivas y fáciles de entender

2.5. D - Dependency Inversion Principle (Principio de Inversión de Dependencias)

2.5.1. Definición

"Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones" [1].

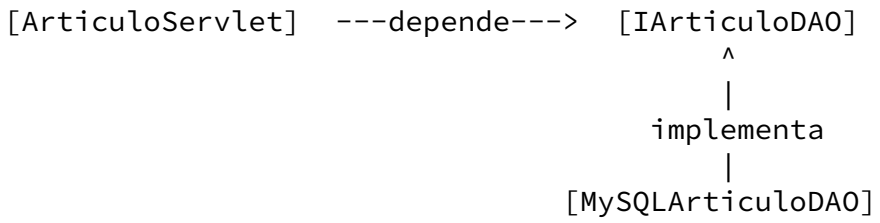
2.5.2. Aplicación en el Proyecto

Arquitectura por Capas El proyecto invierte las dependencias entre capas:

```
// MÓDULO DE ALTO NIVEL (Controlador)
public class ArticuloServlet extends HttpServlet {
    // Depende de la abstracción (interfaz), no de MySQL
    private IArticuloDAO articuloDAO; // <<< ABSTRACCIÓN

    @Override
    public void init() {
        // La implementación concreta se inyecta
        this.articuloDAO = new MySQLArticuloDAO();
    }

    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {
        // Trabaja con la abstracción
        List<Articulo> articulos = articuloDAO.listarTodos();
    }
}

// MÓDULO DE BAJO NIVEL (Implementación)
public class MySQLArticuloDAO implements IArticuloDAO {
    // Implementa la abstracción definida por el módulo de alto nivel
}
```

Listing 9: Inversión de Dependencias

Flujo de dependencias:



Beneficios:

- El controlador no conoce MySQL
- Podemos testear con mocks
- Podemos cambiar de BD sin tocar controladores
- Reducción del acoplamiento

3. Principios de Arquitectura de Paquetes

Robert C. Martin [3] definió seis principios para organizar paquetes (packages) en sistemas orientados a objetos. Se dividen en dos grupos: principios de cohesión y principios de acoplamiento.

3.1. Principios de Cohesión de Paquetes

Estos principios ayudan a decidir qué clases deben ir juntas en un paquete.

3.1.1. REP - Reuse/Release Equivalence Principle

Definición: "El granularado de reutilización es el granularado de liberación" [3].

Aplicación: El paquete `com.blog.dao` es una unidad cohesiva de reutilización:

```
com.blog.dao/
    +── ConexionBD.java          # Singleton de conexión
    +── IArticuloDAO.java        # Interfaz
    +── MySQLArticuloDAO.java    # Implementación
    +── IUsuarioDAO.java
    +── MySQLUsuarioDAO.java
```

Todo el paquete se puede reutilizar como unidad en otro proyecto que necesite acceso a datos de un blog.

3.1.2. CCP - Common Closure Principle

Definición: "Las clases que cambian juntas, deben empaquetarse juntas" [3].

Aplicación:

- Paquete `com.blog.model`: Las clases `Usuario` y `Articulo` cambian cuando cambia el modelo de dominio
- Paquete `com.blog.filter`: Los filtros `AuthFilter`, `CharacterEncodingFilter` y `DatabaseCheck` cambian cuando cambian los requisitos de seguridad o conectividad

3.1.3. CRP - Common Reuse Principle

Definición: "Las clases que se usan juntas, deben empaquetarse juntas".

Aplicación: Si usas `IArticuloDAO`, probablemente uses `ConexionBD`. Ambas están en `com.blog.dao`.

3.2. Principios de Acoplamiento de Paquetes

Estos principios guían las dependencias entre paquetes.

3.2.1. ADP - Acyclic Dependencies Principle

Definición: "El grafo de dependencias de paquetes no debe contener ciclos" [3].

Aplicación en el Proyecto:

```
com.blog.controller  ------>  com.blog.dao
        |                            |
        |                            v
        v                     com.blog.model
   com.blog.filter
```

No hay ciclos: controller usa dao, dao usa model, pero model no depende de nada.

3.2.2. SDP - Stable Dependencies Principle

Definición: "Depende en la dirección de estabilidad".

Aplicación:

- `com.blog.model` es el paquete más estable (casi nunca cambia)
- `com.blog.dao` es estable (cambia poco)
- `com.blog.controller` es menos estable (cambia con requisitos de UI)

Las dependencias fluyen de menos estable a más estable.

3.2.3. SAP - Stable Abstractions Principle

Definición: "Los paquetes estables deben ser abstractos".

Aplicación: El paquete `com.blog.dao` es estable y contiene abstracciones (interfaces `IArticuloDAO`, `IUsuarioDAO`).

4. Otros Principios de Diseño

4.1. DRY - Don't Repeat Yourself (No te Repitas)

4.1.1. Definición

"Cada pieza de conocimiento debe tener una representación única, inequívoca y autorizada dentro del sistema" [7].

4.1.2. Aplicación en el Proyecto

Singleton de Conexión La lógica de carga de db.properties y creación de conexiones está centralizada en ConexionBD:

```
// MALO (DRY violado): Cada DAO carga db.properties
public class MySQLArticuloDAO {
    private void conectar() {
        Properties props = new Properties();
        props.load(new FileInputStream("db.properties"));
        // ... crear conexión
    }
}

// BUENO (DRY respetado): Un solo punto de carga
public class ConexionBD {
    private void loadProperties() {
        // Se carga UNA sola vez
    }
}
```

Listing 10: Evitando duplicación con Singleton

Uso de JSTL en Vistas En lugar de repetir scriptlets Java en cada JSP:

```
<!-- MALO: Repetir lógica en cada JSP -->
<% for(Articulo a : articulos) { %>
    <h3><%= a.getTitulo() %></h3>
<% } %>

<!-- BUENO: JSTL (No Repetición) -->
<c:forEach var="articulo" items="${articulos}">
    <h3>${articulo.titulo}</h3>
</c:forEach>
```

Listing 11: Sin DRY vs Con DRY

4.2. KISS - Keep It Simple, Stupid (Mantenlo Simple)

4.2.1. Definición

"La mayoría de los sistemas funcionan mejor si se mantienen simples en lugar de complicados" [10].

4.2.2. Aplicación en el Proyecto

POJOs Simples Las clases modelo son extremadamente simples:

```java
public class Articulo implements Serializable {
    // Solo campos, getters y setters
    private int id;
    private String titulo;
    private String contenido;

    // Nada de lógica compleja
    // Nada de validaciones complicadas
    // Nada de dependencias externas
}
```

Listing 12: Simplicidad en POJOs

PasswordUtil Implementación simple usando SHA-256 de la biblioteca estándar:

```java
public static String hashPassword(String password) {
    MessageDigest digest = MessageDigest.getInstance("SHA-256");
    byte[] hash = digest.digest(password.getBytes(UTF_8));
    // Convertir a hex (simple y directo)
    return toHexString(hash);
}
```

Listing 13: Implementación simple de hasheo

No se usa ninguna biblioteca de terceros innecesaria.

4.3. YAGNI - You Aren't Gonna Need It (No lo vas a Necesitar)

4.3.1. Definición

"No agregues funcionalidad hasta que sea realmente necesaria" [8].

4.3.2. Aplicación en el Proyecto

Pool de Conexiones Manual En lugar de implementar un pool complejo con todas las campanas y silbatos, se implementó solo lo necesario:

```java
public class ConexionBD {
    // Implementamos:
    // - Pool básico con min/max conexiones
    // - Validación simple
    // - Reintentos con backoff

    // NO implementamos (YAGNI):
    // - Monitoreo JMX
    // - Métricas avanzadas
    // - Configuración dinámica en caliente
    // - Múltiples pools para diferentes BDs
}
```

Listing 14: Solo lo necesario

4.4. SoC - Separation of Concerns (Separación de Responsabilidades)

4.4.1. Definición

"Un programa debe separarse en secciones distintas, de modo que cada sección aborde una preocupación separada" [9].

4.4.2. Aplicación en el Proyecto

Arquitectura MVC La separación de responsabilidades es evidente:

```
// VISTA (JSP): Solo presentación
<h1>Lista de Artículos</h1>
<c:forEach var="articulo" items="${articulos}">
    <h2>${articulo.titulo}</h2>
</c:forEach>

// CONTROLADOR (Servlet): Solo orquestación
public class ArticuloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {
        List<Articulo> articulos = articuloDAO.listarTodos();
        request.setAttribute("articulos", articulos);
        request.getRequestDispatcher("/index.jsp").forward(...);
    }
}

// MODELO (DAO): Solo acceso a datos
public class MySQLArticuloDAO implements IArticuloDAO {
    public List<Articulo> listarTodos() {
        // SQL puro, sin lógica de presentación
    }
}
```

Listing 15: Separación en MVC

4.5. LoD - Law of Demeter (Ley de Deméter)

4.5.1. Definición

"No hables con extraños" – Un método de un objeto solo debe llamar métodos de:

- El objeto mismo
- Objetos pasados como parámetros
- Objetos creados dentro del método
- Objetos que son atributos del objeto

4.5.2. Aplicación en el Proyecto

```
// MALO (violación LoD - cadena de llamadas)
String autorEmail = articulo.getAutor().getEmail();

// BUENO (respeta LoD)
public class Articulo {
    private Usuario autor;
```

```
7
8       // Método de conveniencia que oculta la estructura interna
9       public String getAutorEmail() {
10          return autor != null ? autor.getEmail() : null;
11      }
12  }
13
14  // En el código cliente
15  String autorEmail = articulo.getAutorEmail();
```

Listing 16: Respetando la Ley de Deméter

5. Patrones de Diseño Implementados

5.1. Singleton Pattern

5.1.1. Definición

El patrón Singleton garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella [4].

5.1.2. Implementación en ConexionBD

```
public class ConexionBD {
    // volatile garantiza visibilidad entre threads
    private static volatile ConexionBD instancia;

    // Constructor privado - nadie puede hacer new ConexionBD()
    private ConexionBD() {
        // Inicialización del pool
    }

    // Método estático de acceso
    public static ConexionBD getInstancia() {
        if (instancia == null) {  // Primera verificación (sin lock)
            synchronized (ConexionBD.class) {  // Lock solo si necesario
                if (instancia == null) {  // Segunda verificación (con
                    ↪ lock)
                    instancia = new ConexionBD();
                }
            }
        }
        return instancia;
    }
}
```

Listing 17: Singleton Thread-Safe con Double-Checked Locking

¿Por qué Singleton?

- Un pool de conexiones debe ser único en la aplicación
- Evita múltiples pools compitiendo por conexiones
- Centraliza la configuración de base de datos
- Garantiza uso eficiente de recursos

5.2. DAO Pattern (Data Access Object)

5.2.1. Definición

El patrón DAO abstrae y encapsula todo acceso a la fuente de datos. El DAO gestiona la conexión con el origen de datos para obtener y almacenar datos [11].

5.2.2. Estructura en el Proyecto

```
// 1. INTERFAZ DAO (Contrato)
public interface IArticuloDAO {
    List<Articulo> listarTodos() throws SQLException;
```

```
    Articulo obtenerPorId(int id) throws SQLException;
    boolean crear(Articulo articulo) throws SQLException;
}

// 2. IMPLEMENTACIÓN DAO (Encapsula SQL)
public class MySQLArticuloDAO implements IArticuloDAO {
    @Override
    public List<Articulo> listarTodos() throws SQLException {
        String sql = "SELECT * FROM articulos";
        // ... lógica JDBC encapsulada
    }
}

// 3. USO EN CONTROLADOR (Desacoplado de SQL)
public class ArticuloServlet extends HttpServlet {
    private IArticuloDAO dao = new MySQLArticuloDAO();

    protected void doGet(...) {
        List<Articulo> articulos = dao.listarTodos();
        // No hay SQL aquí, solo lógica de negocio
    }
}
```

Listing 18: Patrón DAO completo

Ventajas del patrón DAO:

- Separa lógica de negocio de lógica de persistencia
- Facilita cambiar de tecnología de BD
- Permite testear con implementaciones mock
- Centraliza manejo de excepciones SQL

5.3. MVC Pattern (Model-View-Controller)

5.3.1. Definición

MVC separa la aplicación en tres componentes interconectados: Modelo (datos), Vista (presentación) y Controlador (lógica) [12].

5.3.2. Implementación en el Proyecto

Modelo (Model)

```
package com.blog.model;

public class Articulo {
    private int id;
    private String titulo;
    private String contenido;
    // Getters y setters
}
```

Listing 19: Modelo - Entidades de dominio

Vista (View)

```jsp
<!-- index.jsp -->
<c:forEach var="articulo" items="${articulos}">
    <article>
        <h2>${articulo.titulo}</h2>
        <p>${articulo.contenido}</p>
    </article>
</c:forEach>
```

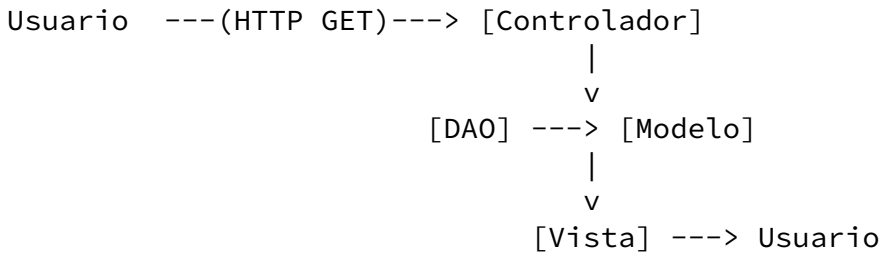
Listing 20: Vista - JSP con JSTL

Controlador (Controller)

```java
public class ArticuloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {
        // 1. Obtener datos del modelo
        List<Articulo> articulos = articuloDAO.listarTodos();

        // 2. Pasar datos a la vista
        request.setAttribute("articulos", articulos);

        // 3. Delegar renderizado a la vista
        request.getRequestDispatcher("/index.jsp")
               .forward(request, response);
    }
}
```

Listing 21: Controlador - Servlet

Flujo de datos en MVC:



5.4. Object Pool Pattern

5.4.1. Definición

El patrón Object Pool mantiene un conjunto de objetos reutilizables listos para usar, evitando la creación y destrucción costosa de objetos [13].

5.4.2. Implementación en ConexionBD

```java
public class ConexionBD {
    private final List<PooledConnection> connectionPool;
    private final int maxConnections = 10;

    // Obtener conexión del pool
    public Connection getConexion() throws SQLException {
        // 1. Buscar conexión disponible en el pool
        for (PooledConnection pc : connectionPool) {
```

```
            if (!pc.isInUse()) {
                pc.setInUse(true);
                return pc.getConnection();
            }
        }

        // 2. Si no hay disponible y no estamos al máximo, crear nueva
        if (connectionPool.size() < maxConnections) {
            Connection conn = createNewConnection();
            PooledConnection pc = new PooledConnection(conn);
            connectionPool.add(pc);
            return conn;
        }

        // 3. Si el pool está lleno, esperar o fallar
        throw new SQLException("Pool lleno");
    }

    // Devolver conexión al pool (no cerrarla)
    public void cerrarConexion(Connection conn) {
        for (PooledConnection pc : connectionPool) {
            if (pc.getConnection() == conn) {
                pc.setInUse(false);  // Marcar como disponible
                return;
            }
        }
    }
}
```

Listing 22: Pool de Conexiones

Beneficios del Pool:

- Evita overhead de crear/cerrar conexiones
- Reutiliza conexiones existentes
- Limita número máximo de conexiones simultáneas
- Mejora rendimiento significativamente

5.5. Strategy Pattern (Implícito)

5.5.1. Aplicación

Aunque no está explícitamente nombrado, el uso de interfaces DAO implementa el patrón Strategy:

```
// La estrategia (algoritmo) de persistencia es intercambiable
public class ArticuloServlet {
    private IArticuloDAO strategy;  // Estrategia de persistencia

    public void setStrategy(IArticuloDAO newStrategy) {
        this.strategy = newStrategy;
    }

    // Usa la estrategia sin saber cuál es
    List<Articulo> articulos = strategy.listarTodos();
}

```

```
// Estrategia concreta 1: MySQL
class MySQLArticuloDAO implements IArticuloDAO { ... }

// Estrategia concreta 2: PostgreSQL (potencial)
class PostgresArticuloDAO implements IArticuloDAO { ... }
```

Listing 23: Strategy mediante interfaces

6. Ejemplos Concretos de Código

6.1. Flujo Completo: Listar Artículos

Veamos cómo todos estos principios y patrones trabajan juntos en un caso de uso real:

6.1.1. Paso 1: Usuario Solicita Ver Artículos

```
GET http://localhost:8080/AdvancedFinalProject/articulos
```

Listing 24: URL solicitada

6.1.2. Paso 2: Filtros Interceptan (SoC)

```
// Filtro 1: Establecer codificación UTF-8
public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain) {
    request.setCharacterEncoding("UTF-8");
    response.setCharacterEncoding("UTF-8");
    chain.doFilter(request, response);  // Siguiente filtro
}
```

Listing 25: CharacterEncodingFilter.java

```
// Filtro 2: Verificar disponibilidad de BD
public void doFilter(...) {
    if (!isDatabaseAvailable()) {
        response.sendRedirect("/setup");  // Redirigir a configuración
        return;
    }
    chain.doFilter(request, response);  // Continuar
}
```

Listing 26: DatabaseCheckFilter.java

6.1.3. Paso 3: Servlet Controlador (MVC - Controller)

```
public class ArticuloServlet extends HttpServlet {
    // DIP: Depende de abstracción
    private IArticuloDAO articuloDAO;

    @Override
    public void init() {
        // Inyección de dependencia
        this.articuloDAO = new MySQLArticuloDAO();
    }

    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {

        String action = request.getParameter("action");

```

```
        if (action == null || action.equals("listar")) {
            listarArticulos(request, response);
        } else if (action.equals("ver")) {
            verArticulo(request, response);
        }
    }

    private void listarArticulos(HttpServletRequest request,
                                 HttpServletResponse response)
            throws ServletException, IOException {
        try {
            // 1. Obtener datos del modelo (DAO)
            List<Articulo> articulos = articuloDAO.listarTodos();

            // 2. Pasar datos a la vista
            request.setAttribute("articulos", articulos);

            // 3. Delegar renderizado a JSP
            request.getRequestDispatcher("/index.jsp")
                   .forward(request, response);

        } catch (SQLException e) {
            request.setAttribute("error",
                "Error al cargar artículos: " + e.getMessage());
            request.getRequestDispatcher("/error.jsp")
                   .forward(request, response);
        }
    }
}
```

Listing 27: ArticuloServlet.java

6.1.4. Paso 4: DAO Accede a Datos (DAO Pattern)

```
public class MySQLArticuloDAO implements IArticuloDAO {
    private final ConexionBD conexionBD;

    public MySQLArticuloDAO() {
        // Singleton: Una sola instancia de ConexionBD
        this.conexionBD = ConexionBD.getInstancia();
    }

    @Override
    public List<Articulo> listarTodos() throws SQLException {
        List<Articulo> articulos = new ArrayList<>();

        // SQL con JOIN para obtener nombre del autor
        String sql =
            "SELECT a.id, a.titulo, a.contenido, " +
            "       a.fecha_publicacion, a.autor_id, " +
            "       COALESCE(u.nombre, 'Desconocido') as autor_nombre " +
            "FROM articulos a " +
            "LEFT JOIN usuarios u ON a.autor_id = u.id " +
            "ORDER BY a.fecha_publicacion DESC";

        Connection conn = null;
        try {
```

```
            // Object Pool: Obtener conexión del pool
            conn = conexionBD.getConexion();

            // PreparedStatement para prevenir SQL Injection
            try (PreparedStatement stmt = conn.prepareStatement(sql);
                 ResultSet rs = stmt.executeQuery()) {

                while (rs.next()) {
                    Articulo articulo = new Articulo();
                    articulo.setId(rs.getInt("id"));
                    articulo.setTitulo(rs.getString("titulo"));
                    articulo.setContenido(rs.getString("contenido"));
                    // ... mapear resto de campos
                    articulos.add(articulo);
                }
            }
        } finally {
            // IMPORTANTE: Devolver conexión al pool, no cerrarla
            if (conn != null) {
                conexionBD.cerrarConexion(conn);
            }
        }

        return articulos;
    }
}
```

Listing 28: MySQLArticuloDAO.java

6.1.5. Paso 5: Vista Renderiza (MVC - View)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<!DOCTYPE html>
<html>
<head>
    <title>Odally | Blog</title>
</head>
<body>
    <h1>Artículos Recientes</h1>

    <!-- JSTL: DRY - No repetir scriptlets -->
    <c:forEach var="articulo" items="${articulos}">
        <article>
            <h2>${articulo.titulo}</h2>
            <p class="meta">
                Por ${articulo.autorNombre}
                el ${articulo.fechaPublicacionFormateada}
            </p>
            <p>${articulo.contenido}</p>
            <a href="articulos?action=ver&id=${articulo.id}">
                Leer más
            </a>
        </article>
    </c:forEach>
</body>
</html>
```

Listing 29: index.jsp

6.2. Análisis del Flujo

1. **SRP:** Cada clase tiene una responsabilidad
   - Filtros: Solo filtrado
   - Servlet: Solo orquestación
   - DAO: Solo persistencia
   - Vista: Solo presentación
2. **OCP:** Podemos cambiar la implementación DAO sin tocar el servlet
3. **LSP:** Cualquier implementación de `IArticuloDAO` funcionaría
4. **ISP:** Interfaces específicas (`IArticuloDAO`, no un DAO genérico gigante)
5. **DIP:** Servlet depende de `IArticuloDAO` (abstracción), no de MySQL
6. **DRY:**
   - Pool de conexiones en un solo lugar
   - JSTL elimina repetición de scriptlets
7. **SoC:** Separación clara entre capas (Filter, Controller, DAO, View)
8. **Patrones:**
   - Singleton: `ConexionBD`
   - DAO: Capa de persistencia
   - MVC: Arquitectura general
   - Object Pool: Pool de conexiones

7. Conclusiones

7.1. Beneficios Obtenidos

La aplicación rigurosa de principios SOLID, principios de arquitectura de paquetes y patrones de diseño en el proyecto Odally | Blog ha resultado en:

1. **Mantenibilidad:** El código es fácil de entender y modificar gracias a SRP y separación de responsabilidades
2. **Extensibilidad:** Se pueden agregar nuevas funcionalidades (como cambiar de BD) sin modificar código existente (OCP)
3. **Testabilidad:** Las interfaces permiten crear mocks para pruebas unitarias (DIP)
4. **Reutilización:** Los paquetes están organizados para reutilización (REP, CRP)
5. **Robustez:** Pool de conexiones con reintentos automáticos mejora la resiliencia
6. **Seguridad:** PreparedStatements previenen SQL injection, contraseñas hasheadas, filtros de autenticación

7.2. Lecciones Aprendidas

- Los principios SOLID no son solo teoría – se aplican directamente en código real
- La inversión de dependencias (DIP) es clave para testear y mantener código
- Los patrones de diseño resuelven problemas recurrentes de forma elegante
- La simplicidad (KISS) es preferible a la complejidad innecesaria
- La separación de responsabilidades (SoC) hace el código más comprensible

7.3. Trabajo Futuro

Posibles mejoras al proyecto:

1. **Inyección de Dependencias:** Usar un framework como CDI para inyectar DAOs en lugar de `new MySQLArticuloDAO()`
2. **Testing:** Agregar pruebas unitarias con JUnit y Mockito aprovechando las interfaces
3. **Pool Profesional:** Migrar a HikariCP o Apache DBCP para pool de conexiones más robusto
4. **Seguridad:** Implementar bcrypt en lugar de SHA-256 para contraseñas
5. **Logging:** Usar SLF4J + Logback en lugar de `System.out.println`

Referencias

[1] Martin, Robert C. (2000). *Design Principles and Design Patterns*. Object Mentor, Inc.

[2] Martin, Robert C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

[3] Martin, Robert C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

[4] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[5] Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

[6] Liskov, Barbara H.; Wing, Jeannette M. (1994). *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems, Vol. 16, No. 6.

[7] Hunt, Andrew; Thomas, David (2019). *The Pragmatic Programmer: Your Journey to Mastery* (20th Anniversary Edition). Addison-Wesley.

[8] Beck, Kent (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

[9] Dijkstra, Edsger W. (1982). *On the Role of Scientific Thought*. Selected Writings on Computing: A Personal Perspective.

[10] Raymond, Eric S. (2003). *The Art of Unix Programming*. Addison-Wesley.

[11] Alur, Deepak; Crupi, John; Malks, Dan (2003). *Core J2EE Patterns: Best Practices and Design Strategies* (2nd Edition). Prentice Hall.

[12] Reenskaug, Trygve (1979). *THING-MODEL-VIEW-EDITOR: An Example from a Planning System*. Xerox PARC Technical Note.

[13] Kircher, Michael; Jain, Prashant (2004). *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley.